Prediction and Indexing of Moving Objects with Unknown Motion Patterns

Yufei Tao[†] Christos Faloutsos[‡] Dimitris Papadias[§] Bin Liu[§]

[†]Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

[‡]Department of Computer Science
Carnegie Mellon University
Forbes Avenue, Pittsburgh, USA
christos@cs.cmu.edu

[§]Department of Computer Science
HKUST
Clear Water Bay, Hong Kong
{*dimitris, liubin*}*@cs.ust.hk*

ABSTRACT

Existing methods for prediction in spatio-temporal databases assume that objects move according to linear functions. This severely limits their applicability, since in practice movement is more complex, and individual objects may follow drastically different motion patterns. In order to overcome these problems, we first introduce a general framework for monitoring and indexing moving objects, where (i) each object computes individually the function that accurately captures its movement and (ii) a server indexes the object locations at a coarse level and processes queries using a filter-refinement mechanism. Our second contribution is a novel recursive motion function that supports a broad class of non-linear motion patterns. The function does not presume any a-priori movement but can postulate the particular motion of each object by examining its locations at recent timestamps. Finally, we propose an efficient indexing scheme that facilitates the processing of predictive queries without false misses.

1. INTRODUCTION

Spatio-temporal databases that manage information about objects moving in two- (or higher) dimensional spaces are important for several emerging applications including traffic supervision, flight control, mobile computing, etc. A large part of the related research (e.g., [KGT99, AAE00, HKTG02, SJLL02, CC02, TSP03, HKT03]) focuses on *predictive queries*, which forecast the objects that will qualify a spatial condition at some future time based on the current knowledge (e.g., "which flights are expected to enter the airspace of California in the next 10 minutes"). In order to avoid frequent location updates, the database stores the *motion function* $\boldsymbol{o}(t)$ of each object o, which returns its location at any future timestamp t. With a single exception ([AA03] that investigates theoretical indexes on non-linear trajectories for nearest neighbor search), existing work on spatio-temporal prediction assumes linear movement. Specifically, $\boldsymbol{o}(t) = \boldsymbol{o}(t_o) + \boldsymbol{v}_o(t-t_o)$ where t_o is the last timestamp that object o issued an update, $\boldsymbol{o}(t_o)$ is the location of o at time t_o, and $\boldsymbol{v}_o$ denotes its current velocity (constant since t_o). Both $\boldsymbol{o}$ and $\boldsymbol{v}_o$ are d-dimensional vectors (where d is the dimensionality of the data space) since they capture the information of o on all axes. An update is necessary whenever $\boldsymbol{v}_o$ (i.e., the speed or direction of the movement) changes.

SIGMOD 2004, June 13-18, 2004, Paris, France.

1.1 Motivation

While in practice most movements are not linear, the adoption of the linear model is often justified in two ways: (i) it avoids the complications of arbitrary motion patterns and permits the analysis of several interesting spatio-temporal problems [TP02, ISS03] that otherwise would be very difficult or intractable, and (ii) piece-wise linear segments can approximate (virtually, to arbitrary precision) any curve, which seems to suggest that linear prediction trivially covers the forecasting of other motion types. This, unfortunately, is not true. Figure 1.1a explains the inadequacy of linear prediction by showing the locations (black dots) of an object o, moving along a curvature during 6 timestamps. Consider a query issued at time 1 that asks for the location of o at the next 4 timestamps. According to the linear velocity of o at the query time (computed using $\boldsymbol{o}(0)$ and $\boldsymbol{o}(1)$), the predicted positions (white dots) deviate from the actual ones significantly. Similar observations hold for a predictive query issued at timestamp 2, where estimation is based on $\boldsymbol{o}(1)$ and $\boldsymbol{o}(2)$. *Although piece-wise line segments can approximate curves, they cannot be effectively applied for prediction, especially in the distant future*. Further, note that the object needs to issue an update at *every single timestamp* to reflect the continuous direction changes.

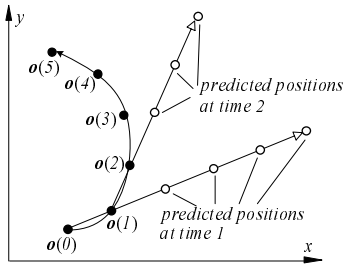


Figure 1.1: Failure of linear prediction

We use the term *motion type* or *pattern* to denote the general form of the motion function (e.g., linear, quadratic, circular), as opposed to its specific parameters, e.g., two linearly moving objects follow the same pattern although their direction or speed may differ. An obvious attempt to alleviate the above problems is to apply a more complex motion type. For example, one could use a quadratic function [AA03], $\boldsymbol{o}(t)=\boldsymbol{o}(t_o)+\boldsymbol{v}_o(t-t_o)+\frac{1}{2}\boldsymbol{a}_o(t-t_o)^2$, where $\boldsymbol{a}_o$ is the acceleration vector of o. Although this model captures linearity as a special case (and therefore has higher applicability), it still cannot represent the curve of Figure 1.1. Further, even if a function describing the particular curve can be obtained, it would not be able to capture other objects in the system, which may follow totally different patterns. In general, due to the enormous diversity of motion types, trying to formulate a universal motion function that captures all possible trajectories would be

unrealistic.

1.2 Contributions

This paper contains three important contributions that solve the problems of prediction on arbitrary motion patterns. We first propose a general client-server architecture for answering typical spatio-temporal queries on objects with unknown, and possibly variable, movement types. In particular, each client (object) computes individually the function that best captures its motion, while a server indexes the object locations at a *coarse* level using a fixed function, which is common for all objects. Queries are processed through a filter-refinement mechanism, where candidate objects are contacted, if necessary, for more refined information. Our solution enables the application of spatio-temporal access methods specifically designed for a particular type, to arbitrary movements without any false misses. Thus, the framework bridges naturally the existing research on the linear model with the non-linear world, permitting the utilization of previous results.

The second contribution is the *recursive motion function*, a systematic and mathematically rigorous technique that expresses, in a concise format, a large number of movement types (e.g., polynomials, ellipses, sinusoids, etc.). In particular, unlike conventional a-priori motion functions that represent location as a closed formula with respect to time, a recursive function relates an object's location to those of the recent past. This broadens the set of expressible trajectories, because they are no longer constrained by a certain default motion; rather, the recurrence causes each object to adapt itself, producing the next location according to the trend of its own movement.

The third contribution is the STP-tree (*spatio-temporal prediction tree*), an access method for indexing the expected trajectories (at the server). Unlike existing indexes which target a specific motion type (most often linear) known in advance, the STP-tree can be used for polynomial functions of any degree. In the special case where the degree is 1, the STP-tree degenerates to the TPR-tree (the current state-of-the-art, discussed in Section 2). Compared to the TPR-tree (or other indexes for linear movement), the STP-tree reduces the number of location updates and false hits during query processing by delivering more accurate approximation of actual object movements.

The rest of the paper is organized as follows. Section 2 surveys previous work, focusing on the TPR-tree and related structures due to their immediate relevance to the STP-tree. Section 3 proposes the general framework for spatio-temporal prediction and overviews our methods. Section 4 discusses computation of unknown movements using the recursive motion function. Section 5 describes the STP-tree as well as the construction and query processing algorithms. Section 6 verifies the effectiveness of the proposed techniques through extensive experiments and Section 7 concludes the paper with directions for future work.

2. RELATED WORK

Among the several spatio-temporal structures [TUW98, KGT99, AAE00] that focus on predictive query processing, the most popular one is the TPR-tree. Because the TPR-tree is an adaptation of the R*-tree, we first provide a description of this structure in Section 2.1. Then, Section 2.2 overviews the TPR-tree and Section 2.3 the TPR*-tree, which improves the original method with enhanced algorithms.

2.1 The R*-tree

The R*-tree [BKSS90] aims at indexing static multi-dimensional data. Figure 2.1 shows a two-dimensional example where 10 rectangles (*a*,*b*,…,*j*) are clustered according to their spatial proximity into 4 leaf nodes N_1,…,N_4, which are then recursively grouped into nodes N_5, N_6 that become the entries of the root. Each entry is represented as a minimum bounding rectangle (MBR). Specifically, the MBR of a leaf entry denotes the extent of an object, while the MBR of a non-leaf entry (e.g., N_1) tightly bounds all the MBRs (i.e., *a*,*b*,*c*) in its child node. The R*-tree algorithms aim at minimizing the following *penalty metrics*: (i) the area, (ii) the perimeter of each MBR, (iii) the overlap between two MBRs (e.g., N_1,N_2) in the same node, and (iv) the distance between the centroid of an MBR (e.g., *a* in Figure 2.1) and that of the node (e.g., N_1) containing it. As discussed in [PSTW93], minimization of these metrics decreases the probability that a node is accessed by a range query.

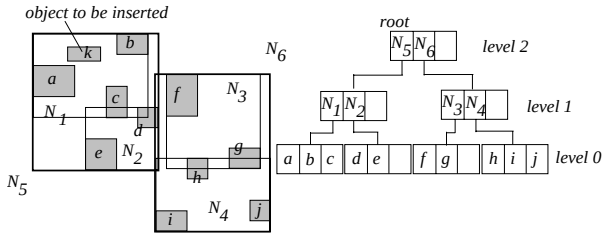


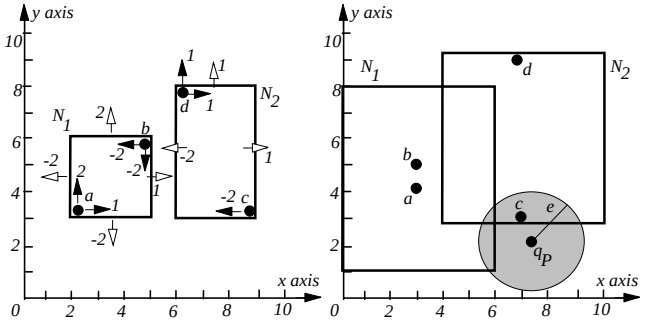
Figure 2.1: An R*-tree

Given a new entry, the insertion algorithm decides, at each level of the tree, the branch to follow in a greedy manner. Assume that we insert an object *k* into the tree in Figure 2.1. At the root level, the algorithm chooses the entry whose MBR needs the least area enlargement to cover *k*; N_5 is selected because its MBR does not need to be enlarged, while that of N_6 must be expanded considerably. Then, at the next level (i.e., child node of N_5), the algorithm chooses the entry whose MBR enlargement leads to the smallest overlap increase among the sibling entries in the node. Note that different metrics are considered at level 1 (leaf nodes are at level 0) and higher levels. An *overflow* occurs if the leaf node reached (i.e., N_1 in the example) is full (i.e., it already contains the maximum number of entries). In this case the algorithm attempts to remove and re-insert a fraction of the entries in the node, trying to avoid a split if any entry could be assigned to other nodes. The set of entries to be re-inserted are those whose centroid distances are among the largest 30%. In Figure 2.1, *b* is selected since its centroid is the farthest from that of N_1 (compared to *a*,*k*,*c*). Node splitting is performed if the overflow persists after the re-insertion (e.g., *b* is re-inserted back to N_1 in Figure 2.1, causing N_1 to overflow again). The deletion algorithm of the R*-tree is relatively simple. First, the leaf node that contains the entry to be removed is identified. If the node does not generate an underflow (i.e., it does not violate the minimum node utilization), the deletion terminates. Otherwise, the underflow is handled by simply re-inserting all the entries of the node, using the regular insertion algorithm. Both overflows and underflows may propagate to upper levels, which are handled in the same way.

