# An Indexing Scheme for Fast Similarity Search in Large Time Series Databases

Eamonn J. Keogh and Michael J. Pazzani
*Department of Information and Computer Science*
*University of California, Irvine, California 92697 USA*
*{eamonn,pazzani}@ics.uci.edu*

## Abstract

*We address the problem of similarity search in large time series databases. We introduce a novel indexing algorithm that allows faster retrieval. The index is formed by creating bins that contain time series subsequences of approximately the same shape. For each bin, we can quickly calculate a lower-bound on the distance between a given query and the most similar element of the bin. This bound allows us to search the bins in best first order, and to prune some bins from the search space without having to examine the contents. Additional speedup is obtained by optimizing the data within the bins such that we can avoid having to compare the query to every item in the bin. We call our approach STB-indexing and experimentally validate it on space telemetry, medical and synthetic data, demonstrating approximately an order of magnitude speed-up.*

## 1. Introduction

Time series account for much of the data stored in business, medical, engineering and social science databases. There are innumerable statistical tests one can perform on time series, such as determining autocorrelation coefficients, measuring linear trends, etc. Much of the utility of collecting this data, however, comes from the ability of humans to visualize the shape of the (suitably plotted) data. For example:

- Cardiologists view electrocardiograms to diagnose arrhythmias.
- Astronomers examine star light curves (frequency change over time) to classify stars [18].
- Chartists examine stock market data, searching for certain shapes, which are thought to be indicative of a stock's future performance [12].

Unfortunately, the sheer volume of data collected means that only a small fraction of the data can ever be viewed.

### 1.1 Sequential Scanning

Finding patterns in a time series database requires a representation of the data, a distance (or similarity) measure and a search technique. We will briefly discuss the Sequential-Scanning (also called linear scanning), the simplest retrieval algorithm and a standard 'strawman' commonly used in the literature [8], [24], [1], [25]. The data *S* is represented as a sequence of real numbers (which we will refer to as 'raw' data for the rest of the paper). The distance between two sequences *S1* and *S2* of length *l* is the Euclidean distance, defined as:

$$D(S1, S2) \equiv \sqrt{\sum_{i=1}^{l} (S1_i - S2_i)^2}$$

The search algorithm depends on how the database is structured. The are essentially two ways the data might be organized:

- *Whole Matching.* Here it assumed that all sequences to be compared are the same length.
- *Subsequence Matching.* Here we have a query sequence *Q*, and a longer sequence *S*. The task is to find the subsequence in *S*, beginning at $S_i$, which best matches *Q*, and report its offset within *S*.

Whole matching trivially requires comparing the query sequence to each candidate sequence by evaluating the distance function and keeping track of sequence with the lowest distance. Subsequence matching requires that the query *Q* be placed at every possible offset within the longer sequence *S*. If *Q* contains *n* datapoints and *S* contains *m* datapoints, then the time complexity is O(*mn*) (assuming $m \gg n$). For some of the queries presented in section 5 of this paper m = 938,105 and n = 2,000. Such a query would require more than 1.8 billion operations.

There is a simple and useful optimization that can be used to speed up Sequential-Scanning. The idea is to keep a *best-so-far* variable that contains the distance associated with closest match encountered thus far. This allows the distance calculation to be abandoned if the partial sum of error ever exceeds *best-so-far*. We call this optimization Branch and Bound evaluation. Even with the Branch and Bound evaluation optimization, Sequential-Scanning is clearly too slow to be practical.

### 1.2 Abstract Representation of Time Series

Abstract representations of time series permit more efficient computation than using the raw time series data and may allow for a more sophisticated search (or indexing) technique. Many different representations have been proposed, including Discrete Fourier Transformations [8], [1], [24], [25]. Relational Trees [22] and envelope matching/R+ trees [2]. Here, we explore indexing with a piecewise linear representation. This representation has numerous advantages. Pavlidis and Horowitz [19] point out that it provides a useful form of data compression and noise filtering. Shatkay and Zdonik [11] describe a method for fuzzy queries on linear (and higher order polynomial) segments. Furthermore, in previous work we have demonstrated a framework for probabilistic pattern matching using linear segments [17] and have shown how a linear segment representation can be used to allow relevance feedback in time series databases [16].

