Kalman-Tree: an Index Structure on Spatio-Temporal Data

Ning Hu
Computer Science Department
Carnegie Mellon University
ninghu@cs.cmu.edu

Minglong Shao
Computer Science Department
Carnegie Mellon University
shaoml@cs.cmu.edu

April 25, 2003

Abstract

In the research area of spatio-temporal databases, we need to answer the queries like "select the objects which will be in a specific area at a future time" quickly and accurately. Solutions to this question include two aspects: a mathematic model to predict future location with adequate accuracy and an index structure that can retrieve the qualified objects quickly. Currently, the proposed future location estimation in this field is limited to some simple linear functions of velocity. It might not be able to get ideal prediction results as the prediction algorithms are too simple.

This paper proposes a technique targeting the future location estimation problem in spatio-temporal databases. It uses the useful estimator *Kalman filter* to predict the continuous trajectory of the moving objects in multi-dimensional space. Based on the algorithm, we propose a modified R-tree indexing structure (KR-tree) that is well suitable and efficient for the Kalman equations.

1 Introduction

In the applications of wireless communication and mobile computing, tracking the changing position of objects is becoming more and more necessary. This poses new challenges on spatio query processing. Traditional spatio algorithms/structures/database systems, which mainly focus on storing and retrieving locations of static objects, don't suffice in the new mobile environment. New spatio-temporal solutions are called for in order to answer the location queries with temporal parameters.

The work presented in this paper targets at providing a fast and accurate method to answer the spatio-temporal queries like "select the mobile users who will be in a specific area in the next five minutes". This involves two aspects:

- an algorithm that can predict future locations of moving objects based on the past information;
- a time-parameterized index structure that can retrieve spatio locations effectively.

We argue that the desirable solution should meet the following requirements:

- first, the prediction algorithm to be as simple as possible in the sense that the training and predicting process is fast and the index structure integrated with it is not too complex;
- secondly, the parameters of the prediction algorithm should be calculated incrementally so that when the predicted positions are too far from the actual positions, the algorithm can adapt itself in the changing environment with little efforts;
- last but not least, it should be accurate as much as possible in order to give right answers.

In this paper, we adopted a simplified Kalman function to predict future object positions. Intuitively, Kalman equations can describe the trajectory of a moving object more accurately than the existing models that use velocity. Based on this prediction algorithm, we proposed Kalman-R-tree (KR-tree for short), an R-tree like index structure, which can locate the qualified objects quickly.

This project is inspired by the work of TPR-tree [11] (time-parameterized R-tree). Therefore, KR-tree belongs to the same classification of indexing [11]. Similar to the TPR-tree, KR-tree has the following characteristics: data is indexed its native, d-dimensional space. We will parameterize the index structure using n previous positions so it can predict the future positions; KR-tree index structure partitions the data, using an R-tree like partition strategy (minimum bounding rectangle); KR-tree does not employ replications; KR-tree doesn't require periodic rebuilding. But the performance might deteriorate as time progresses. The bounding rectangles in the KR-tree are functions of time. Due to its R-tree origin, KR-tree is capable of indexing points in one-, two-, and three- dimensional space, which suffice the most cases in real applications.

After the brief introduction of KR-tree, the rest of the paper is organized as the following: section 2 presents a survey of moving objects indexing, Kalman filter and the on-line update methods for its parameters. Section 3 describes the proposed method in details, including the architecture of the system, the customization of Kalman filters and design details, the structure of KR-tree and its related algorithm. Experiment setup and results will be discussed in section 4. We conclude our work and propose possible future work in Section 5.

2 Survey

2.1 Indexing Moving Objects

Index on moving objects is proposed to support database applications involving continuous movement. It usually targets at answering queries like "Given a rectangle R and real value t_q, report all K points of S that lie inside R at time t_q" [1]. It can be classified into two different types: one is indexing current

and anticipated future positions of moving object; the other is indexing histories of the positions of moving objects [11]. We will focus on the former indexing problem in this project.

- Index on historic data

  In many applications, such as traffic supervision or mobile communication systems, spatio-temporal queries asking for historic data mainly focus on summarized data (or aggregate information). Aggregate R-tree [8] improves the original R-tree towards aggregate processing by storing, in each intermediate entry, summarized data about objects residing in the subtree. In the situation where some MBRs are covered by query rectangle, such aggregate information stored in the node can avoid accessing nodes in the subtree since it already has sufficient information.

- Index on current and future positions

  TPR-tree (time-parameterized R-tree) [11] is a balanced, multi-way tree with the structure of an R-tree that can answer prediction queries on moving objects. Entries of TPR-Tree in leaf nodes are pairs of the position of a moving point which is represented by a reference position and a corresponding velocity – (x, v). Reference position is the location at time t_{ref} which usually takes the index creating time t_l. A internal node of TPR-Tree is represented with (1) minimum bounding rectangle (MBR) that bounds its child nodes at reference time t_l, and (2) a velocity vector. As in traditional R-trees, the MBR tightly encloses all entries in the node at time t_l. The velocity vector of the (intermediate) MBR is determined as follows: (1) the velocity of the upper (right) edge is the maximum of all velocities on this dimension in the sub-tree; (2) the velocity of the lower (left) edge is the minimum of all velocities on this dimension. This ensures that the MBR always encloses the underlying objects at any time no earlier than t_l, but it is not necessarily tight. This kind of MBR is termed "conservative bounding rectangles".