2.2 The TPR-tree

The TPR-(*time parameterized R*-) tree [SJLL00] extends the basic concepts of the R*-tree to linearly moving objects. We illustrate its functionality using the four points (*a*, *b*, *c*, *d*) in Figure 2.2a. The black dots illustrate their position at the current time 0, and

the arrows (values) indicate the direction (speed) of their movements. For example, the velocity value of *a* along the x-dimension is 1, while that on the y-axis equals 2, i.e., *a* moves northeast with slope 2 and speed $\sqrt{5}$. A negative velocity indicates that the object moves towards the minus direction on the corresponding axis. Figure 2.2b shows the object positions at timestamp 1, estimated (based on the linear model) according to their velocities at the current time. A node in the TPR-tree is represented using a MBR and a *velocity bounding vector* (VBV), which enclose the location and velocities of the covered objects, respectively. For example, the VBV of node N_1 in Figure 2.2a is {–2, 1, –2, 2} where the first/second number equals the smallest/largest object velocity on the x-dimension (decided by *b*, *a* respectively). Similarly, the third and fourth values (–2, 2) capture the object velocities on the y-axis (VBV velocities are depicted by white arrows). The extent of a node grows with time (at the speed indicated by its VBV) so that at *any* future timestamp it contains the locations of the underlying objects, although it is not necessarily tight. For example, in Figure 2.2b, both N_1 and N_2 are considerably larger than the corresponding minimum rectangles. The node extents for some future time are computed dynamically based on the MBRs and VBVs at the current time.



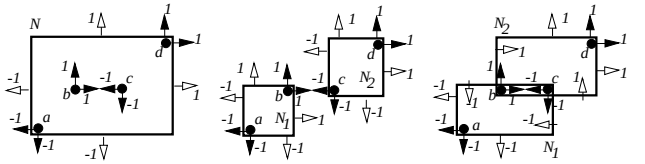
(a) MBRs & VBRs at time 0 (b) MBRs at time 1

Figure 2.2: Entry representations in a TPR-tree

Given a point q_P, a distance *e*, and a future time interval q_T, a *predictive range* query finds all objects *o* such that $dist(\boldsymbol{o},q_P,q_T) \leq e$, i.e., the distance between $\boldsymbol{o}(t)$ and q_P is equal to, or smaller than *e* at some timestamp $t \in q_T$. The shaded circle of Figure 2.2b corresponds to a range query with $q_T = [1,1]$. A TPR-tree answers such queries by accessing all nodes (N_1 and N_2 in Figure 2.2b) that intersect the circle (q_P,*e*) during q_T. The same approach is used for (window) queries with rectangular regions. TPR-trees are optimized for queries with q_T in the interval $[T_C, T_C+H]$, where the *reference time* T_C is the current timestamp, and the *horizon H* determines how far the tree should "see" in the future. The update algorithms are exactly the same as those of the R*-tree, by simply replacing the four penalty metrics with their *integral* counterparts. Specifically, the area (or perimeter) of an entry *N* equals $\int_{T_C}^{T_C+H} A(N,t)dt$ (or $\int_{T_C}^{T_C+H} P(N,t)dt$), where $A(N,t)$ (or $P(N,t)$) returns the area (perimeter) of *N* at time *t*. Similarly, the overlap (or the centroid distance) between two MBRs N_1 and N_2 is computed as $\int_{T_C}^{T_C+H} OVR(N_1,N_2,t)dt$ (or $\int_{T_C}^{T_C+H} CDist(N_1,N_2,t)dt$), where $OVR(N_1,N_2,t)$ (or $CDist(N_1,N_2,t)$) returns the overlapping area (centroid distance) between N_1 and N_2 at time *t*. These integrals are solved into closed formulae [SJLL00]. Saltenis and Jensen [SJ02] describe a method for improving the performance of TPR-trees when the time of the next update for each object is known in advance.

2.3 The TPR*-tree

The TPR*-tree [TPS03] follows the general update methodology of TPR- (and R-) trees, but includes some enhanced heuristics. When an object *o* is inserted, the TPR*-tree first identifies the leaf *N* that will accommodate *o* with the *choose path* algorithm, which, instead of the greedy traversal of the R*- and TPR*-trees, follows the path that leads to the minimization of the penalty metrics in a branch-and bound manner. If *N* is full, a set of entries, selected by *pick worst*, are removed from *N* and re-inserted. These objects are such that, their removal minimizes the MBR and VBV of the parent node. Any node that overflows during the re-insertion is split using *sorting split*, which decides the entry distribution by sorting all the spatial and velocity dimensions. Consider Figure 2.3a, where node *N* overflows, assuming that the node capacity is 3, and the minimum node utilization is 2. The algorithm first sorts the objects by their x-coordinates (i.e., the sorted order is *a*, *b*, *c*, *d*), and then groups the first two entries *a*, *b* into node N_1, and *c*, *d* into N_2. Based on the MBRs and VBVs of N_1 and N_2 (shown in Figure 2.3b), the algorithm computes the sum of their integrated perimeters, and uses it as the *penalty* of this split. Next, the algorithm performs another sorting on the x-velocities of these objects, (i.e., order *a*, *c*, *b*, *d*), distributes the entries into N_1 and N_2 accordingly (Figure 2.3c), and computes the split penalty. The same process is repeated on the y-axis and the final entry distribution is the one (among the four possible) with the smallest penalty.



(a) Node to split (b) Split on x-axis (c) Split on x-velocity

Figure 2.3: The split algorithm of the TPR*-tree

3. SYSTEM OVERVIEW

This section provides a general framework for spatio-temporal prediction and describes the proposed methodology at an abstract level, before proceeding with the details in subsequent sections. We assume a client-server architecture, where each moving client (object) *o* can measure its location through a GPS device (at discrete timestamps) and has some processing power and memory, so that it maintains the most recent locations and continuously revises its individual motion function $\boldsymbol{o}(t)$. The server collects information from objects over time, indexes their expected trajectories, and answers predictive range queries[1] issued by the users.

While different objects can follow distinct patterns, the server assumes the *same* motion type for *all* objects, which differs from their individual functions. Thus, the server maintains only imprecise information and has to process queries in a *filter-refinement* manner. Specifically, at the filter step, it first retrieves (i) a set of objects that definitely satisfy the query predicates, and (ii) a set of candidates (which may or may not qualify the query).

[1] For simplicity we focus on static range queries. Nevertheless, our methods also capture moving queries as well as other types of predictive queries (e.g., nearest neighbor search).

During refinement, the server contacts the objects of the second set, which evaluate the query conditions using their own (precise) motion function, and inform the server accordingly. Thus, the *correctness* of a query is defined with respect to individual objects' motion functions. Figure 3.1 shows the overall system architecture.

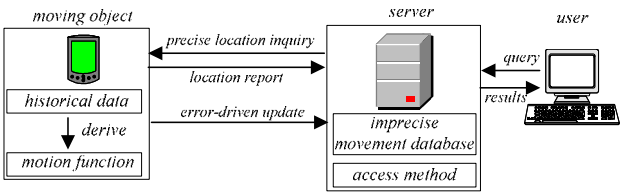


Figure 3.1: System architecture

Similar to [SJLL00, TSP03], we use a *horizon* parameter H, and optimize the system for queries whose intervals q_T fall in $[T_C, T_C+H]$ where T_C is the reference time. Let $\boldsymbol{m}^S_o(t)$ be the motion function for object o at the server. In order to guarantee the correctness of query results, each object transmits (i) the parameters of $\boldsymbol{m}^S_o(t)$, formulated according to its own motion function $\boldsymbol{o}(t)$ and (ii) a maximum distance d_o (called the *horizon bound*) between $\boldsymbol{o}(t)$ and $\boldsymbol{m}^S_o(t)$ during the time interval $[T_C, T_C+H]$. To illustrate this, consider Figure 3.2 which builds on the example of Figure 1.1 assuming that T_C =1, H=4 and that the server accepts only linear movements. The object computes the parameters of $\boldsymbol{m}^S_o(t)$ based on its location at timestamps 0 and 1, and estimates its future positions using both $\boldsymbol{o}(t)$ and $\boldsymbol{m}^S_o(t)$ (shown with black and white dots, respectively) at the next 4 timestamps. In this case, d_o equals the distance $d(5)$ between $\boldsymbol{o}(5)$ and $\boldsymbol{m}^S_o(5)$. Consider now a range query (at time 1) asking for all objects in the circle centering at q_P with radius e during interval q_T = [4,5]. At the filter step, the server retrieves the objects such that $dist(\boldsymbol{m}^S_o,q_P,q_T) \leq e+d_o$, where $dist(\boldsymbol{m}^S_o,q_P,q_T)$ equals the minimum distance between $\boldsymbol{m}^S_o$ and q_P during [4,5] (in this example $dist(\boldsymbol{m}^S_o,q_P,q_T) = dist(\boldsymbol{m}^S_o(4),q_P)$). Although o does not satisfy the query (both $\boldsymbol{o}(4)$ and $\boldsymbol{o}(5)$ are outside the range), it passes the filter step and becomes a candidate. Therefore, it is requested to evaluate the query based on its own prediction for $\boldsymbol{o}(4)$ are $\boldsymbol{o}(5)$, which will cause its elimination from the actual query result.

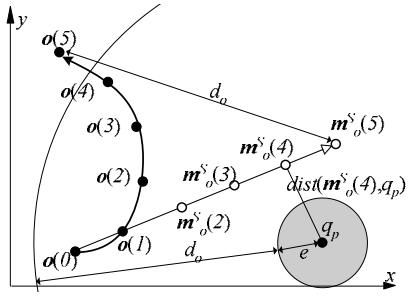


Figure 3.2: Coordinating object and server functions (T_C =1)

Subsequent object updates follow an *error-driven* strategy. In particular, each object o records the last transmitted values of $\boldsymbol{m}^S_o(t)$ and d_o, and issues an update whenever this information cannot correctly capture its *current* movement. Continuing the example, at the next timestamp T_C =2, o re-computes its location during the next H (=4) timestamps using both $\boldsymbol{m}^S_o(t)$ and $\boldsymbol{o}(t)$, as shown in Figure 3.3. The distance $d(6)$ between $\boldsymbol{m}^S_o(6)$ and $\boldsymbol{o}(6)$ is larger than the current horizon bound d_o and the server must be informed in order to avoid false misses. Thus, the object derives a new linear function $\boldsymbol{m}^{S'}_o(t)$ (based on its position at timestamps 1 and 2) and (using this function) revises the value of d_o to $d'(6)$ (i.e., the distance between $\boldsymbol{m}^{S'}_o(6)$ and $\boldsymbol{o}(6)$). The new $\boldsymbol{m}^{S'}_o(t)$ and d_o are then sent to the server. At timestamp T_C =3, the distance between $\boldsymbol{o}(7)$ and $\boldsymbol{m}^{S'}_o(7)$ is smaller than the last reported value of d_o (=$d'(6)$); therefore, the object does not issue an update and the server assumes that it continues moving according to the previous motion parameters. This update policy trivially captures the case where the object's motion pattern changes over time (since the object revises $\boldsymbol{o}(t)$ at each timestamp).

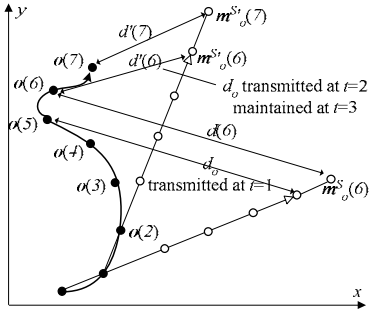


Figure 3.3: Error-driven update policy (T_C =2 and T_C =3)

Finally, it should be pointed out that the existing methods for supporting predictive spatio-temporal queries constitute special cases of our framework, where the motion types are identical for all objects (and the server), and known in advance. As discussed in the introduction, this severely restricts their applicability to practical problems. On the other hand, *the separation of motion functions (at the object and server sides) and the filter-refinement mechanism (i) lift these restrictions and, at the same time (ii) they permit the application of previous spatio-temporal research on linear movement*. For instance, the TPR-tree (or any other spatio-temporal index) can be used directly to index the object representations at the server, while selectivity estimation techniques [CC02, HKT03, TSP03] can predict the output size of the filter step.