None of the previous work on piecewise linear representations has used an index structure. Speedup is obtained instead because the piecewise linear segmentation is a form of abstraction, and searching over these abstract features is much more efficient than searching over the raw data. Here, we introduce a further abstraction of the piecewise linear segment representation by creating an index that stores the direction but not the magnitude of the change of the variable value over time. We call this representation STB (Shape To Bit-vector). We demonstrate that this representation can be used as index for time series and that it permits more efficient similarity search.

The rest of this paper is organized as follows. In Section 2 we introduce the segmented representation. In Section 3 we introduce our proposed indexing technique and show how we index objects of a fixed length. In Section 4 we consider how to deal with longer and shorter queries. Section 5 contains a discussion of how we select a key parameter, and Section 6 surveys related work.

## 2. Piecewise Linear Representation

There are numerous algorithms available for segmenting time series, many of which where pioneered by Pavlidis and Horowitz [19]. An open question is how to best choose *K*, the 'optimal' number of segments used to represent a particular time series. This problem involves a trade-off between accuracy and compactness, and clearly has no general solution. For this paper, we utilize the segmentation algorithm proposed in Keogh [15]. This method segments a time series, and automatically selects the best value for *K*.

### 2.1 Notation

For clarity we will refer to 'raw', unprocessed temporal data as time series, and a piece-wise representation of a time series as a sequence. We will use the following notation throughout this paper. A time series, sampled at *k* points, is represented as an uppercase letter such as A. The segmented version of A, containing *K* linear segments, is denoted as a bold uppercase letter such as **A**, where **A** is a 4-tuple of vectors of length *K*.

$$\mathbf{A} \equiv \{AXL, AXR, AYL, AYR\}$$

The $i^{th}$ segment of sequence **A** is represented by the line between $(AXL_i, AYL_i)$ and $(AXR_i, AYR_i)$. Figure 1 illustrates this notation and Table 1 contains a summary of the notation used in this paper.

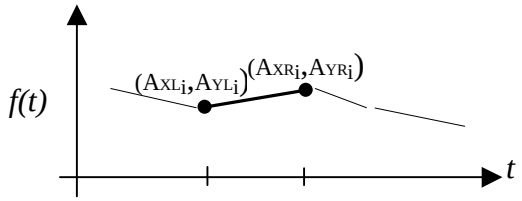



**Figure 1:** We represent times series by a sequence of straight segments

| Symbols | Definitions |
|---|---|
| *S* (upper-case italic) | A time series of real valued data. |
| *k* (lower-case italic) | Number of datapoints in *S* |
| $S_i$ | The $i^{th}$ data point in S ($1 \le i \le k$) |
| | |
| **A** (upper-case bold) | The segmented representation of the time series *A*. |
| *K* | The number of segments in **A**. |
| $AXL_i$, $AXR_i$, $AYL_i$, $AYR_i$ | The X (Y) coordinate of the left (right) side of the $i^{th}$ segment ($1 \le i \le K$). |
| | |
| **Q** | A query sequence |
| *l* | The length of **Q** |
| B | A bitstring which is a label for a bin |
| *b* | The number of bits in B |

**Table 1:** The notation used in this paper

### 2.2 Comparing time series

One of the advantages of using piecewise linear segments is that distance measures can be calculated approximately *k*/*K* times faster than on the raw data. A further advantage is that it possible to define different distance measures customized for different domains [16], [17], [11]. In this paper however, we will use Euclidean distance for simplicity and to facilitate comparisons to other systems.

The distance between two sequences **A** and **B** is denoted as *DS*(A,B), and is designed to closely approximate the distance measure defined on the raw data. To enhance the flow of the paper, we will defer the details of the distance measure to Appendix A. We note here however that *DS* is a metric, and in particular it obeys the triangle inequality.