  When answering query about future location, TPR-tree employs a simple linear function $x(t) = x(t_{ref}) + v(t - t_{ref})$ to predict object position at future time t. The MBRs at the future query time will also be computed based on the current extents and velocity vectors. So the queries at future time can be processed in exactly the same way as in the tradition R-Tree, except that the extents of the MBRs are computed dynamically and then compared with the query window. TPR-Tree is a successful index method in answering queries on moving objects. But due to its simple prediction method, it is unavoidably inaccuracy on prediction. Queries far in the future may be of little value. So it has parameters of querying window (W) which specifies how far queries can "look" into the future, index usage time (U) which is the time interval during which an index will be used for querying, and time horizon (H) which is the sum of index usage time (U) and the querying window (W).

  [5] employs the dual data transformation where a line $x = x(t_{ref}) + v(t - t_{ref})$ is transformed to

the point $(x(t_{ref}), v)$. Thus it can use regular spatial indexes to access spatio-temporal data. This paper focused studying the one and two dimensional case and gave a dynamic, external memory algorithm with guaranteed worst case performance and linear space. It also gave other practical approximation algorithms with different restrictions. More details can be found in the paper.

The common characteristic of the two index methods is that they both adopt simple linear function to predict future positions.

2.2 Trajectory Prediction

Trajectory prediction is the process of predicting the future positions of a moving object based on the historic data. So we can answer the queries like "reporting all the cars that will in the region R in the next 5 minutes". It is usually implemented by using a pre-defined moving function. Linear function is the easy-to-thought method which predicts position at time t using $x(t) = x(t_{now}) + v(t - t_{now})$.

Though the linear function is simple and requires fewer storage requirements, the too simple model will adversely affect the accuracy of prediction which may further impair the performance of the index. For example, the index needs to be update more frequently and the query window might be restricted to a small range.

Kalman filtering is a widely used method in the field of signal processing. It is more powerful in extracting trajectory features than the simple velocity function. We can expect that trajectory prediction with Kalman Filter will be more accurate.

2.3 Kalman Filter

The *Kalman filter* is an iterative model designed to calculate forecasts and forecast variances for time series models. It is deemed as the best possible (optimal) estimator for a large class of problems and a very effective and useful estimator for an even larger class [15]. Since the time of its introduction, the Kalman filter has been the subject of extensive research and application, particularly in several areas like autonomous or assisted navigation, computer graphics and computer vision. This is likely due in large part to advances in digital computing that made the use of the filter practical, but also to the relative simplicity and robust nature of the filter itself. So it is not unexpected to observe that the Kalman filter has recently popping up in a even wider variety of computer applications.

The *Kalman filter* targets on modeling a Linear Dynamical System (LDS), which is a partially observed stochastic process with linear dynamics and linear observations, both subject to Gaussian noise. It can be defined as follows, where x_t is the hidden state at time t, and z_t is the observation.

$$x_k = Ax_{k-1} + w_k, \qquad \text{State Transition} \tag{1}$$

$$z_k = Hx_k + v_k, \qquad \text{Observation} \tag{2}$$

A is the transition matrix that relates the state at the previous time step $k-1$ to the state at the current step k, and H is the coefficient matrix that relates the state to the measurement z_k. The process and measurement noise w_k and v_k are assumed to be independent (of each other), white, and with normal probability distributions, i.e. distributed according to $w \sim N(0, Q)$ and $v \sim N(0, R)$.

The *Kalman filter* trys to perform filtering on this model, i.e., computing possibility $P(x_t|y_1, ..., y_t)$.

From the statistical perspective, the *Kalman filter* model can be infered from Hidden Markov Model (HMM). It is reasonable because Linear Dynamic Systems (LDSs) and Hidden Markov Models are based on the same assumption: a hidden state variable, of which we can make noisy measurements, evolves with Markovian dynamics. And both have the same independent diagram and consequently the learning and inference algorithms for both have the same structure. Thus the forward-backward equations for the HMM, specialized to linear-Gaussian assumptions, would lead directly to *Kalman filtering* [6].

It can be also explained in the sense of signal processing. The *Kalman filter* is essentially a set of mathematical equations that implement a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated *error* covariance – when some presumed conditions are met. The way the Kalman filter works is to estimate a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups: *time update ("predictor")* equations and *measurement update ("corrector")* equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a *priori* estimates $\hat{x}_k^- \in \Re^n$ for the next time step. The measurement update equations are responsible for the feedback – i.e. for incorporating a new measurement into the a *priori* estimate $\hat{x}_k^-$ to obtain an improved a *posteriori* estimate $\hat{x}_k \in \Re^n$.

The final estimation algorithm resembles that of a *predictor-corrector* algorithm for solving numerical problems as shown in Figure 1.

Besides the variables described previously, other variables in Figure 1 are the *a priori* estimate error covariance P_k^-, the *a posteriori* estimate error covariance P_k, and the Kalman *gain* $K_k \in \Re^{n \times m}$ producing a state estimate that minimizes the mean squared error of the reconstruction.

However, if the process to be estimated and (or) the measurement relationship to the process is non-linear, the *Kalman filter* is no longer applicable here due to the condition restrictions. To tackle such restrictions, various versions of Kalman filter have been proposed. Some of the representatives filters are: the *Extended Kalman filter (EKF)* [15] uses the gradient linearization Jacobian to linearizes the non-linear process difference and measurement relationships; the *Unscented Kalman filter (UKF)* [10] is also a non-linear model but it replaces the Jacobians used in the EKF with unscented non-linear transformations, reducing linearization errors in all cases.