The above discussion serves as a high-level description, omitting, however, two fundamental issues: (i) the derivation of the individual motion functions for each object, and (ii) the development of a novel access method, which reduces the number of false misses by tuning the motion "resolution" depending on the application needs. These issues are addressed in Sections 4 and 5, respectively.

4. Derivation of Motion Functions

Let o be an object whose motion type is *unknown*. Given the actual locations of o at the h most recent timestamps, our objective is to derive a motion function that (i) can correctly capture all these locations, and (ii) can predict the future trajectory of o, by following the *tendency* of the movement (i.e., linear, quadratic, curving, etc.). Section 4.1 proposes a novel recursive motion function, which is significantly more powerful in terms of expressive power than the existing closed functions of time. Then, Section 4.2 presents a methodology for deciding the function parameters.

4.1 Recursive functions and motion matrices

Although individual trajectories may vary significantly, most motion types demonstrate a self-similar behavior, in the sense that the current location can be usually predicted from those in the recent past. This is most obvious for linear movements with fixed velocity $\boldsymbol{v}$, where the location $\boldsymbol{o}(t)$ of o equals $\boldsymbol{o}_o(t-1)$ +

$[\boldsymbol{o}_o(t-1)-\boldsymbol{o}_o(t-2)] = 2\boldsymbol{o}_o(t-1)-\boldsymbol{o}_o(t-2)$ (note that $\boldsymbol{o}_o(t-1)-\boldsymbol{o}_o(t-2)$ gives exactly the velocity vector $\boldsymbol{v}$). Interestingly, it turns out that, for a large number of movements, $\boldsymbol{o}_o(t)$ can be represented as a *linear function* of $\boldsymbol{o}_o(t\text{-}1)$, $\boldsymbol{o}_o(t\text{-}2)$, …. We demonstrate this with two examples: *accelerative* and *circular* movements.

Example 4.1: Consider the motion function $\boldsymbol{o}(t) = \boldsymbol{o}(t_o)+\boldsymbol{v}_o(t-t_o)+\frac{1}{2}\boldsymbol{a}_o(t-t_o)^2$, where $\boldsymbol{o}(t_o)$ is the location of o at the reference time t_o, $\boldsymbol{v}_o$ is the velocity vector at t_o, and $\boldsymbol{a}_o$ is the acceleration. This function can be easily re-written in the form $\boldsymbol{o}(t)=\boldsymbol{c_0}+\boldsymbol{c_1}\cdot t+\boldsymbol{c_2}\cdot t^2$ where $\boldsymbol{c_0}, \boldsymbol{c_1}, \boldsymbol{c_2}$ are $d\times1$ vectors of constants, and d is the dimensionality of the data space. Taking the discrete differentiation (with respect to t) on both sides of the equation results in $\boldsymbol{o}(t)-\boldsymbol{o}(t-1) = \boldsymbol{c_1}+2\boldsymbol{c_2}\cdot t$. A second differentiation yields the linear form: $[\boldsymbol{o}(t)-\boldsymbol{o}(t-1)]-[\boldsymbol{o}(t-1)-\boldsymbol{o}(t-2)]=2\boldsymbol{c_2}$. The constant can be eliminated by yet another differentiation, leading to:

$$\boldsymbol{o}(t)=3\boldsymbol{o}(t-1)-3\boldsymbol{o}(t-2)+\boldsymbol{o}(t-3) \quad (4\text{-}1)$$

In general, it is easy to verify that any polynomial motion function of degree D can be converted to a linear recurrence after D+1 differentiations. ■

Example 4.2: If a 2D object moves with angular speed[2] ω on a circle that centers at (c_1, c_2) with radius r, its coordinates $\boldsymbol{o}(t).x_1$ and $\boldsymbol{o}(t).x_2$ at time t are given by: $\boldsymbol{o}(t).x_1 = c_1+ r\cdot\cos(\omega\cdot t)$ and $\boldsymbol{o}(t).x_2 = c_2+r\cdot\sin(\omega\cdot t)$. Following a derivation similar to Example 4.1, we have:

$$\boldsymbol{o}(t)=\begin{bmatrix} 1+\cos(\omega) & -\sin(\omega) \\ \sin(\omega) & 1+\cos(\omega) \end{bmatrix}\boldsymbol{o}(t-1)+\begin{bmatrix} -\cos(\omega) & \sin(\omega) \\ -\sin(\omega) & -\cos(\omega) \end{bmatrix}\boldsymbol{o}(t-2) \quad (4\text{-}2)$$ ■

Motivated by these observations, we introduce the following recursive motion function:

$$\boldsymbol{o}(t) = \mathbf{C_1}\cdot\boldsymbol{o}(t-1) + \mathbf{C_2}\cdot\boldsymbol{o}(t-2) + \ldots + \mathbf{C}_f\cdot\boldsymbol{o}(t-f) \quad (4\text{-}3)$$

where $\mathbf{C}_i$ ($1\le i\le f$) is a $d\times d$ constant matrix, and f is a system parameter called *retrospect*. Equation 4-3 is much more powerful than conventional motion functions, since it can express an extensive number of simple and complex movement types (by varying $\mathbf{C}_i$), including polynomials, ellipses, sinusoids, etc. Obviously, the expressive power increases with f (in our experiments, the value f=5 already models accurately all the motion types tested).

We define the *motion state* $\boldsymbol{s}_o(t)$ of an object o at time t as a vector $\{\boldsymbol{o}(t), \boldsymbol{o}(t-1), \ldots, \boldsymbol{o}(t-f+1)\}$ enclosing its location at the f most recent timestamps (i.e., $\boldsymbol{s}_o(t)$ is a $(d\cdot f)\times1$ vector). Equation 4-3 becomes much friendlier (especially for performing future prediction, as elaborated shortly) when transformed to the equivalent matrix form:

$$\boldsymbol{s}_o(t) = \mathbf{K}_o\cdot\boldsymbol{s}_o(t-1) \quad (4\text{-}4)$$

where $\mathbf{K}_o$ is a constant $(d\cdot f)\times(d\cdot f)$ *motion matrix* for o. We use the following notations: (i) k_{ij} is the element of $\mathbf{K}_o$ at the i-th row and j-th column ($1\le i, j\le d\cdot f$), (ii) $\boldsymbol{k_{i*}}$ is the i-th row, i.e., a $1\times(d\cdot f)$ vector, and (iii) $\boldsymbol{o}(t).x_i$ is the co-ordinate of $\boldsymbol{o}(t)$ on the i-th dimension. For clarity, in the following discussion we usually illustrate the properties of $\mathbf{K}_o$ for d=2 (i.e., 2D space) and f=2 (i.e., each state captures the two most recent locations), before extending to the general case. For the simple case, equation 4-4 becomes:

$$\begin{bmatrix} \boldsymbol{o}(t).x_1 \\ \boldsymbol{o}(t).x_2 \\ \boldsymbol{o}(t-1).x_1 \\ \boldsymbol{o}(t-1).x_2 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{o}(t-1).x_1 \\ \boldsymbol{o}(t-1).x_2 \\ \boldsymbol{o}(t-2).x_1 \\ \boldsymbol{o}(t-2).x_2 \end{bmatrix} \quad (4\text{-}5)$$

Notice that we can immediately decide the third and fourth rows of $\mathbf{K}_o$, e.g., $\boldsymbol{k_{3*}}=\{1,0,0,0\}$ is the only possible choice for making $\boldsymbol{o}(t-1).x_1 \equiv \boldsymbol{k_{3*}}\cdot\boldsymbol{s}_o(t-1)$. In general, we have:

$$\begin{aligned} &k_{ij}=0 \text{ for } i\ge d+1 \text{ and } i\ne j+d \\ &k_{ij}=1 \text{ for } i\ge d+1 \text{ and } i=j+d \end{aligned} \quad (4\text{-}6)$$

Equivalently, the motion matrix has only $d^2\cdot f$ unknowns (i.e., the number of unknowns in $\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C}_f$ of equation 4-3).

A very important observation is that *all objects that follow the same movement type have identical motion matrices*. For example, the motion matrices $\mathbf{K}_{line}$ (for linear movement) and $\mathbf{K}_{circle}$ (for circular movement) are:

$$\mathbf{K}_{line} = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 2 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (4\text{-}7)$$

$$\mathbf{K}_{circle} = \begin{bmatrix} 1+\cos(\omega) & -\sin(\omega) & -\cos(\omega) & \sin(\omega) \\ \sin(\omega) & 1+\cos(\omega) & -\sin(\omega) & \cos(\omega) \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (4\text{-}8)$$

As another example, the quadratic polynomial function (for 2D spaces) requires a motion matrix with six rows and columns. By equation 4-2, the first two rows of the matrix are $\begin{bmatrix} 3 & 0 & -3 & 0 & 1 & 0 \\ 0 & 3 & 0 & -3 & 0 & 1 \end{bmatrix}$ (the other rows are set according to equation 4-6).

Motion matrices indicate whether there are dependencies among the dimensions. For instance, the linear model (and polynomial functions in general) is *axis-independent*, e.g., the x_1-coordinate of a point is decided solely by its velocity on the x_1-axis, and is unrelated to those on the other dimensions, because $k_{12}=k_{14}=k_{21}=k_{23}=0$. On the other hand, circular movement is axis-*dependent* because of the non-zero k_{12} and k_{21}; for example, $\boldsymbol{o}(t).x_1 = (1+\cos(\omega))\cdot\boldsymbol{o}(t-1).x_1 - \sin(\omega)\cdot\boldsymbol{o}(t-1).x_2 - \cos(\omega)\cdot\boldsymbol{o}(t-1).x_1 + \sin(\omega)\cdot\boldsymbol{o}(t-1).x_2$, (i.e., the x_1-coordinate of o is related to the x_2-coordinate of its location at the previous timestamp). In general, a type of movement is axis-independent if and only if[3]:

$$k_{ij}=0, \forall 1\le i\le d \text{ and } j\%d \ne i\%d \quad (4\text{-}9)$$

Another crucial observation is that all motion matrices should satisfy the *translation rule*:

$$\boldsymbol{s}_o(t) = \mathbf{K}_o\cdot\boldsymbol{s}_o(t-1) \Rightarrow \boldsymbol{s}_o(t) + \boldsymbol{c} = \mathbf{K}_o\cdot(\boldsymbol{s}_o(t-1) + \boldsymbol{c}) \quad (4\text{-}10)$$

where the *translation vector* $\boldsymbol{c}$ is a $(d\cdot f)\times1$ vector in the form $\boldsymbol{c}=(c_1,c_2,\ldots,c_d, \ldots, c_1,c_2,\ldots,c_d)$ (i.e., $c_1,c_2,\ldots,c_d$ repeated f times), for arbitrary constants $c_1, c_2,\ldots, c_d$. Since the intuition behind this equation is not obvious, we give a concrete example. Assume that the coordinates of a linearly-moving 2D object are $\boldsymbol{o}(0)=\{1,1\}$, $\boldsymbol{o}(1)=\{2,3\}$, $\boldsymbol{o}(2)=\{3,5\}$ respectively (i.e., its x_1- and x_2-velocities are 1 and 2). If we set up a state with f=2 locations, i.e., $\boldsymbol{s}_o(1)=\{\boldsymbol{o}(0), \boldsymbol{o}(1)\}$ and $\boldsymbol{s}_o(2)=\{\boldsymbol{o}(1), \boldsymbol{o}(2)\}$, then they are captured by the motion matrix of $\mathbf{K}_{line}$ (given in equation 4-7), namely,

[2] Angular speed is the angle (with respect to the center of the circle) traveled by the object in a time unit.