## 3. Proposed Method

We begin this section with a sketch of the proposed method, followed by a more detailed explanation of each component. To build the index we move a fixed-length sliding window across the data sequence. The window contains an equi-spaced grid as illustrated in figure 2.

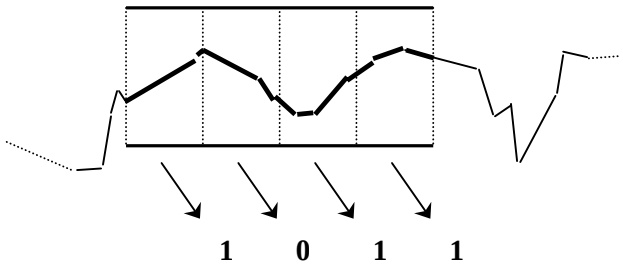



**Figure 2:** An illustration of the declustering technique used in our indexing scheme. The template grid (shown here with just 4 sub-divisions for clarity) is moved across the sequence and aligned at the left edge of every segment. The subsequence that falls within the grid (shown in bold) is examined and used to produce a bitstring. This bitstring is used to determine into which bin a copy of the subsequence should be placed

The section of the time series which falls within each part of the grid is examined and discretized into two possible classes 'up' or 'not-up' which are represented as one or zero respectively. The bit pattern obtained by reading these bits from left to right is used to decide into which bin a pointer to the time series should be placed. After all subsequences have been copied to their respective bins, we compare every subsequence to every other subsequence within the same bin, storing the results in a distance matrix. So after indexing is complete, we have a set of bins, which are labeled by unique bit patterns. Each bin contains a copy of one or more subsequences of the original data, together with a precomputed distance matrix which contains the distance between every possible pair of subsequence in that bin. This distance matrix enables pruning of the search space within the bin by using the triangular inequality [21].

When given a query, the retrieval algorithm achieves sub-linear search times by using three techniques.

1) **Whole bin pruning:** A simple procedure exists that given the current best match, determines whether any element in a particular bin may be more similar to the query. This permits a branch and bound search to eliminate some bins from consideration since they cannot contain a more similar time series.
2) **Bin ordering:** It is possible to order the bins in approximate best-first order. That is, the first bins searched are the ones most likely to contain the best matching sequence. This allows the search to extract the additional benefit from whole bin pruning.
3) **Intra-bin pruning:** After comparing the query to an item in a bin, the triangular inequality can be exploited to possibly prune other items in the bin.

We next consider each step in greater detail.

### 3.1 Building the Index

To build the index it is necessary to decide on the length of the queries that are most likely to be encountered. Some applications, e.g., financial ones, have a common fixed size query such as one week of data. Longer and shorter queries can be dealt with, but require some additional processing as discussed in section 4. Assume the length decided on is *l*. A template grid of length *l* with *b* equi-spaced subdivisions is created (we defer the discussion of how to choose the value of *b* to Section 5, when the reader's intuitions about our overall approach is more fully developed). Each subdivision of the grid is called a "window". This grid is placed at the leftmost edge of every segment in **S** and the subsection of **S** that falls within the grid is examined. Figure 2 illustrates this notation. Note that some segments may straddle a gridline between two windows. These segments are 'broken' at the gridline, so that each window contains a unique set of segments. The $k^{th}$ window contains a set of these segments that we denote as $\mathbf{W_k}$, $1 \le k \le b$. Clearly these k sets are mutually exclusive and their intersection is simply the entire sequence within the template grid.