The *Kalman filter* would not work in the system with non-Gaussian noise either. The represented method that can model the system with unknown distribution noise is the *Particle filter*, which is a

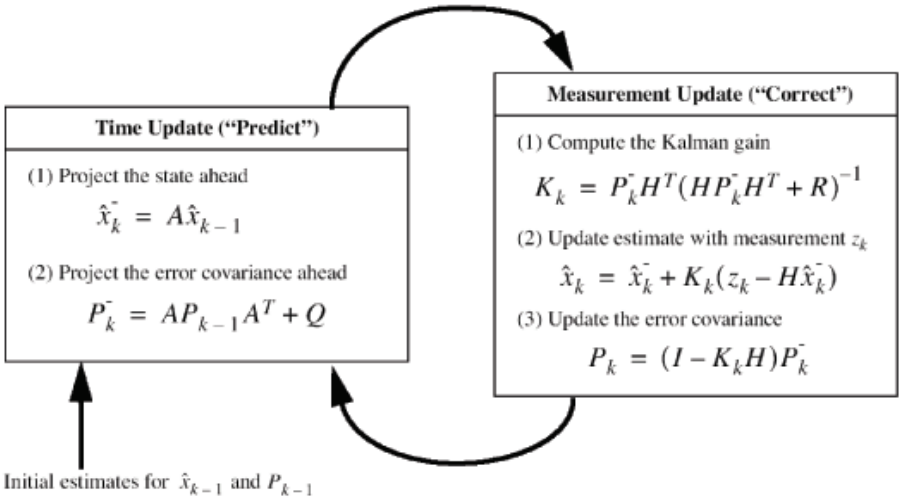

Figure 1: A complete picture of the operation of the Kalman filter

popular sequential Monte Carlo technique. And it has the reputation of best performance over all similar filters.

The *Kalman filter* is essentially a one-step predictor. But it can also be used for multi-step prediction. Consider a sequence of predictions into the future which are made at step k, and let us juxtapose with these predictions the expressions for the corresponding values of the true state variables. Then, on the assumption that $A_k = A$ is a constant matrix, we have

$$\begin{array}{rclcrcl} \hat{x}_{k+1}^- & = & A\hat{x}_k & & x_{k+1} & = & Ax_k + w_{k+1} \\ \hat{x}_{k+2}^- & = & A^2\hat{x}_k & & x_{k+2} & = & A^2x_k + Aw_{k+1} + w_{k+2} \\ \hat{x}_{k+3}^- & = & A^3\hat{x}_k & & x_{k+3} & = & A^3x_k + A^2w_{k+1} + Aw_{k+2} + w_{k+3} \\ & \vdots & & & & & \\ \hat{x}_{k+t}^- & = & A^t\hat{x}_k & & x_{k+t} & = & A^tx_k + \sum_{j=0}^{t-1} A^j w_{k+t-j} \end{array} \tag{3}$$

It follows that the error in predicting t periods into the future is given by

$$\hat{x}_{k+t}^- - x_{k+t} = A^n(\hat{x}_k - x_k) - \sum_{j=0}^{t-1} A^j w_{k+n-j} \tag{4}$$

As illustrated above, the *Kalman filter* is ideal for trajectory prediction. It is way more advanced than the current published trajectory prediction algorithms used in spatio-temporal databases [11] and will definitely produce much more precise predictions. Even for the computational cost, though of course it is more expensive than the simple linear function of time, in practical it will not greatly reduce system performance.

2.4 On-line Update of Kalman Filter Parameters

The *transition matrix* A, H, *process noise covariance* Q and *measurement noise covariance* R are important parameters of the *Kalman filter*. The filter performance is largely determined by how well the parameters are tuned according to the system. The closer they are to the actual characteristics of the current system, the better filtering result it will get.

There are some techniques for off-line parameters training. A popular one is em [4] [12]. It finds maximum likelihood estimates of the parameters from sequence of training data. But EM is quite computationally expensive. That is why it'd best to be used off-line.

However in real applications, it is pretty common that the dynamics, process and measurement noise of the system is unknown before the filtering process, or they will change from time to time. In this case, on-line update of the parameters during the filtering process is necessary.

Several methods are proposed for on-line parameters update. A representative branch is called *Kalman adaptive filter* (AKF) [3]. A considerable amount of research has been carried out in this area, but in practice it is often necessary to redesign the adaptive Kalman scheme according to the particular characteristics of the problem at hand. For example, it is hard to directly use Kalman adaptive filter for multi-step prediction, as the state transition matrix A is replaced by the transition matrix of the state estimation error in AKF. But A is particulaly important to multi-step prediction, see Equation 4.

[7] proposed another kind of *Adaptive Kalman filter* and gave out a method to update A on-line. However in their paper, A is estimated with only two last state vectors. This can result in un-stable A during the process, while from database point of view, A needs to converge and become stable as fast as possible to avoid frequently indexing update.

Eventually we found Recursive Least-Squares Estimation (RLS), which is the recursive version of Least Squares Estimation (LSE). LSE is designed for solving the classical linear regression model $y = Ax$ for A by minimizing the squares error. The formal definition is: given N samples, $(x_1[i], ..., x_v[i], y[i]), i = 1, ...N$, find the $a_1, ..., a_v$ that minimize

$$\min_{a_1,\cdots,a_v} \sum_{i=1}^{N} (y[i] - a_1 x_1[i] - \cdots a_v x_v[i])^2 \tag{5}$$

Using matrix notation, the solution is:

$$a = (X^T \times X)^{-1} \times (X^T \times y) \tag{6}$$

The matrix X is the $N \times v$ matrix with the N samples of the v dimensions. Let $D = X^T \times X$, so the key step is to get D^{-1}. Inversion of a $v \times v$ matrix takes $O(v^3)$ time. This is acceptable only when the number of data samples is fixed and small. However, it is not suitable for applications where there are large number of data sampels and new samples are added dynamically, because the new matrix D and its inverse should be computed whenever a new set of samples arrive.