[3] Operator % returns the residue after the modulo operation (e.g., 5%3=2, 6%3=0).

$\boldsymbol{s}_o(2)=\mathbf{K}_{line}\cdot\boldsymbol{s}_o(1)$. Now consider that we translate all locations $\boldsymbol{o}(0)$, $\boldsymbol{o}(1)$, $\boldsymbol{o}(2)$ by the same offsets c_1=10, c_2=20 on the two dimensions respectively, obtaining $\boldsymbol{o}'(0)$ = {10,21}, $\boldsymbol{o}'(1)$ = {12,23}, $\boldsymbol{o}'(2)$ = {13,25}. Obviously $\boldsymbol{o}'(0)$, $\boldsymbol{o}'(1)$, $\boldsymbol{o}'(2)$ still form a line and hence should be captured by $\mathbf{K}_{line}$, meaning that, for the resulting states $\boldsymbol{s}_o'(1)$ = {$\boldsymbol{o}'(0)$, $\boldsymbol{o}'(1)$} and $\boldsymbol{s}_o'(2)$ = {$\boldsymbol{o}'(1)$, $\boldsymbol{o}'(2)$}, we have $\boldsymbol{s}_o'(2)$ = $\mathbf{K}_{line}\cdot\boldsymbol{s}_o'(1)$. Notice that the effect of the translation on states $\boldsymbol{s}_o(0)$ and $\boldsymbol{s}_o(1)$ is such that, each new state $\boldsymbol{s}_o'(i)$ ($1\le i\le 2$) equals $\boldsymbol{s}_o'(i)+\boldsymbol{c}$, where $\boldsymbol{c}$ = {c_1,c_2,c_1,c_2}, leading to the fact that $\boldsymbol{s}_o(2)+\boldsymbol{c}$ = $\mathbf{K}_{line}\cdot(\boldsymbol{s}_o(1)+\boldsymbol{c})$. In general, the translation rule indicates that, if $\mathbf{K}_o$ captures a trajectory o_1, it also expresses any other trajectory o_2 translated from o_1.

Equation 4-10 imposes some important constraints on the elements of a motion matrix, i.e., the translation rule implies:

$$\boldsymbol{c} = \mathbf{K}_o\cdot\boldsymbol{c} \quad (4\text{-}11)$$

where $\boldsymbol{c}$ is any translation vector. In case that $\mathbf{K}_o$ is a 2×2 matrix, it can be shown that $\mathbf{K}_o$ satisfies equation 4-11 if and only if *all* the following conditions hold: (i) $k_{11}+k_{13}=1$, (ii) $k_{12}+k_{14}=0$, (iii) $k_{21}+k_{23}=0$, and (iv) $k_{22}+k_{24}=1$ (these conditions become obvious by writing each component on the left-hand side $\boldsymbol{c}$ as a function of the coefficients of $\mathbf{K}_o$ and the right-hand side $\boldsymbol{c}$). Observe that each condition is on the sum of elements of $\mathbf{K}_o$ in the same row, interleaved by d (=2 in this case). For arbitrary values of d and f, equation 4-11 holds if and only if:

$$\begin{array}{c} k_{ij} + k_{i(j+d)} + k_{i(j+2d)} + \ldots + k_{i(j+f\cdot d)} = 1 \text{ if } j=i \bmod d, \text{ and} \\ k_{ij} + k_{i(j+d)} + k_{i(j+2d)} + \ldots + k_{i(j+f\cdot d)} = 0 \text{ otherwise} \end{array} \quad (4\text{-}12)$$

The last, but not least, important property of $\mathbf{K}_o$ is that, based on the motion state $\boldsymbol{s}(T_C)$ at the current time T_C, we can efficiently[4] compute the state at T_C+t (i.e., t timestamps later) as:

$$\boldsymbol{s}_o(T_C+t) = \mathbf{K}_o^{\,t}\cdot\boldsymbol{s}_o(T_C) \quad (4\text{-}13)$$

Evidently, $\mathbf{K}_o^{\,t}$ (also a $(d\cdot f)\times(d\cdot f)$ matrix) must satisfy equation 4-11, or specifically: $\boldsymbol{c} = \mathbf{K}_o^{\,t}\cdot\boldsymbol{c}$ for any translation vector $\boldsymbol{c}$. This is implied by 4-11, since $\mathbf{K}_o^{\,t}\cdot\boldsymbol{c} = \mathbf{K}_o^{\,t-1}\cdot(\mathbf{K}_o\cdot\boldsymbol{c}) = \mathbf{K}_o^{\,t-1}\cdot\boldsymbol{c} = \ldots = \mathbf{K}_o\cdot\boldsymbol{c} = \boldsymbol{c}$.

Finally, note that a motion matrix does not specify the concrete function parameters, which are implicitly determined by state $\boldsymbol{s}_o(t-1)$. For example, the velocity $\boldsymbol{v}$ in the linear model is decided by $\boldsymbol{o}(t-1)-\boldsymbol{o}(t-2)$, whereas $\mathbf{K}_{line}$ simply indicates a line trajectory. Thus, the prediction task is now reduced to finding the correct $\mathbf{K}_o$ (i.e., determining the corresponding movement *type*) after which the parameters of this movement are *automatically finalized* by $\boldsymbol{s}_o(t-1)$. In the next section, we provide a technique to compute $\mathbf{K}_o$ and the individual recursive function for each object, using its locations at the recent past.

4.2 Motion estimation

Let $\boldsymbol{l}_o(t)$ be the actual location of object o at time t. Given $\boldsymbol{l}_o(T_C-h+1)$, $\boldsymbol{l}_o(T_C-h+2)$, …, $\boldsymbol{l}_o(T_C)$ at the h most recent timestamps, our goal is to decide a function $\boldsymbol{o}(t)$ that minimizes the summed squared distances (*ssd*) between the computed (by $\boldsymbol{o}$) and actual locations:

$$ssd = \sum_{t=T_C-h+1}^{T_C} |\boldsymbol{l}_o(t)-\boldsymbol{o}(t)|^2 \quad (4\text{-}14)$$

where $|\boldsymbol{l}_o(t)-\boldsymbol{o}(t)|^2=\sum_{i=1}^{d}[\boldsymbol{l}_o(t).x_i-\boldsymbol{o}(t).x_i]^2$ and $\boldsymbol{l}_o(t).x_i$ is the coordinate of $\boldsymbol{l}_o(t)$ on the i-th dimension. Equation 4-14 provides a metric to evaluate the quality of any motion function $\boldsymbol{o}(t)$. For instance, in the linear model minimizing *ssd* is equivalent to finding the best fitting line (going through all $\boldsymbol{o}(i)$, $T_C-h+1\le i\le T_C$), which can be easily computed using the *least squared error* method. On the other hand, for recursive motion functions the objective is to decide the optimal $\mathbf{K}_o$ that minimizes equation 4-11, using the properties discussed in Section 4.1.

Recall that $\mathbf{K}_o$ involves $d^2\cdot f$ unknowns, which constitute the first d rows from $\boldsymbol{k}_{1*}$ to $\boldsymbol{k}_{d*}$. The h locations define $h-f+1$ motion states (where f is the retrospect)[5], or specifically:

$$\begin{array}{l} \boldsymbol{s}_o(T_C-h+f)=\{\boldsymbol{l}_o(T_C-h+f), \boldsymbol{l}_o(T_C-h+f-1),\ldots, \boldsymbol{l}_o(T_C-h+1)\} \\ \boldsymbol{s}_o(T_C-h+f+1)=\{\boldsymbol{l}_o(T_C-h+f+1, \boldsymbol{l}_o(T_C-h+f),\ldots, \boldsymbol{l}_o(T_C-h+2)\} \\ \ldots \\ \boldsymbol{s}_o(T_C)=\{\boldsymbol{l}_o(T_C), \boldsymbol{l}_o(T_C-1),\ldots, \boldsymbol{l}_o(T_C-f+1)\} \end{array} \quad (4\text{-}15)$$

Therefore, the optimal $\mathbf{K}_o$ should satisfy the following $h-f$ equations:

$$\boldsymbol{s}_o(t) = \mathbf{K}_o\cdot\boldsymbol{s}_o(t-1) \text{ for } T_C-h+f+1\le t\le T_C \quad (4\text{-}16)$$

We solve the equations corresponding to a row (from $\boldsymbol{k}_{1*}$ to $\boldsymbol{k}_{d*}$) at a time. It suffices to elaborate the solution for $\boldsymbol{k}_{1*}$ (i.e., an $1\times(d\cdot f)$ vector) as the extension to the other rows is straightforward. Consider, for simplicity, the case d=f=2, where each equation in the set 4-16 is in the form of formula 4-5. Since we focus on $\boldsymbol{k}_{1*}$, we extract the relevant part of the formula into equation 4-17:

$$\boldsymbol{l}(t).x_1=k_{11}\cdot\boldsymbol{l}(t-1).x_1+k_{12}\cdot\boldsymbol{l}(t-1).x_2+k_{13}\cdot\boldsymbol{l}(t-2).x_1+k_{14}\cdot\boldsymbol{l}(t-2).x_2 \quad (4\text{-}17)$$

The above equation can be written into the following form which also holds for general d and f (recall that $\boldsymbol{s}(i-1)$ is a $(d\cdot f)\times 1$ vector):

$$\boldsymbol{l}(t).x_1 = \boldsymbol{k}_{1*}\cdot\boldsymbol{s}(t-1) \quad (4\text{-}18)$$

Since a similar formula is obtained from all $(h-f)$ equations in 4-16, we obtain $h-f$ (linear) equations for the $d\cdot f$ variables in $\boldsymbol{k}_{1*}$ and organize them into a matrix format (notice that $\boldsymbol{k}_{1*}$ now appears on the right of the multiplication sign):

$$\begin{bmatrix} \boldsymbol{s}(T_C-1)^T \\ \boldsymbol{s}(T_C-2)^T \\ \ldots \\ \boldsymbol{s}(T_C-h+f)^T \end{bmatrix}\cdot\boldsymbol{k}_{1*} = \begin{bmatrix} \boldsymbol{l}(T_C).x_1 \\ \boldsymbol{l}(T_C-1).x_1 \\ \ldots \\ \boldsymbol{l}(T_C-h+f+1).x_1 \end{bmatrix} \quad (4\text{-}19)$$

where the superscript T stands for transpose. Let us denote:

$$\mathbf{S} = \begin{bmatrix} \boldsymbol{s}(T_C-1)^T \\ \boldsymbol{s}(T_C-2)^T \\ \ldots \\ \boldsymbol{s}(T_C-h+f)^T \end{bmatrix}, \text{and } \boldsymbol{l} = \begin{bmatrix} \boldsymbol{l}(T_C).x_1 \\ \boldsymbol{l}(T_C-1).x_1 \\ \ldots \\ \boldsymbol{l}(T_C-h+f+1).x_1 \end{bmatrix} \quad (4\text{-}20)$$

where $\mathbf{S}$ is a $(h-f)\times(d\cdot f)$ matrix and $\boldsymbol{l}$ a $(h-f)\times 1$ vector. A robust solution of the linear equation set $\mathbf{S}\cdot\boldsymbol{k}_{1*}=\boldsymbol{l}$ can be obtained using singular value decomposition (SVD) [PFTV02]. SVD decomposes $\mathbf{S}$ into $\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{V}^{\mathrm{T}}$, where $\mathbf{U}$ is a $(h-f)\times(d\cdot f)$ column-

[4] $\mathbf{K}_o^{\,t}$ can be computed in $O(\log_2(t))$. For example, if t=13, we first calculate $\mathbf{K}_o^{\,2}$, $\mathbf{K}_o^{\,4}$ (obtained by $\mathbf{K}_o^{\,2}\cdot\mathbf{K}_o^{\,2}$), $\mathbf{K}^8$ (=$\mathbf{K}_o^{\,4}\cdot\mathbf{K}_o^{\,4}$) and finally $\mathbf{K}_o^{\,13}=\mathbf{K}_o\cdot\mathbf{K}_o^{\,4}\cdot\mathbf{K}_o^{\,8}$.