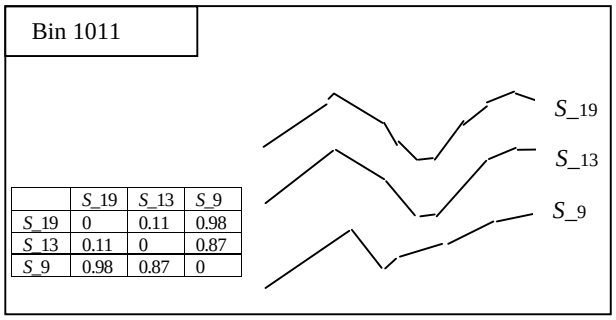
All *b* sets of segments are examined and used to produce a bitstring B of length *b*. Intuitively, the $k^{th}$ bit is a one if the set of segments in $\mathbf{W_k}$ are 'mostly rising', and zero if the set of segments in $\mathbf{W_k}$ are 'mostly falling' or constant. More formally:

$$B_k = \begin{cases} 1, \sum_i WkYR_i - WkYL_i > 0 \\ 0, \qquad otherwise \end{cases}$$

Note, at first glance it may appear that a faster way to evaluate the above function would be to simply examine y-value difference of the left endpoint of the leftmost segment and the right endpoint of the rightmost segment to determine the parity of the bit. In most cases this shortcut would give the same answer, however it does not hold in general because we do not require that the endpoints of consecutive segments line up (as in spline-base representations).

Given that we can produce a bitstring representation from each subsequence that appears in our template grid, we use that bitstring to determine into which bin to place a copy of that subsequence. Rather than creating all $2^b$ possible bins at the start of the indexing algorithm, bins are only created as needed. This is to prevent storing deadspace (i.e. empty bins). When the algorithm is finished distributing all subsequences of length *l* into bins,

the processing of the individual bins can take place. This step simply consists of comparing every pair of items and precomputing a distance matrix for the bin. Figure 3 illustrates the structure of a bin.




| | S_19 | S_13 | S_9 |
|---|---|---|---|
| S_19 | 0 | 0.11 | 0.98 |
| S_13 | 0.11 | 0 | 0.87 |
| S_9 | 0.98 | 0.87 | 0 |

**Figure 3:** An example of a bin. Each bin has a unique bitstring as a label. It contains a copy of one or more subsequences from the original sequence been indexed, together with a distance matrix containing the distance between each pair of subsequences

## 3.2 Bin Ordering

In Section 1 we discussed the Branch and Bound evaluation optimization that can speed up Sequential-Scanning. It is trivial to modify this optimization to work with linear segments. The distance function *DS* is simply passed an extra parameter, the current *best-so-far*. When summing up the individual contributions of error from each segment, the algorithm constantly checks to see if the sum exceeds the *best-so-far*. If so, the algorithm can abandon the comparison because the current candidate could not possibly be a closer match than the best current match. Note that the utility of this optimization is highly variable. In the best possible case, the very first item you compare to the query might be a perfect match, in which case you can abandon the search almost immediately. In the worst (highly pathological) case, it is possible that no benefit can be extracted from Branch and Bound evaluation, and that the extra overhead will slow the search slightly.

To extract a significant benefit from Branch and Bound evaluation, a good match to the query should be evaluated first. It is possible to exploit the STB representation to present the candidate subsequences in an *approximate* best first order.

We begin by placing a template grid over the query **Q**. This template is exactly the same as the one used to build the original index. As before, we 'break' segments that cross a gridline to produce *b* sets of segments. We denote the $k^{th}$ set of these segments as $\mathbf{Q_k}$, $1 \le k \le b$. We can now examine these sets of segments to produce a vector *R* of length *b*. Intuitively the $k^{th}$ element of the vector contains a real number that represents the net amount the query rises within the $k^{th}$ window. More formally:

$$R_k = \sum_i QkYR_i - QkYL_i$$

Naturally, if $k^{th}$ set of segments mostly have downward trends, $R_k$ will be a negative number. Additionally, if within a single window, the segments go up and down (i.e. a peak or valley), the two trends will tend to cancel each other out and $R_k$ will be close to zero.