Recursive Least-Squares Estimation (RLS) provides an alternative to compute D^{-1} incrementally. Specifically, it computes D_n^{-1} from D_{n-1}^{-1}, where D_n denotes the D matrix at time n. This is made possible by the *matrix inversion lemma* [9]. And its computation cost is reduces to $O(v^2)$. Its applications prove that RLS is robust as well as efficient.

The RLS formulas for the solutions is: defining the so called *RLS gain matrix* $G_n = D_n^{-1}$. $G_n(n = 1, \cdots)$ can be computed recursively as follows,

$$G_n = \frac{1}{\lambda} G_{n-1} - \frac{1}{\lambda}(\lambda + x[n] \times G_{n-1} \times x[n]^T)^{-1} \times (G_{n-1} \times x[n]^T) \times (x[n] \times G_{n-1}), n > 1. \quad (7)$$

λ is called *forgetting factor* ranging from 0 to 1 ($\lambda = 1$ means "not forgetting"). It represents the diminishing rate of the effect of each sample as time elapses. In practise, λ is set to be 0.99. $x[n]$ is the x value at time n. Thus the coefficient vector a_n after the n-th sample has arrived, can also be updated incrementally,

$$a_n = a_{n-1} - G_n \times x[n]^T \times (x[n] \times a_{n-1} - y[n]) \quad (8)$$

The initial $G_0 = \delta^{-1} \times I$, and $a_0 = \mathbf{0}$, where δ is a small positive number (e.g., 0.004), I is the identity matrix, and $\mathbf{0}$ is a column vector of v zeros.

From the description above, we can see that RLS can be used for recursively updating the transition matrix A, sicne the linear algebraic equation to be solved in RLS has similar form of the state trantision equation Eq.1. Actually RLS is also used in filtering and can be deemed as a special Kalman filter with no process or measurement noise. A nice description of RLS can be found at [2], algorithm details at [9].

3 Proposed Method

Since Kalman filtering is a more accurate model to predict future positions, we propose to replace the linear prediction module in the existing index algorithm with Kalman filter. The project can be divided into following parts:

3.1 Customization of the Kalman Filter

Designing the state representation is a crucial part. It should be able to properly represent the true hidden state of the system. Otherwise the performance will be greatly hurted. For modeling roughly smooth trajectories, some common forms of the state design are $x_k = [p, v]_k^T$ or $x_k = [p, v, a]_k^T$ [14] with the obervation state as $z_k = p_k$ [13], where p, v, and a represent the position, velocity and acceleration at step k respectively. However, in order to simplify structure and reduce computational cost, we design the state in the following form:

$$x_k = [p_k, p_{k-1}, p_{k-2}, ...p_{k-n}]_k^T \quad (9)$$

It means the last n positions will be included in x_k. Thus the observation state of the filter in our system is

$$z_k = x_k, \tag{10}$$

which means, H is an identity matrix. Further more, the state transition matrix A will look like

$$A = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-v-1} & a_{1,n-v} & \cdots & a_{1,n\times v} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-v-1} & a_{2,n-v} & \cdots & a_{2,n\times v} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{v,1} & a_{v,2} & \cdots & a_{v,n-v-1} & a_{v,n-v} & \cdots & a_{v,n\times v} \\ 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \end{vmatrix}. \tag{11}$$

where v is the dimension of the position. So only v rows of A needs update, other parts are constant, i.e. when RLS is applied to estimate A, it just computes the first v rows of A respectively.

Each trajectory object has its own current state x_k, observation state z_k, state transition A, and RLS gain G. Currently we simplify the system assumption that the process and measurement error are 0. That is to say, the current system doesn't model noise Q and R. But it could be implemented in the future extention.

Moreover, as the process noise w is centered, the latter complex term in the multi-step prediction quation Eq.4 can be ignored to some extent. H is an identity matrix, the term related to measurement noise v can be ignored too. Thus we only need state x and transition A for prediction. This proves that our simplied assumption is reasonable.

3.2 Overview of the System

In this section, we will describe the typical scenario of using KR-tree. The whole system consists of two parts: moving objects (clients) and a central KR-tree index (server). We assume that each object has a small buffer (location buffer) that can store the most recently locations and it also has a limited computing ability. The objects will measure (sense) their locations at a fixed or changing time interval and update the location buffers. The objects are responsible to decide whether the prediction is accurate enough. If the parameters of the prediction function are out of date that can't describe the trajectory with required accuracy, the object will update the parameters through an incremental algorithm (please refer to the section of RLS algorithm). These new parameters will be sent to the server part and used

to update the corresponding entries in the KR-tree. A graphic description of the system is shown in Figurei 2.