[5] We assume that $f\le h$, namely, the number of locations in one state cannot exceed the total number of historical locations maintained.

orthogonal matrix, $\mathbf{W}$=[diag(w_1,w_2,…,$w_{d\cdot f}$)] a ($d\cdot f$)×($d\cdot f$) diagonal matrix with positive elements w_1, w_2, …, $w_{d\cdot f}$ on the diagonal, and $\mathbf{V}$ a ($d\cdot f$)×($d\cdot f$) orthogonal matrix (i.e., $\mathbf{V}\cdot\mathbf{V}^{\mathrm{T}}=\mathbf{I}$, the identity matrix). Thus, $\boldsymbol{k_{1*}}$ is solved as:

$$\boldsymbol{k_{1*}} = \mathbf{V}\cdot[\mathrm{diag}(1/w_1, 1/w_2,\ldots, 1/w_{d\cdot f})]\cdot(\mathbf{U}^{\mathrm{T}}\cdot\boldsymbol{l}) \quad (4\text{-}21)$$

Two points worth mentioning are: (i) if $|w_i|$ is sufficiently close to zero (typically, $w_i<10^{-12}$), its corresponding element in equation 4-21, is simply replaced by 0; (ii) when $h-f\leq d\cdot f$, equation 4-21 yields a $\boldsymbol{k_{1*}}$ which *strictly satisfies* equation 4-19. On the other hand, if $h-f>d\cdot f$, we have more equations than unknowns, in which case SVD produces a $\boldsymbol{k_{1*}}$ that *minimizes* $|\mathbf{S}\cdot\boldsymbol{k_{1*}}-\boldsymbol{l}|^2$ [PFTV02]. This is exactly what we need in order to minimize *ssd*, since equation 4-14 can be re-written as:

$$ssd = \sum_{i=1}^{d}\sum_{t=T_C-h+1}^{T_C} |\boldsymbol{l}_o(t).x_i - \boldsymbol{o}(t).x_i|^2 \quad (4\text{-}22)$$

$|\mathbf{S}\cdot\boldsymbol{k_{1*}}-\boldsymbol{l}|^2=\sum_{t=T_C-H+1}^{T_C}|\boldsymbol{l}_o(t).x_1-\boldsymbol{o}(t).x_1|^2$, i.e., i=1 in the above equation. Similarly, the solution for the i-th row $\boldsymbol{k_{i*}}$ of $\mathbf{K}_o$ minimizes $\sum_{t=T_C-H+1}^{T_C}|\boldsymbol{l}_o(t).x_i-\boldsymbol{o}(t).x_i|^2$; thus the overall $\mathbf{K}_o$ optimizes *ssd*. Note that, if the object locations $\boldsymbol{l}_o(T_C-h+1)$, $\boldsymbol{l}_o(T_C-h+2)$, …, $\boldsymbol{l}_o(T_C)$ fall perfectly on a curve represented by $\mathbf{K}_o$, equation 4-21 will *always* yield a perfect solution (i.e., strictly satisfying equation 4-19), even if $h-f>d\cdot f$ (i.e., in this case, some equations are unnecessary due to their linear dependence on the others).

We close this section with another heuristic that produces a robust motion matrix $\mathbf{K}_o$ (prior to performing SVD). The heuristic is based on equation 4-12, which gives d additional constraints for solving $\boldsymbol{k_{1*}}$ from equation 4-19. Let us introduce d translation vectors ($1\leq i\leq d$) $\boldsymbol{c^i}$={$c_1,c_2,\ldots,c_d$, $c_1,c_2,\ldots,c_d$, …, $c_1,c_2,\ldots,c_d$} (as mentioned in Section 4.1, $c_1,c_2,\ldots,c_d$ is repeated f times) with c_j=1 ($1\leq j\leq d$ and j=i) and c_j=0 ($1\leq j\leq d$ and $j\neq i$). For example, $\boldsymbol{c^1}$={1,0, …,0, 1,0,…,0, …, 1,0,…,0} and $\boldsymbol{c^d}$={0,0,…,1, 0,0,…, 1,…,0, 0,…,1}. We modify equation 4-19 by adding d rows to $\mathbf{S}$ and $\mathbf{L}$ as follows:

$$\begin{bmatrix} \boldsymbol{s}(T_C-1)^T \\ \boldsymbol{s}(T_C-2)^T \\ \ldots \\ \boldsymbol{s}(T_C-h+f)^T \\ c^1 \\ \ldots \\ c^d \end{bmatrix}\cdot k_{1*} = \begin{bmatrix} \boldsymbol{l}(T_C).x_1 \\ \boldsymbol{l}(T_C-1).x_1 \\ \ldots \\ \boldsymbol{l}(T_C-h+f+1).x_1 \\ 1 \\ \ldots \\ 0 \end{bmatrix} \quad (4\text{-}23)$$

In general, when solving $\boldsymbol{k_{i*}}$, the i-th number added in $\mathbf{L}$ (i.e., the right hand side vector of equation 4-23) should be 1 and the other ones should be 0. After obtaining $\mathbf{K}_o$, we can predict the motion state at any future timestamp with equation 4-16, using the most recent state $\boldsymbol{s}(T_C)$.

5. THE SPATIO-TEMPORAL PREDICTION TREE

As discussed in Section 3, we can choose any function for representing trajectories at the server, including the linear model, in which case the TPR-tree can be applied directly. A coarse function, however, may lead to poor performance, because it is likely to generate (i) large horizon bounds d_o (and therefore numerous false hits during query processing) and (ii) frequent error-driven updates (and large communication costs). On the other hand, although a refined function provides more accurate approximation alleviating the above problems, it involves a large number of parameters, requires more storage space, and is more difficult to manipulate. Our solution constitutes a trade-off between the two cases. In particular, we assume that the motion of all objects at the server side is represented as a polynomial function $\boldsymbol{m}^S{}_o$ with arbitrary degree D:

$$\boldsymbol{m}^S{}_o(t)=\boldsymbol{o}(t_o)+\boldsymbol{c_1}\cdot(t-t_o)+\boldsymbol{c_2}\cdot(t-t_o)^2+\ldots+\boldsymbol{c_D}\cdot(t-t_o)^D \quad (5\text{-}1)$$

where (i) t_o is the reference time of object o (i.e., the last timestamp that o updated $\boldsymbol{m}^S{}_o(t)$), (ii) $\boldsymbol{o}(t_o)$ is its location at t_o, and (iii) $\boldsymbol{c_1}$, $\boldsymbol{c_2}$, ..., $\boldsymbol{c_D}$ are constant d×1 vectors decided based on the precise motion function $\boldsymbol{o}(t)$. Equation 5-1 can be transformed into the matrix form.

$$\boldsymbol{m}^S{}_o(t) = \boldsymbol{o}(t_o)+\mathbf{B}\cdot\{t-t_o, (t-t_o)^2, ..., (t-t_o)^D\}^{\mathrm{T}} \quad (5\text{-}2)$$

where *the polynomial matrix* $\mathbf{B}$ is a d×D constant matrix whose D columns are $\boldsymbol{c_1}$, $\boldsymbol{c_2}$, ..., $\boldsymbol{c_D}$, and $\{t-t_o, (t-t_o)^2, ..., (t-t_o)^D\}$ is a 1×D vector (equation 5-2 uses its transpose). Following the terminology of Section 4, b_{ij} is the element of $\mathbf{B}$ on the i-th row and j-th column, and $\boldsymbol{b_{i*}}$ (a 1×D vector) is the i-th row.

In addition to their flexibility for tuning the motion resolution (by setting the appropriate value of D), polynomial functions are axis-independent, which, as shown shortly, facilitates the design of efficient spatio-temporal access methods. Section 5.1 discusses the computation of $\boldsymbol{m}^S{}_o(t)$ based on the individual object function $\boldsymbol{o}(t)$, and clarifies the *error-driven* update algorithm. Section 5.2 presents the spatio-temporal prediction tree (STP-tree), used to support general polynomial movements at the server. Finally, Section 5.3 discusses processing of predictive range queries.

5.1 Optimal polynomial derivation

Given the degree D of $\boldsymbol{m}^S{}_o(t)$ (which is set in advance based on the aforementioned trade-off) and an individual motion function $\boldsymbol{o}(t)$ (computed using the techniques of Section 4), our goal is to compute the parameters of $\boldsymbol{m}^S{}_o(t)$ that minimize the squared sum of the distances between $\boldsymbol{m}^S{}_o(t)$ and $\boldsymbol{o}(t)$ during the next H timestamps:

$$ssd^S = \sum_{t=T_C}^{T_C+H}|\boldsymbol{m}^S{}_o(t)-\boldsymbol{o}(t)|^2 = \sum_{i=1}^{d}\sum_{t=T_C}^{T_C+H}|\boldsymbol{m}^S{}_o(t).x_i-\boldsymbol{o}(t).x_i|^2 \quad (5\text{-}3)$$

where $\boldsymbol{m}^S{}_o(t).x_i$ is the coordinate of $\boldsymbol{m}^S{}_o(t)$ on the i-th axis (similarly for $\boldsymbol{o}(t).x_i$). Since $\boldsymbol{m}^S{}_o(t)$ is axis-independent, it suffices to discuss metric $ssd_1{}^S$ on the first dimension:

$$\begin{aligned} ssd_1{}^S &= \sum_{t=T_C}^{T_C+H}|\boldsymbol{m}^S{}_o(t).x_1-\boldsymbol{o}(t).x_1|^2 \\ &= \sum_{t=T_C}^{T_C+H}|\boldsymbol{o}(t_0).x_1+\boldsymbol{b_{1*}}\cdot\{t-t_0,(t-t_0)^2,\ldots,(t-t_0)^D\}^{\mathrm{T}}-\boldsymbol{o}(t).x_1|^2 \end{aligned} \quad (5\text{-}4)$$

The value of $\boldsymbol{b_{1*}}$ that minimizes $ssd_1{}^S$ is the one that best satisfies the following equation:

$$\begin{bmatrix} T_C-t_o & (T_C-t_o)^2 & \ldots & (T_C-t_o)^D \\ T_C+1-t_o & (T_C+1-t_o)^2 & \ldots & (T_C+1-t_o)^D \\ \ldots & \ldots & \ldots & \ldots \\ T_C+H-t_o & (T_C+H-t_o)^2 & \ldots & (T_C+H-t_o)^D \end{bmatrix}\cdot\boldsymbol{b_{1*}}^{\mathrm{T}} = \begin{bmatrix} \boldsymbol{o}(T_C).x_1-\boldsymbol{o}(t_o).x_1 \\ \boldsymbol{o}(T_C+1).x_1-\boldsymbol{o}(t_o).x_1 \\ \ldots \\ \boldsymbol{o}(T_C+H).x_1-\boldsymbol{o}(t_o).x_1 \end{bmatrix} \quad (5\text{-}5)$$

The solution (similar to equation 4-19) is again based on SVD and omitted. The point that requires clarification concerns the frequency of the above computations. At every timestamp, each object first calculates its individual function $\boldsymbol{o}(t)$ and the maximum difference d_{max} between $\boldsymbol{o}(t)$ and $\boldsymbol{m}^S{}_{old}(t)$ for the next H timestamps, where $\boldsymbol{m}^S{}_{old}(t)$ is its previous representation at the server. According to the error-driven update policy of Section 3, if

$d_{max} \leq d_o$ (last reported value of the horizon bound), the object does not need to issue an update (since it will not cause a false miss) and the computation of $\boldsymbol{m}^S_o(t)$ is avoided. On the other hand, if $d_{max} > d_o$, the object calculates the new $\boldsymbol{m}^S_o(t)$ and d_o (this time based on $\boldsymbol{m}^S_o(t)$) and transmits them to the server. Figure 5.1 summarizes these procedures.