Given the vector *R* derived from some query **Q**, and a bitstring B, which is a label for some bin, we can quickly calculate the lower-bound between the query and any candidate match in that bin. The formula is:

$$\min\_ dis(R,B) = \sum_i |R_i| * agree(B_i R_i)$$

$$agree(B_i,R_i) = \begin{cases} 1, & if\ ((B_i = 1)\ and\ (R_i > 0))\ or\ ((B_i = 0)\ and\ (R_i < 0)) \\ & else\ \ 0 \end{cases}$$

This formula allows us to quickly calculate the best order in which to present the bins to the search algorithm. We begin by initializing a heap which can store bitstrings of length *b*. We take the list of all bin labels $B_n$ that were used building the index, and insert these labels onto the heap using min_dis(*R*,$B_i$) as the heap-key for the $i^{th}$ bin. The order, in which we remove these labels from the heap, is the order in which we retrieve and examine the bins. Figure 4 illustrates how effective this simple strategy is:

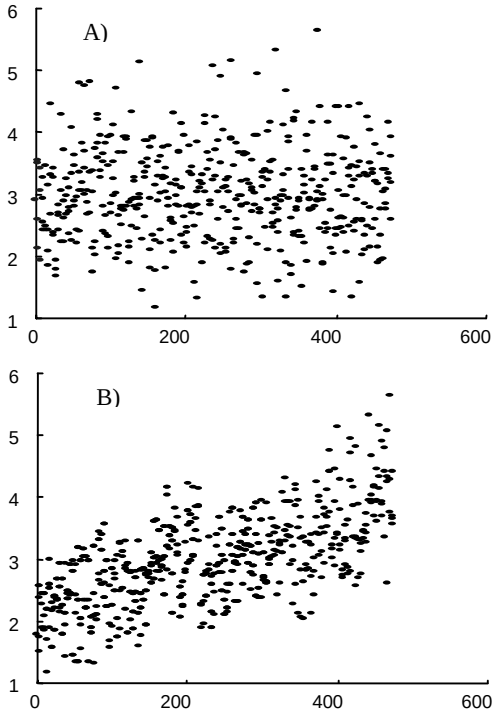



**Figure 4:** An illustration of the effectiveness of our ordering technique. Each point represents one of 500 sequences that were compared to a query. The Y-axis represents the distance, and the X-axis represents the order in the sequences where evaluated, from left to right. **A)** Shows the sequences as they where evaluated in order, by Sequential-Scanning. **B)** Displays the sequences as they evaluated in the order produced by the lower bound ordering technique. The ordering technique produces a strong positive correlation between the order in which the sequences are examined and the quality of the match

### 3.3 Whole-Bin Pruning

Given that we are able to order the bins *best-first* and that function exists which can determine the minimum distance between a query and any of the items contained in a given bin, we can easily prune bins from the search space. When the search algorithm pops the label of the next bin to be examined from the heap, it checks the following condition. Is min_dis(*R*,*B*) ≥ *best-so-far* ? If so, the search can be abandoned, and the current best match returned. Note that for this technique to perform significant pruning, it is important to find good matches quickly. This pruning technique can prune a bin just by examining its label, it is not necessary to access the disk on which the bin resides.

### 3.4 Intra-Bin Pruning

Whole-bin pruning allows the search algorithm to prune entire bins; we can also exploit the triangular inequality to (possibly) prune some items in a bin once we have examined one or more items within that bin.

Suppose we have *best-so-far* ≤ *DS*(**Q**,**A**) – *DS*(**A**,**B**) and *DS*(**Q**,**A**) - *DS*(**A**,**B**) ≤ *DS*(**Q**,**B**). If we add the two inequalities and simplify, we have *best-so-far* ≤ *DS*(**Q**,**B**), which tells us that the sequence **B** is not a better match to our query **Q**, than the current *best-so-far*. However, *DS*(**Q**,**A**) - *DS*(**A**,**B**) ≤ *DS*(**Q**,**B**) is simply a rearrangement of the definition of the triangular inequality. Therefore a sufficient condition for pruning a sequence **B** is:

*best-so-far* ≤ *DS*(**Q**,**A**) – *DS*(**A**,**B**) ⇒ **B** can be pruned

Because we have precomputed the distance between all items in the bin, we can utilize this rule after examining each sequence in the bin. Table 2 contains an outline of the intra-bin pruning algorithm.

```
function in-bin-search(Q, bin, best-so-far)
  let C be the set of items in the bin
 while C ≠ ∅
  use a heuristic to pick an object C_i in C
  update(C_i, Q, best-so-far)
  C := {(C_k| DS(Q,C_i) – DS(C_i,C_k) < best-so-far) ∧
                        DS(Q,C_k) is not computed)}
 end
```

**Table 2:** (Modified) Burkhard and Keller algorithm

After comparing the query **Q** to item $\mathbf{C}_j$ in a bin, and finding them to be a distance *D* apart. The search algorithm next examines the $j^{th}$ column of the bin matrix. The row indices of elements in the column which are less than *D* - *best-so-far* refer to items in the bin that can be pruned.

For example, consider the bin shown in figure 3. Suppose we have a current *best-so-far* of 0.10 and we compare the query sequence **Q** to the item *S*_13, finding *DS*(**Q**,*S*_13) to be 1.0. Because precomputed distance between S_13 and S_19 is 0.11 and 1.0 – 0.10 > 0.11, we can prune the item *S*_19. We can also prune item *S*_9 because 1.0 – 0.10 > 0.87.

Note that in the above example, the order in which we examine items affects the efficiency of the pruning. If we had looked at *S*_9 first, we could have pruned *S*_13 but not *S*_19. In general the optimal item to examine first is the most central item. This item can be quickly found by finding the column of the distance matrix with the minimum sum.

It might be suggested that this technique of in-bin pruning could be used by itself as a standalone technique (A similar idea has been suggested by Smeaton and Van Rijsbergen for small text databases [23]). There are two reasons why we do not do this.

1. The time and space complexity of the index would be increased. The increase is only a constant, but it is a large constant, at least 4 or 5 orders of magnitude.
2. The efficiency of this technique depends on how quickly you find a good match. If reasonably close matches to a given query are very common in the dataset this is not a problem. If however, there are few matches that are even close to the query in the dataset, the 'single-bin' technique will be very inefficient at pruning. In contrast our ability to order bins ensures that we will retrieve reasonably good matches quickly, thus extracting the maximum benefit from this technique.

A review of the search algorithm is presented in table 3. The parameter `bin_labels` is simply a list of the bin labels created while building the index, together with pointers to the bins themselves.

```
procedure search(bin_labels)
initialize best-so-far to infinity
get query shape Q from user
let R be the discretization vector obtained from Q
                        // Defined in section 3.2
place bin_labels onto heap using min_dis(R,B) as
heap key
                        // Defined in section 3.2
   do
      remove B from top of the heap
      retrieve corresponding bin
          in-bin-search(Q, bin, best-so-far)
                            // Defined in table 2
   while best-so-far > min_dis(R,B)
 end
```

**Table 3:** An outline of the search algorithm

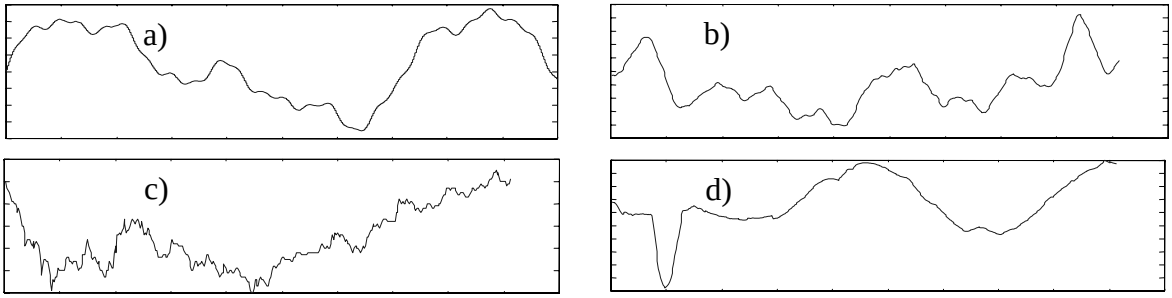



**Figure 5:** Samples of the datasets used for experimental evaluation. Each plot contains approximately 5% of the entire dataset. **a)** Synthetic **b)** Photon **c)** Financial **d)** Shuttle.