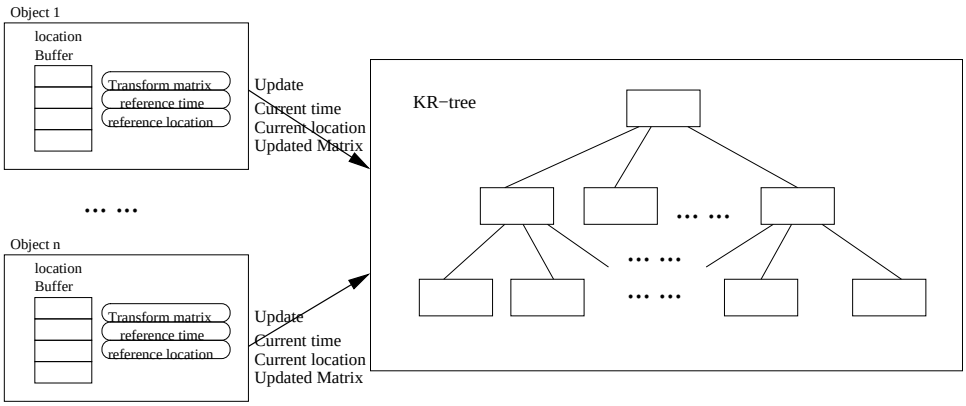

Figure 2: Overview of the System

3.3 Integration of Kalman Filter and R-tree

Different from the indexes on static spatio data, where coordinates of locations provide enough information to retrieve object through index, indexes on spatio-temporal data should be a function of time and locations so that it is possible to answer the queries asking future positions. For each entry in the leaf node in the KR-tree, which corresponds to a moving object, it is natural to store the transform matrix, reference time, and locations at that time. New locations at future time can be easily calculated through Kalman functions. For internal nodes within KR-tree, it is not straightforward on what kind of information should be stored and how to apply Kalman functions to prediction locations. The challenges here lie in the following aspects: first, we want the internal nodes to use similar prediction functions as the leaf entries. But it is not clear on how to define the motion of an internal node and what the transform matrix will look like; secondly, according to the property of R-tree, rectangles of parent nodes should fully contain their child nodes. In the context of KR-tree, it means that this property should always hold at any time in the future; thirdly, we hope that the bound rectangles should be as tight as possible so that the search performance of KR-tree won't be too bad.

In this project, we proposed a method that uses two bound transform matrixes ($Amin$ and $Amax$) and two bound locations ($Xmin$ and $Xmax$) to "describe" the motion of the internal nodes.

3.3.1 Structure of the KR-tree

- Entry in the leaf node

The structure of entries in the leaf nodes includes three parts:

- a transform matrix as defined in Kalman equations, denoted as A;
- a reference time, denote as t_{ref};
- locations information at t_{ref}, denoted as X_{ref}.

Note that X_{ref} might contain location information at t_{ref-1}, t_{ref-2} or more. The choice of how many previous locations should be used depends on the desired update frequency and accuracy, and the technical limitations. For example, in the two-dimensional space, if we use two locations to predict, the entry in the leaf node will contain the following data:

- A: a $2 * 4$ matrix,
- X: a $1 * 4$ vector,
- t timestamp

According to the equation:

$$\begin{bmatrix} x_t \\ y_t \\ x_{t-1} \\ y_{t-1} \end{bmatrix} = A * \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ x_{t-2} \\ y_{t-2} \end{bmatrix}$$

The last two rows of A are [1 0 0 0] and [0 1 0 0]. Therefore, we compress A into a $2 * 4$ matrix. This saves space, thus increase the fan out of the KR-tree. It is especially useful when we have to use more steps to predict locations in high-dimensional space.

- Internal node

  The structure of internal nodes includes five parts:

  - two bound matrixes, denoted as $Amin$ and $Amax$;
  - two bound locations, denoted as $Xmin$ and $Xmax$;
  - a reference time t_{ref}.

  $Amin$ is a "minimum matrix" of its child nodes. It is defined as follows:

$$(a_{min})_{ij} = \min_n (a_n)_{ij}, n = 1..K$$

K is the number of children of the node $(a_{min})_{ij} = Amin$. $Amax$ is a "maximum matrix" of the child nodes. It is defined as:

$$(a_{max})_{ij} = \max_n (a_n)_{ij} \tag{12}$$

$Xmin$ is a "minimum location" of the child nodes. It is defined as:

$$(x_{min})_i = \min_n (x_n)_i \tag{13}$$

$Xmax$ is a "maximum location" of the child nodes. It is defined as:

$$(x_{max})_i = \max_n (x_n)_i \tag{14}$$

t_{ref} is deinfed as

$$\max_n (t_{ref})$$

We assume that the coordinates of any moving objects are always positive. This is a reasonable assumption since we can define the origin as we want.

According to the definitions above, it is easy to see that $Xmin$ and $Xmax$ will give us the bound rectangle at the reference time. For the bound rectangle at a future time t_q, X_{t_q} is calculated by the equation:

$$X_{new} = A^n * X_{old} (n = t_q - t_{ref})$$

We need to define a way to calculate $Amin^n$ and $Amax^n$ so that $Xmin_{new}$ and $Xmax_{new}$ will give use the bound rectangle at time t_q that still cover all the child rectangles.

we define $A * B = C$ as

$$(c_{min})_{ij} = \sum_{k=1}^{n} \min(a_{ik} * b_{kj}) \tag{15}$$

$$(c_{max})_{ij} = \sum_{k=1}^{n} \max(a_{ik} * b_{kj})$$

In this way, we will first calculate $A, A^2, A^4, ..., A^{log_2 n}$, then use these matrixes to finally obtain A^n. This method can guarantee that the $Amin^n$ and $Amax^n$ are bound matrixes at time t_q. So $Xmin_{new} = Amin^n * Xmin_{old}$ and $Xmax_{new} = Amin^n * Xmax_{old}$ are bound locations at time t_q. Therefore, bound rectangle at t_q still cover its kids. This is a conservative method, which may result in a relatively loose bound rectangle.