Algorithm object_update
Input $\boldsymbol{l}_o(T_C-h+1)$, $\boldsymbol{l}_o(T_C-h+2)$, …, $\boldsymbol{l}_o(T_C)$: object locations at the h most recent timestamps, $\boldsymbol{m}^S_{old}(t)$ and d_{old}: last transmitted values for the server motion function and the horizon bound
1. compute current motion function $\boldsymbol{o}(t)$
2. $d_{max}=max\{dist(\boldsymbol{o}(t), \boldsymbol{m}^S_{old}(t))$, for $T_C \leq t \leq T_C+H\}$
3. if $d_{max} > d_{old}$
4. compute the new optimal $\boldsymbol{m}^S_o(t)$
5. $d_o = max\{dist(\boldsymbol{o}(t), \boldsymbol{m}^S_o(t))$, for $T_C \leq t \leq T_C+H\}$
6. transmit $\boldsymbol{m}^S_o(t)$ and d_o to server

End object_update

Figure 5.1: Error-driven update algorithm at each timestamp

The computations of $\boldsymbol{m}^S_o(t)$ and the update messages can be reduced at the expense of the query cost. In particular, instead of the horizon bound d_o, the object can send a larger value in order to delay its potential violation in the future. However, this will increase the number of false hits during query processing. The trade-off depends on the relative frequency of updates and queries. In update-intensive applications a large bound would be appropriate, while in systems with heavy query workload the actual horizon bound should be used.

5.2 STP-tree construction algorithms

The STP-tree constitutes the generalization of the TPR and TPR*-trees to arbitrary polynomial functions. Each leaf entry keeps the reference time t_o of an object $\boldsymbol{o}$, its location $\boldsymbol{o}(t_o)$ at t_o, the horizon bound d_o, and the polynomial matrix $\mathbf{B}$. A non-leaf entry e stores (i) a timestamp t_e, which is the maximum of all the reference timestamps of the objects in its sub-tree, (ii) a distance d_e which is the largest horizon bound of its children (iii) two points $\boldsymbol{e}_{min}$ and $\boldsymbol{e}_{max}$ that define the opposite corners of a d-dimensional rectangle enclosing the locations of the children at time t_e, and (iv) two matrices $\mathbf{B}_{min}$ $\mathbf{B}_{max}$ such that each element in $\mathbf{B}_{min}$ ($\mathbf{B}_{max}$) is the smallest (largest) of the corresponding element (i.e., at the same row and column) of all the polynomial matrices in its sub-tree. For D=1 (i.e., linear movement), the representation of the STP-tree degenerates to that of the TPR-tree.

For an intermediate entry e, we define its MBR $e_{MBR}(t)$ at any future timestamp t as the rectangle with opposite corners $\boldsymbol{e}_{min}(t)$, $\boldsymbol{e}_{max}(t)$ computed as follows.

$$\begin{aligned} \boldsymbol{e}_{min}(t) &= \boldsymbol{e}_{min} + \mathbf{B}_{min} \cdot \{t-t_e, (t-t_e)^2, ..., (t-t_e)^D\}^{\mathrm{T}} \\ \boldsymbol{e}_{max}(t) &= \boldsymbol{e}_{max} + \mathbf{B}_{max} \cdot \{t-t_e, (t-t_e)^2, ..., (t-t_e)^D\}^{\mathrm{T}} \end{aligned} \quad (5\text{-}5)$$

It can be shown that $e_{MBR}(t)$ covers the location $\boldsymbol{o}(t)$ of each child object o, which is a prerequisite for avoiding false misses. We further define the *integrated perimeter* of e during the horizon $[T_C, T_C+H]$ as $\sum_{t=T_C}^{T_C+H}(2 \cdot Manh(\boldsymbol{e}_{max}(t) - \boldsymbol{e}_{min}(t)))$, where *Manh* is the *Manhattan norm* of a vector, i.e., for a d-dimensional vector $\boldsymbol{v}$, $Manh(\boldsymbol{v})=\sum_{i=1}^{d}(\boldsymbol{v}.x_i)$, where $\boldsymbol{v}.x_i$ is its coordinate on the i-th axis.

The construction algorithms of the STP-tree follow those of the TPR- and TPR*-trees as reviewed in Section 2. The insertion algorithm first identifies the path (from the root to a leaf) that incurs the minimum increase of the integrated perimeter sum (among all the paths), using the TPR*-tree *choose path* method. If the leaf node overflows, some entries, selected by the TPR*-tree *pick worst* algorithm, are re-inserted, after which any node that overflows is split. Deletion first locates the object (with polynomial matrix $\mathbf{B}$) to be deleted, using its location at the current time, by descending nodes whose $\mathbf{B}_{min}$ and $\mathbf{B}_{max}$ include $\mathbf{B}$ (i.e., each element of $\mathbf{B}$ falls in the range defined by the corresponding elements in $\mathbf{B}_{min}$ and $\mathbf{B}_{max}$). If the deletion causes the corresponding leaf node to underflow (the minimum utilization is set to 40% of the capacity), the node is removed and its entries are re-inserted. Overflows and underflows at the upper levels are treated in the same way; e.g., the removal of a leaf node will delete an entry from its parent, which may lead to an underflow at the next level etc.

The STP-tree also adapts the sorting split method of the TPR*-tree (reviewed in Section 2.3), which obtains D+1 sorted lists on the coefficients of the polynomial motion function (i.e., $\boldsymbol{o}(t_o)$, $\boldsymbol{c}_1$, ..., $\boldsymbol{c}_D$ in equation 5-1) and decides the final split according to the entry distribution that minimizes the integrated perimeter sum. The efficiency of the original algorithm, however, drops as D increases because, for larger D, the effect of each coefficient on the MBR size diminishes; thus, lowering its value may not decrease the MBR size considerably, which is also decided by the other coefficients. Motivated by this, we present an alternative split strategy called the *rank split*, which works as follows. Starting with the first axis, the algorithm sorts the coordinates of the current locations, and associates each object with a *rank*, i.e., its sequence number in the sorted list. To illustrate this, consider Figure 5.2, which shows the location (black dots) of 4 linearly moving points at the current time T_C=0, and their location (white dots) at time H. The sorted list (at time 0) on the x-dimension is $\{o_1, o_3, o_2, o_4\}$, and the ranks of o_1, o_2, o_3, o_4 are 1, 3, 2, 4 respectively.

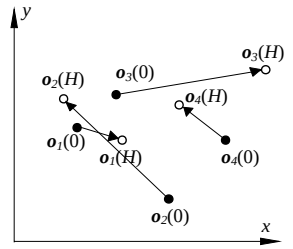


Figure 5.2: Illustration of the STP split algorithm

Next, the algorithm performs another sorting of the coordinates (on the same dimension) at time H and assigns ranks accordingly. The ranks of o_1, o_2, o_3, o_4 in the second sorted list are 2, 1, 4, 3. The total rank of each object is the sum of its ranks in the two lists, i.e., 3, 4, 6 and 7 for o_1, o_2, o_3 and o_4, respectively. Finally, the objects are sorted on their total ranks and distributed accordingly to the new nodes. Assuming that each node contains at least two entries, o_1, o_2 are placed into the first node and o_3, o_4 into the second one. Having finished with the x-axis, the algorithm repeats the above steps on the y-dimension, and the final split is the better of the two distributions. The total number of sorting operations is $3d$ (i.e., independent of D) as opposed to $d \cdot (D+1)$ for the sorting split method. As shown in the experiments, the rank split leads to a more efficient STP-tree for $D \geq 3$. Finally, splitting a non-leaf node is reduced to the above case by taking the centroids of the MBRs.

5.3 Query processing

We now illustrate the algorithm for answering a range query, which constitutes the main target of predictive spatio-temporal

indexes (e.g., TPR- and TPR*- trees aim at the optimization of range search). Given a query centered at q_P, with radius e and interval $q_T=[q_{T-}, q_{T+}]$, we first show how to decide whether a leaf entry is a definite or a candidate result. Figure 5.3 illustrates the approximate locations of two objects o_1, o_2 at timestamps q_{T-} (black dots) and q_{T+} (white) respectively. Particularly, the circle around each dot indicates the area that contains the actual object location at the corresponding timestamp, according to its horizon bound. We can assert that o_1 definitely satisfies the query (without examining its individual motion function), since there exists a timestamp $t \in q_T$ such that $dist(\boldsymbol{m}^S_{o1}(t),q_P)+d_e \le e$, i.e., the circle centering at $\boldsymbol{m}^S_{o1}(t)$ with radius d_{o1} completely falls in the query region. On the other hand, object o_2 is not a definite answer because $dist(\boldsymbol{m}^S_{o2},q_P,q_T)+d_e > e$ for each $t \in q_T$. Nevertheless, o_2 is a candidate since the circle around $\boldsymbol{m}^S_{o2}(q_{T-})$ intersects the search area, implying that the actual location $\boldsymbol{o}_2(q_{T-})$ may be within the area.

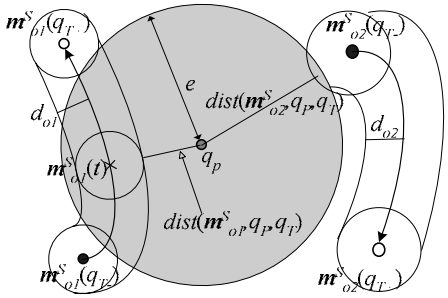


Figure 5.3: Qualifying and candidate objects

Figure 5.4 formally demonstrates the processing method based on the above description. The algorithm starts from the root of the STP-tree and descends each intermediate *entry* satisfying $dist(entry,q_P,q_T) \le e+d_e$, where $dist(entry,q_P,q_T)$ is the minimum distance between $entry_{MBR}$ and q_P during q_T, and d_e is the maximum horizon bound of all the children of *entry*. If $dist(entry,q_P,q_T) > e+d_e$, the entry is pruned, because for all objects o in its sub-tree: $dist(\boldsymbol{o},q_P,q_T) \ge dist(entry,q_P,q_T) > e+d_e \ge e+d_o$. When a leaf entry (e.g., an object o) is encountered, the algorithm, (i) either reports o immediately, if it is a definite result, or (ii) it requests its precise location, if o is a candidate.