One potential problem in the calculation of $Amin^n$ and $Amax^n$ is that for entries that greater than 1, they will increase very quickly which may end up with a very loose rectangle. This potential drawback will limit the steps we can look into the future. In our experiments, we found that in most cases, the entries in the transform matrix are less than 1 which doesn't have this problem. So combining with the predication accuracy provided by Kalman function, we can look further in the future than TPR-tree.

3.3.2 Search, Insertion, Deletion, and Update

- Search

  KR-tree accepts queries like "report the objects which will be at some specific area in a future time t_q". When searching the KR-tree, rectangles of each node will be calculated using the above method. The other process is the same as the traditional R-tree.

- Insertion

  Insertion is a little bit tricky, because we have to choose which subtree to follow and how to deal with overflow. In out current implementation, we adopt the quadratic algorithm which chooses the subtree whose cover rectangle needs the least enlargement. As for the split algorithm, we use the split strategy that optimizes area. For moving object, the currently close objects may be far away in the near future, so we look n steps forward to calculate the bound rectangles. n is a tunable parameter. We use two as a default value in our experiment. If new inserted node has a more recent timestamp, the root reference time will be updated according to the new timestamp. The $Amin$ and $Amax$, $Xmin$ and $Xmax$ will be adjusted accordingly.

- Deletion and Update

  The operation of deletion is essentially the same as the traditional R-tree except the fact that we calculate cover rectangle of a node dynamically. We implement update as a combination of deletion and insertion. We argue that deletion followed by an insertion is actually better than a "real" update because first the cost of deletion and insertion is similar to the cost of real update; secondly, the combination of deletion and insertion may put the node in a more optimal place that results in a smaller bound rectangle.

4 Experiment

We implemented KR-tree based on the DR-tree code. The clients (moving objects) are organized using a B+tree structure. We did the experiments on a Pentium III machine running Linux. Although the experiments mainly focus on two dimensional space, KR-tree has no difficult dealing with data at a higher dimension because of its R-tree origin.

KR-tree has to store more information than TPR-tree. In the two-dimensional space, if we use two steps in prediction, a 1k page can contain only 9 entries. This is a big problem of the current KR-tree.

In the following experiments, we will use the same configurations, shown in Table 1 .

Our experiments are limited by the lack of test data. Our synthesized data has limitations of representativeness and amount. So we didn't evaluate the KR-tree in terms of time and I/O performance

Dim of the space	2
Number of previous locations used in prediction	2
Steps used to calculate rectangle during insertion	2
Trajectories used	synthesized data including circle, line, parabola, etc.

Table 1: configurations in the experiments

which require a statistical approach. The results showed in this section mainly focus on comparing the prediction accuracy. We also show a graph of KR-tree.

4.1 Prediction Accuracy Comparison

Our K-Rtree utilizes the state transition matrix A for prediction, which is recursively computed by RLS, while the comparable indexing structure TPR tree utilizes velocity. We are interested in comparing the prediction accuracy of these two kinds of R-tree structure to show the advantage of our K-Rtree over other comparable indexing structures.

As we are using RLS to update A recursively and estimated process or measurement noise are 0, we call ours *RLS* algorithm; We don't have the TPR-tree system at hand to do the real comparison, thus we simulates its prediction principle accordingly, namely *Velocity* algorithm.

We generate several representative trajectories on 2-dimensional plane and feed them into these two different algorithms. At each time step, the algorithms will predict the position at future time t. The *RLS* algorithm uses the last n samples to be the state x. Error is measured in the sense of distance between the predicted position and the actual position.

Figure 3 shows a circling trajectory and Figure 4 shows the prediction error on that circling trajectory for at future time $t = 10$ from *Velocity* and *RLS*. *RLS* uses $n = 2$ last samples.

As we can see in Figure 4, the prediction error from *Velocity* is constantly high, while that from *RLS* is clearly converging to minimum. Figure 5 shows the prediction error on the same circling trajectory and same t, except that $n = 3$ here. Comparing it with Figure 4, we can see the prediction error from *RLS* with $n = 3$ reduces even faster. It is reasonable, as more information contained in the state x will certainly help model the system better and converge A faster.

Figure 6 shows a straight line trajectory with some noise. Figure 7 shows the prediction error on such trajectory with $n = 2, t = 10$. Here we can see for those trajectories with noise, which is a common case in real world, *Velocity* certainly performs badly, while *RLS* can still successfully handles it. Note that process and measurement noise Q and R haven't been embedded into our current K-Rtree structure yet.

We also tried the parabola trajectories, with or without noise, and other common trajectories such as sinusoid. In all those experiments, *RLS* always outperforms *Velocity*.

From the experiments above, we can clearly see that using merely velocity to predict future position is good only when it has constant velocity. It would not work if there is any derivative of the velociy, no matter whether it is noise or arbitrary accleration, while *RLS* can model those pretty well. One may say what if accleration is taken into account in TPR-tree, but K-Rtree should still perform better, as it has better flexibility and scalability.

As future prediction time t increase, the error from *Velocity* and *RLS* both increase too. However, It is clearly that the error increasing rate of *Velocity* is faster than that of *RLS*.

The experiments also help make up some decision on our system design. For example, it shows that there should be an offset time t_{init} for building the KR-tree, as A needs some time to adjust to the system and converge to a stable state. Normally setting $t_{init} >= 50$ is ok. Also since updating the indexing structure is somewhat expensive, we also design the system that on the client object side, the A matrix needs not to be updated at every step unless the one-step prediction error is greater than some threshold. Once it is updated, the client side will send a request to the indexing server side to delete and re-insert the trajectory object. The experiments also show that even for $n = 2$, only including last two samples in the state, is surprisingly good enough for most trajectories we've tested. Of course greater n will result in better result, but it will also cost extra computing power and storage.

4.2 MBR of KR-tree

The following KR-tree uses two-step forward to calculate rectangle during the insertion, bound rectangle is looser than common Rtree.

5 Conclusions and Future Work

KR-tree is a new structure toward providing a fast and accurate method to index spatio-temporal data. By integrating the Kalman prediction function into existing R-tree, it can answer the queries like "find the objects which will be in some area at a future time" effectively. KR-tree has better performance in the sense of prediction accuracy when compared with TPR-tree. What's more, the parameters of Kalman function can be updated incrementally so that the functions are adaptive to the changes in the trajectories.

We currently implemented a simplified K-Rtree, though its assumption is reasonble for prediction purpose and gives out satisfying result, further extention to fully utilize the power of Kalman filter would be exciting. On indexing side, more interesting work can be done in designing tighter rectangles, compressing information at each node to increase the fan out of KR-tree, designing a better split strategy so that the overall bound rectangles will overlap less and have smaller area.

6 Acknowledgement

We are grateful to Dr. Tony Tao for his invaluable guidance and thought invoking discussions. We thank professor Christos Faloutsos for his advice on incremental update of transforming matrix. We would like to thank Edoardo Airoldi for his help in understanding Kalman filter, and Simonas Saltenis for his previous work on TPR-tree.

References

[1] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Symposium on Principles of Database Systems*, pages 175–186, 2000.

[2] Theodore Johnson H. V. Jagadish Christos Faloutsos Alexandros Biliris Byoung-Kee Yi, N. D. Sidiropoulos. Online data mining for co-evolving time sequences. May 1999.

[3] A. Gelb. Editor. Applied optimal estimation. 1974.

[4] Z. Ghahramani and G. E. Hinton. Parameter estimation for linear dynamical systems. Technical Report CRG-TR-96-2, Department of Computer Science, University of Toronto, 1996.

[5] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 261–272. ACM Press, 1999.

[6] Tom Minka. From hidden markov models to linear dynamical systems. Technical report, 1998.

[7] JP Urban P. Wira. A new adaptive kalman filter applied to visual servoing tasks.

[8] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. Efficient OLAP operations in spatial data warehouses. *Lecture Notes in Computer Science*, 2121:443–??, 2001.

[9] D.S.G. Pollock. Time series analysis. May 1999.

[10] J. K. Uhlmann S. J. Julier and H. F. Durrant-Whyte. A new approach for filtering nonlinear systems. *Proceedings of the American Control Conference*, pages 1628–1632, June 1995.

[11] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.

[12] J. R. Rohlicek V. Digalakis and M. Ostendorf. Ml estimation of a stochastic linear system with the em algorithm and its application to speech recognition. pages 431–442, oct 1993.

[13] Y. Gao W. Wu, M. J. Black and E. Bienenstock etc. Inferring hand motion from multi-cell recordings in motor cortex using a kalman filter. In *SAB02-Workshop on Motor Control in Humans and Robots Edinburgh Scotland (UK)*, pages 66–73, aug 2002.

[14] Y. Gao W. Wu, M. J. Black and E. Bienenstock etc. Neural decoding of cursor motion using a kalman filter. In *Neural Information Processing Systems*, dec 2002.

[15] Greg Welch and Gary Bishop. An introduction to the kalman filter. *ACM SIGGRAPH 2001 Course Notes*, 2001.

Contents

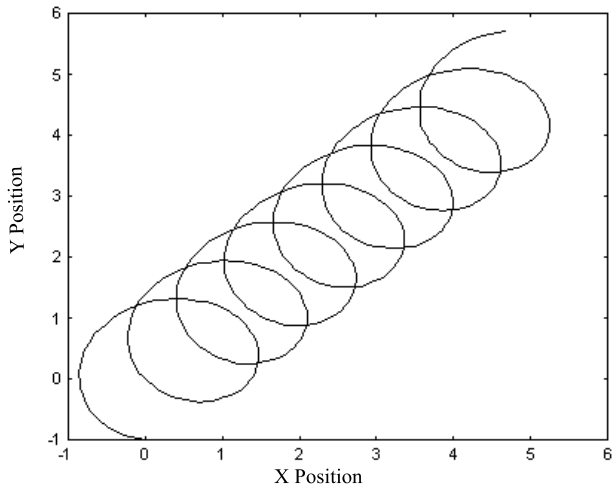

Figure 3: The circling trajectory

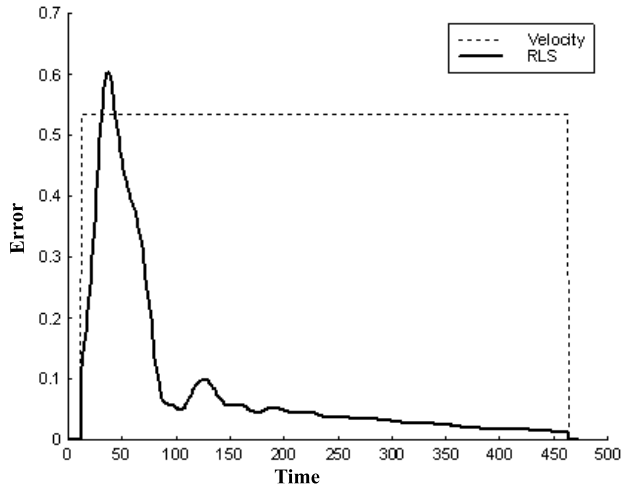

Figure 4: Prediction error on the circling trajectory ($t = 10, n = 2$)