Algorithm range_query (q_P, e, q_T, *nd*)
Input: query point q, search radius e, query interval $q_T=[q_{T-}, q_{T+}]$, node *nd* being processed
Output: objects satisfying q
1. if *nd* is an intermediate node
2. for each *entry* in *nd*
3. if $dist(entry,q_P,q_T) \le e+d_e$
4. let *cnd* be the child node of *entry*
5. **range_query**(q_P, e, q_T, *cnd*)
6. else // *nd* is a leaf node
7. for each object o in *nd*
8. if $dist(\boldsymbol{m}^S_{o1},q_P,q_T)+d_o \le e$
9. output o // definite result
10. else if $dist(\boldsymbol{m}^S_{o},q_P,q_T) < e+d_o$ //candidate
11. contact o and wait for $\boldsymbol{o}(t)$
12. if $dist(\boldsymbol{o},q_P,q_T) < e$
13. output o
End range_query

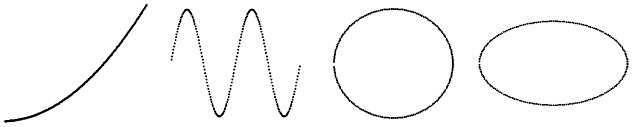
Figure 5.4: The predictive range search algorithm

6. EXPERIMENTS

In this section, we demonstrate the effectiveness of the proposed techniques with an extensive experimental evaluation. Section 6.1 first investigates the expressive power of the recursive motion function, and then Section 6.2 studies the performance of the query processing architecture.

6.1 Motion function evaluation

The first experiment aims at verifying the correctness of the theoretical derivation in Section 4. Towards this, we use four types of mathematical curves, namely, *polynomial*, *sinusoid*, *circle*, *ellipse* as demonstrated in Figure 4.1. In particular, *polynomial* is the composite of two independent movements $x(t)$, $y(t)$ on the x- and y-axes respectively, where $x(t)=v \cdot t$ (i.e., constant velocity v=10) and $y(t)=v \cdot t+(a \cdot t^2)/2$ (i.e., v=10 and constant acceleration a=1). *Sinusoid* is also the result of two independent motions: $x(t)=t$, $y(t)=\sin(\omega \cdot t)$, where ω is fixed to $\pi/50$. The *circle* and *ellipse* are obtained using a fixed angular speed $\omega=\pi/50$; for *circle*, $x(t)=\cos(\omega \cdot t)$, $y(t)=\sin(\omega \cdot t)$, while for *ellipse* $x(t)=100\cos(\omega \cdot t)$, $y(t)=50\sin(\omega \cdot t)$ (i.e., the major and minor axes of the ellipse have lengths 100 and 50, respectively). These curves are *regular*, meaning that they can be captured *precisely* by the proposed recursive motion function (RMF). The first two lines of their motion matrices are: $\begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & 3 & 0 & -3 & 0 & 1 \end{bmatrix}$ (*polynomial*), $\begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1-\alpha-\beta & 0 & \alpha & 0 & \beta \end{bmatrix}$ (*sinusoid*) and $\begin{bmatrix} 1+\cos(\omega) & -2\sin(\omega) & -\cos(\omega) & 2\sin(\omega) \\ \sin(\omega)/2 & 1+\cos(\omega) & -\sin(\omega)/2 & -\cos(\omega) \end{bmatrix}$ (*ellipse*), where $\alpha=\frac{1-\cos(\omega)}{\cos(2\omega)-\cos(\omega)}$ and $\beta=\frac{2\cos^2(\omega)-2\cos(\omega)}{\cos(2\omega)-\cos(\omega)}$. The remaining lines are determined by equation 4-6, whereas the matrix for *circle* is given in equation 4-8. Note that the motion matrices for *polynomial* and *sinusoid* have 6 rows and columns, in contrast to 4 (rows/columns) for *circle* and *ellipse*.



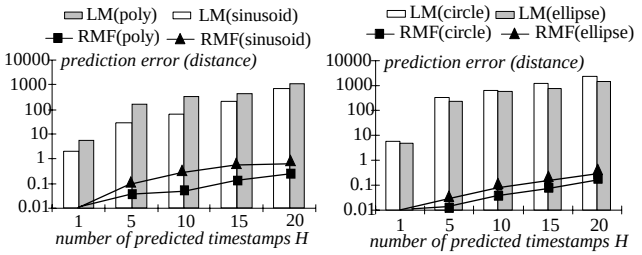
(a) Polynomial (b) Sinusoid (c) Circle (d) Ellipse

Figure 6.1: Movements with known motion matrices

We utilize the techniques described in Section 4 to perform prediction on these curves using, however, the minimum amount of space. Specifically, the retrospect f (i.e., the number of locations in one state) equals the minimum value that is necessary for deriving the motion matrix of the corresponding trajectory. For *polynomial* and *sinusoid*, f=3 (hence, their matrices have 6=3×2 rows, for dimensionality d=2), and for *circle* and *ellipse* f=2. Further, the number of historical locations maintained at all times equals 3f, which is the minimum requirement in order to *correctly* solve the motion function; otherwise, the number of equations is smaller than that of unknowns. We do not explicitly input the motion matrices, but use our algorithm to discover them and then apply equation 4-13 to compute the location of the moving point at H (i.e., horizon) timestamps later. The error is defined by the distance between the computed and actual locations. All the curves are scaled so that their minimum bounding rectangles have length 10000 on each axis. For comparison, we also employ a linear model (denoted as LM in the sequel), where the velocity of the line is decided from the same number of historical locations using the *least square error* method, that minimizes the sum of the squared distances between the actual and computed locations.

Figure 6.2 illustrates the average prediction error over 200 consecutive timestamps for all trajectories. It is clear that, even for the farthest horizon (H=20 timestamps), RMF incurs negligible

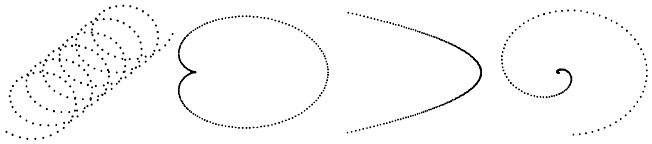
error (i.e., below 1 in a data space 10000^2), confirming the validity of our analysis. As expected, the linear model completely fails to capture these movements, i.e., its largest error is over 2000. The slight degradation of RMF as H increases is caused by some imprecision in the motion matrix, which is "magnified" in the final prediction due to the matrix-power computation (i.e., we need to calculate the power H of the motion matrix). Since LM is erroneous in all cases, we omit it in the following discussion.



(a) *Polynomial* and *sinusoid* (f=3) (b) *Circle* and *ellipse* (f=2)

Figure 6.2: Prediction error for regular movements

The next experiment evaluates the expressive power of the recursive function using *unknown* complex movements that cannot be represented as the linear recursive form (see Figure 6.3) in an obvious manner. Specifically, *spiral* is the composition of the linear and circular movements, i.e., $x(t)=v\cdot t+\cos(\omega\cdot t)$, and $y(t)=v\cdot t+\sin(\omega\cdot t)$. The values of v and ω are set to 10 and $\omega=\pi/50$ respectively. *Peach*, *swirl*, *parabola* are generated using the formula (in the polar space) $r(t)^p=c^p\cdot\cos(c\cdot\omega\cdot t)$, where c (=3) and ω (=π/2) are constants deciding the curve size, and p another constant that determines the motion type (p=0.5, 0.1, –0.5 for *peach*, *swirl*, and *parabola*, respectively[6]). All curves are normalized to the data space with axis length equal to 10000.

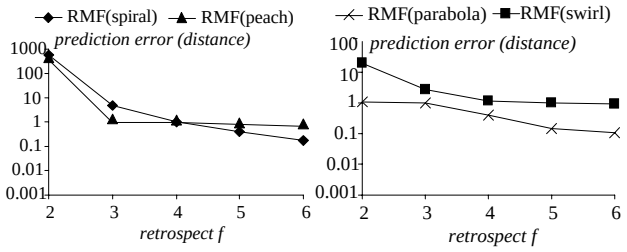


(a) Spiral (b) Peach (c) Parabola (d) Swirl

Figure 6.3: Movements with unknown motion matrices

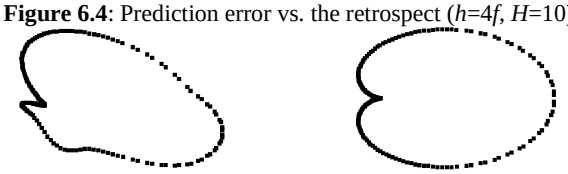
The difference between the unknown and the regular movements of Figure 6.1 is that, for unknown curves, the motion matrix varies (sometimes periodically) with the concrete location of the point (i.e., the matrix captures the tendency of the recent movement). Therefore, we do not know the minimum number f of locations in one state required to capture this movement. Further, unlike the regular motions (where we can restore the entire curve using a few locations), for unknown ones we may need longer history (i.e., larger h) in order to sufficiently train the matrix. Obviously, both f and h are highly related to H (i.e., how many timestamps in the future we wish to predict). For example, a good estimation for the near future may be possible by inspecting only recent history.

In order to study the relationship between these factors, in Figure 6.4 we vary the retrospect f from 2 to 6 and measure the average prediction error of 200 timestamps, after setting h=4f and H to 10. When f=2, the error for *spiral* and *peach* is large (Figure 6.4a), indicating that RMF cannot capture these movements by using a state with only two locations. However, if we allow only one more location in each state, the estimation error drops from around 500 to below 10. The precision continuously improves as each state includes more locations and the error eventually drops below 1. Figure 6.5 shows the trajectories of the predicted *peach* for f=2 and 3, where the improvement is clearly visible. Similar behavior is also observed for *parabola* and *swirl* in Figure 6.4b, except that the error for f=2 is smaller.



(a) *Spiral* and *peach* (b) *Parabola* and *swirl*

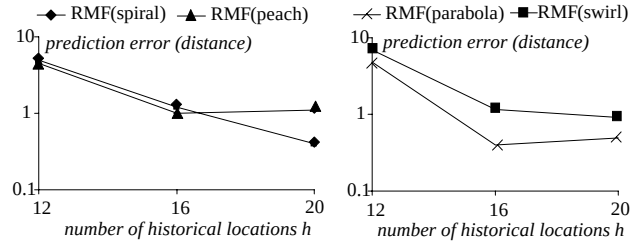
Figure 6.4: Prediction error vs. the retrospect (h=4f, H=10)



(a) f=2 (b) f=3

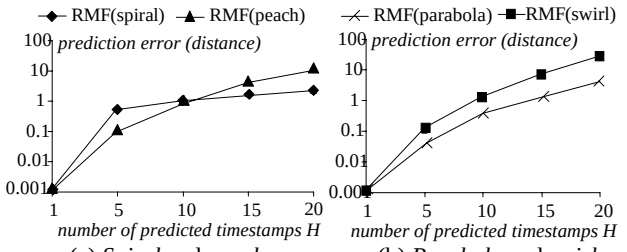
Figure 6.5: Improvements in *peach* with larger retrospect

In Figure 6.6 we fix f=4, H=10 and measure the prediction error as a function of h (from 3f=12 to 5f=20). For *spiral* and *swirl* the accuracy increases monotonically with h, but for *peach* and *parabola*, the error initially decreases, and then grows (nevertheless, the final error is still very small). In these cases, as h becomes larger, it is more difficult for RMF to quickly adapt to the motion changes since it is also influenced by old locations leading to biased estimation. Therefore, blindly increasing the history length does not necessarily enhance the prediction. Figure 6.7 illustrates the error of predicting different timestamps in the future using f=4 and h=16. As in Figure 6.2, the accuracy gradually decreases, but the highest error is still very small (25, or 0.25% of the length of the data axis) even for the farthest horizon.



(a) *Spiral* and *peach* (b) *Parabola* and *swirl*

Figure 6.6: Prediction error vs. the history length (f=4, H=10)



(a) *Spiral* and *peach* (b) *Parabola* and *swirl*

Figure 6.7: Prediction error vs. horizon (f=4, h=16)

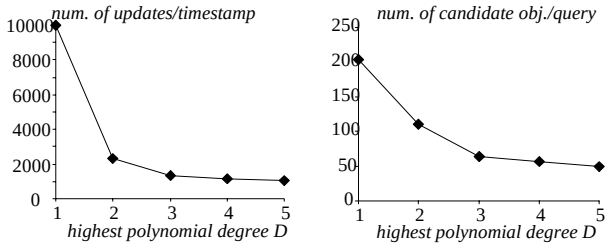
[6] We mention that the parabola thus generated cannot be written as the conventional form $x(t)=y^2(t)+2y(t)+1$ (which can be captured by RMF). Particularly, it is not the composition of two independent x- and y- movements.