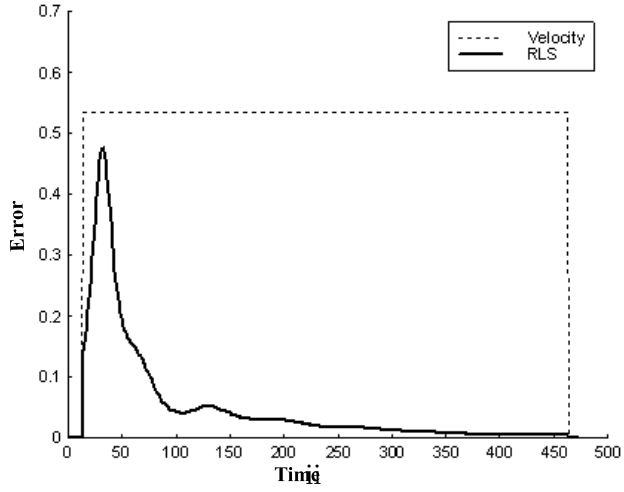

Figure 5: Prediction error on the circling trajectory ($t = 10, n = 3$)

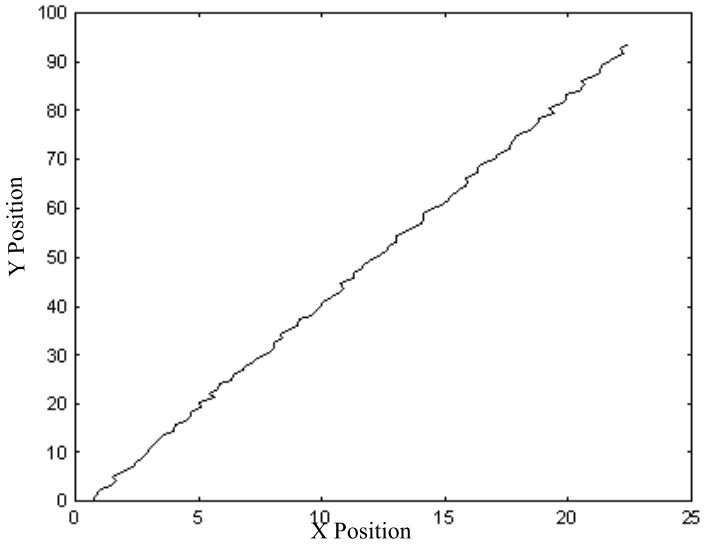

Figure 6: The noisy line trajectory

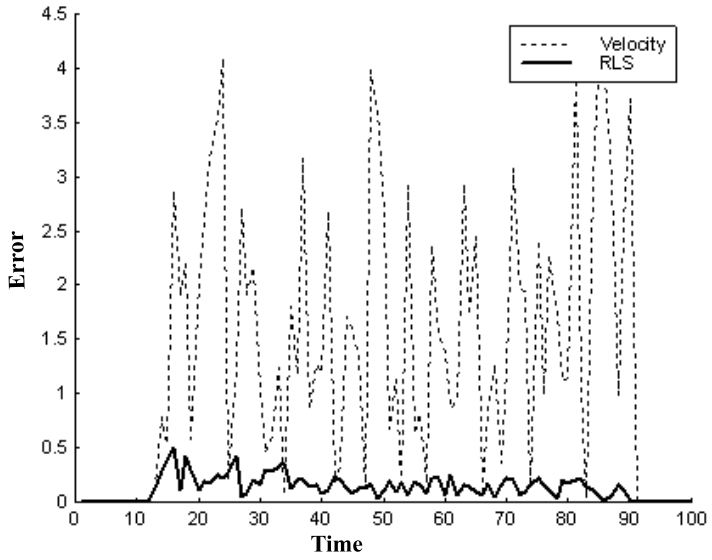

Figure 7: Prediction error on the noisy line trajectory ($t = 10, n = 2$)

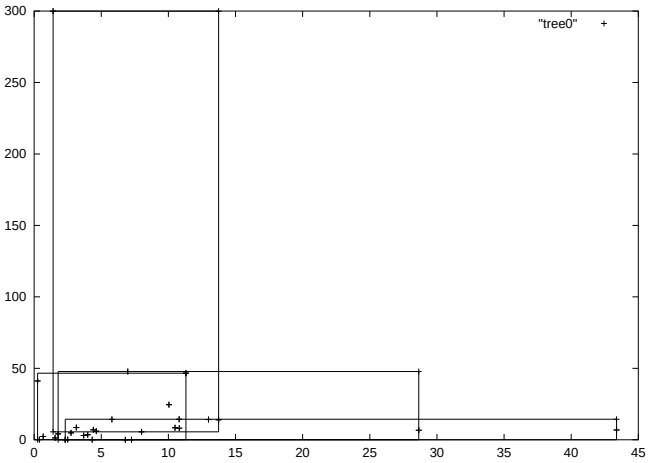

Figure 8: MBR of KR-tree with 30 trajectories