6.2 Query processing evaluation

This section evaluates the efficiency of the STP-tree as well as the filter-refinement processing strategy. Due to the lack of real spatio-temporal data, we generate synthetic movements following the methodology of previous work [SJLL00, TPS03]. Specifically, 1k terminals are first randomly selected from a real point dataset ([Tiger]) and the data space is scaled to length 10000 on each axis. Each object decides a source and destination terminal, whose distance is larger than 1000 (i.e., 1/10 of the axis length). The movement from the source to the destination consists of three phases. In the first phase, the object accelerates with constant acceleration 2 on each axis towards the direction of the destination. The accelerative phase lasts 10 timestamps after which the velocity equals 20. In the second phase, the object moves towards the destination along an arc of a circle or a parabola. These two curves are representatives of the regular and unknown movements examined in the previous section. Their parameters are decided so that the object, maintaining constant speed 20, travels roughly 50 timestamps to reach the position where it starts the last decelerating phase. In this phase, the object slows down with acceleration −2 on each dimension before arriving at the destination (the phase lasts 10 timestamps). Then, the object selects the next destination, and repeats the above movements. The dataset contains the simulation of 10k objects over 200 timestamps, i.e., a total of 2 million location changes.

At each timestamp, we execute 50 queries whose locations follow the distribution of the terminals, and the radii of their (circular) search area constitute a workload parameter e (denoted as a percentage of the axis length). Each query is associated with a (future) time interval $q_T=[q_{T-}, q_{T+}]$ such that (i) the interval length *qtlen* is also a parameter, and (ii) q_T randomly distributes in next H=20 timestamps (i.e., the horizon). We measure two types of costs: (i) the number of candidate objects that qualify the filter step (which determines the communication overhead of the refinement step), and (ii) the I/O cost (in terms of the number of node accesses) at the server index. The page size is set to 4k bytes for all experiments. The node capacity of the STP-tree depends on the highest degree D of the polynomial motion function and ranges from 37 (D=5) to 92 (D=1).

The first experiment studies the effect of D on the communication overhead, which involves (i) the error-driven object updates, and (ii) the refinement step of query evaluation. Figure 6.8a illustrates the average number of updates per timestamp, as a function of D. When D=1, *every object must issue an update at each timestamp, resulting in prohibitive overhead*. In this case an object is approximated by the line tangent to the actual motion at its current location. As it deviates from the location, its distance to the tangent line *always* increases (a property for all smooth curves), thus triggering the update at the next timestamp. The number of updates decreases sharply for D=2, indicating that the expressive power of a 2-degree polynomial is significantly higher. As D increases further, however, the update savings diminish and the improvement for $D\geq 3$ is only marginal. This indicates that the assumed (accelerative, circular, parabola) movement types can be adequately approximated using a 3-degree polynomial. Figure 6.8b plots the average number of refinement candidates per query (e=2.5% and *qtlen*=10%). Similar to Figure 6.8a, the improvement drops quickly when D exceeds 3.



(a) Update cost vs. D (b) Refinement cost vs. D (e=2.5%, *qtlen*=10%)

Figure 6.8: Influence of D on the network cost

Figure 6.9 shows the average number of node accesses for answering a query (e=2.5% and *qtlen*=10) as a function of D, for the trees obtained by the *sorting split* and *rank split* algorithms. Interestingly, the query cost decreases until D reaches a certain threshold 3 (for *rank split*) or 2 (for *sorting split*), but increases as D grows further. To understand this, note that a larger D lowers both the node fanout (which deteriorates query performance) and the objects' distance bounds (which improves performance). When D is small, the benefits of increasing the node size outweigh the shortcomings, explaining the initial performance improvement. As D crosses the threshold, however, the shortcomings dominate the benefits. *Rank split* outperforms the *sorting split* for $D\geq 3$, while as discussed in Section 5.2 it incurs smaller computational overhead. For the remainder of the section we apply *rank split* and set D=3, since this value leads to a balanced behavior with respect to update, refinement and query processing costs.

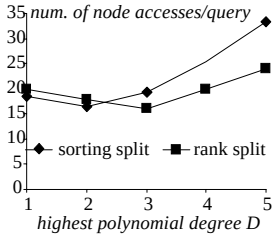
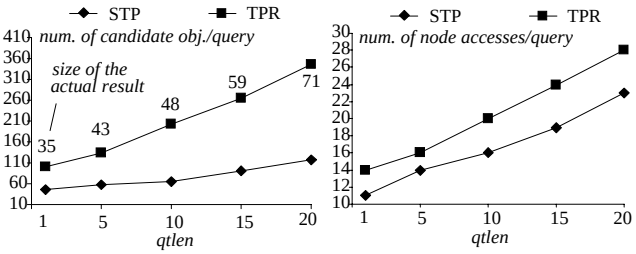


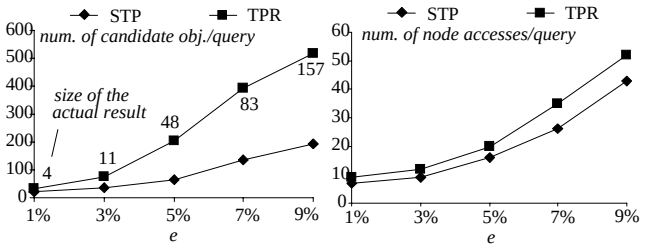
Figure 6.9: Server I/O cost vs. D (e=2.5%, *qtlen*=10)

The next experiment compares STP- and TPR-trees (we use the TPR* implementation). Figure 6.10a (6.10b) fixes e to 2.5%, and illustrates the refinement (I/O) cost, as a function of *qtlen*. In both cases, the STP-tree outperforms the TPR-tree significantly. In particular, the number of candidate objects in STP is about 15% larger than the actual query size (also included in the diagrams). On the other hand, the number of candidate objects retrieved by TPR is 3-4 times larger. The difference in the performance of the two structures increases with *qtlen*. Similar observations hold for Figure 6.11, which measures performance with respect to the range radius e, fixing *qtlen* to its median value 10.



(a) Refinement cost (b) Server I/O cost

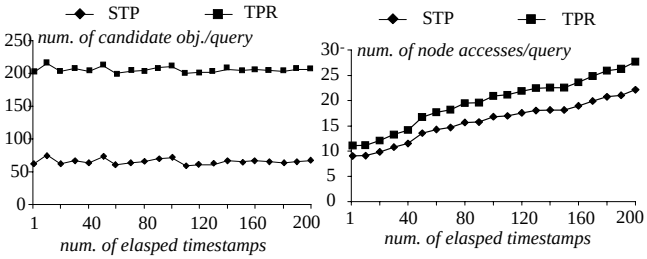
Figure 6.10: Query costs vs. *qtlen* (e=2.5%)



(a) Refinement cost (b) Server I/O cost

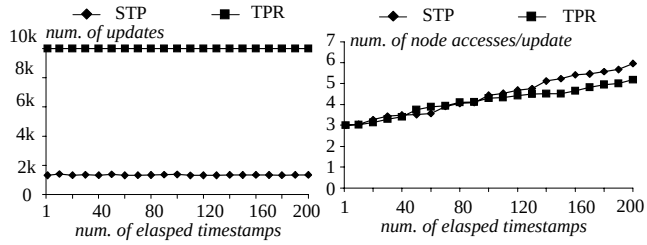
Figure 6.11: Query costs vs. *e* (*qtlen*=10)

Figures 6.12a and 6.12b exhibit the average refinement and server costs at individual timestamps, for queries with parameters *e*=2.5% and *qtlen*=2.5%. Although the number of candidate objects remains roughly the same with time, the I/O overhead gradually increases due to the structural deterioration of the both trees. This is consistent with the results of the previous work [SJLL00, TPS03]. Nevertheless, the STP-tree still outperforms the TPR-tree in all cases. The last experiment concerns the update cost. Specifically, Figure 6.13a plots the number of updates per timestamp, issued by objects whose horizon bound has been exceeded. As discussed in the context of Figure 6.8a, in case of *D*=1 (i.e., the TPR-tree) each object must issue an update at every timestamp. Since the STP-tree uses a higher (*D* =3) degree, the number of updates is significantly smaller. Figure 6.13b illustrates the average cost of each update (including insertions and deletions) in the corresponding structure. Similar to Figure 6.12b, the increase in the update cost is caused by the structural degradation.



(a) Refinement cost vs. time (b) Server I/O cost vs. time

Figure 6.12: Query cost vs. time (*e*=2.5%, *qtlen*=10)



(a) Number of updates vs. time (b) Server I/O cost

Figure 6.13: Update cost vs. time

7. CONCLUSION

Previous techniques for prediction in spatio-temporal databases usually assume linear movements, which seriously hinders their applicability in practice. Our paper overcomes the shortcomings of linear prediction with a novel architecture that supports arbitrary motion patterns, not necessarily known in advance. Further, we propose the concept of recursive motion functions and prove, both theoretically and empirically, that it can accurately express a large number of movements. Finally, we develop a spatio-temporal access method that generalizes the current state-of-the-art indexes to polynomial of higher (than 1) degrees. We believe that this work lays down a solid foundation for future research on the management and indexing of moving objects. For example, existing results on predictive selectivity estimation [CC02, TSP03] are applicable only to the filter step of our architecture, while the selectivity estimation of actual object trajectories remains an open problem.

ACKNOWLEDGEMENTS

This work was supported by grant HKUST 6180/03E from Hong Kong RGC.

REFERENCES

[AAE00] Agarwal, P., Arge, L., Erickson, J. Indexing Moving Points. *PODS*, 2000.

[AA03] Aggarwal, C., Agrawal, D. On Nearest Neighbor Indexing of Nonlinear Trajectories. *PODS*, 2003

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[CC02] Choi, Y., Chung, C. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. *SIGMOD*, 2002.

[HKT03] Hadjieleftheriou, M., Kollios, G., Tsotras, V. Performance Evaluation of Spatio-temporal Selectivity Estimation Techniques, *SSDBM*, 2003.

[HKTG02] Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D. Efficient Indexing of Spatiotemporal Objects, *EDBT*, 2002.

[ISS03] Iwerks, G., Samet, H., Smith, K. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. *VLDB*, 2003.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. *PODS*, 1999.

[PFTV02] Press, W., Flannery, B., Teukolsky, S., Vetterling, W. Numerical Recipes in C++ (second edition). *Cambridge University Press*, ISBN 0-521-75034-2, 2002.

[PSTW93] Pagel, B., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance in Spatial Data Structures. *PODS*, 1993.

[SJ02] Saltenis, S., Jensen, C. Indexing of Moving Objects for Location-Based Services. *ICDE*, 2002.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. *SIGMOD*, 2000.

[Tiger] http://www.census.gov/geo/www/tiger/

[TP02] Tao, Y., Papadias, D. Time-Parameterized Queries in Spatio-Temporal Databases. *SIGMOD*, 2002.

[TPS03] Tao, Y., Papadias, D., Sun, J. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *VLDB*, 2003.

[TSP03] Tao, Y., Sun, J., Papadias, D. Selectivity Estimation for Predictive Spatio-Temporal Queries. *ICDE*, 2003.

[TUW98] Tayeb, J., Ulusoy, O., Wolfson, O. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3): 185-200, 1998.