### 3.5 Experimental results

To evaluate the indexing scheme proposed in this paper, we performed experiments on real and synthetic data. Samples of each dataset are plotted in figure 5. All data used is publicly available at the UCI repository.

- **Financial**: The US Daily 5-Year Treasury Constant Maturity Rate over 24 years (1972-96).
- **Photon**: A dataset that describes the "density" of photon arrivals over a period of one tenth of a second [20].
- **Synthetic**: Ten artificially generated datasets, each consisting of 4 sinusoids (with different aptitudes and frequencies). This dataset was used to challenge the linear segment representation in a domain that is highly spectral. The data are also interesting because they have structure at different resolutions. The data were generated by the function:

$$\bar{y} = \sum_{i=3}^{7} \frac{1}{2^i} \sin(2\pi(2^{2+i} + rand(2^i))\,\bar{t}) \qquad 0 \le \bar{t} \le 1$$

  Where *rand*(x) is a random integer between zero and x.
- **Shuttle:** This dataset consists of ten time series that are a subset of the sensors that describe the orientation of the Space Shuttle during the first eight hours of mission STS-57.

The Financial and Photon datasets were tested using holdout. That is, the small subsection of the data that was used as the query was removed before the index was built. This process was repeated 10 times, and the results averaged.

The Synthetic and Shuttle datasets were tested slightly differently. In both cases we had 10 time series of equal length, we used all the data from one time series to build the index, and selected the query at random from one of the other 9 series. We did this for each of the 10 time series and average the results. We chose this method of testing for the Synthetic and Shuttle datasets, because otherwise the query might have had a very close match in the index, producing optimistic results.

We compared the following search techniques:

- Sequential scanning using the raw data (**SSR**). Optimized with Branch and Bound evaluation.
- Sequential scanning using the segmented data (**SSS**). Optimized with Branch and Bound evaluation.
- The STB-indexing algorithm proposed in this paper.

Table 4 contains the results:

| | Size of dataset | Number of segments | Time **SSR** (seconds) | Time **SSS** (seconds) | Time STB (seconds) | Indexing speedup |
|---|---|---|---|---|---|---|
| Financial | 8,749 | 842 | 202.92 | 23.73 | 4.30 | **5.64** |
| Photon | 938,105 | 1773 | 413,901.00 | 151.78 | 18.82 | **8.06** |
| Synthetic | (10*) 100,000 | (mean) 531 | 18,084.60 | 14.99 | 1.23 | **12.18** |
| Shuttle | (10*) 10,000 | (mean) 47 | 154.39 | 6.19 | 0.67 | **9.23** |

**Table 4:** Experiment results.

## 4.0 Dealing with Longer and Shorter Queries

As discussed in section 3.1, the index is built for queries of a particular, fixed length. Obviously it is necessary to be able to deal with queries of different lengths. Our indexing scheme has a storage redundancy of 1 [13], but this in relation to the segmented representation. The storage redundancy in relation to the original 'raw' data is much smaller that 1. This very small space overhead gives us the luxury of simply building multiple indices for different length queries. This approach may particularly make sense in certain domains. For example, Chartists are typically only interested in weekly, fortnightly and monthly trends. Nevertheless in some domains we may need a general method to be able to handle queries of arbitrary lengths.

### 4.1 Dealing with Queries Longer Than *l*

The problem with dealing with a query **QLONG**, which is longer than *l*, is that our indexing scheme only 'knows' about queries of length *l*. The intuition behind our solution is as follows. We extract the prefix of the query, i.e. the query up to length *l*. We then locate the K best matches to the prefix in feature space and test to see if that region of feature space also contains the best match to the original query. If it does not, we keep expanding the region of feature space until it does. Table 5 contains an outline of the algorithm.

```
long_query_search(Original_Sequence,QLONG,K,l)
  Extract PREFIX of length l from QLONG
  let Kbest be a list of the K best matches to
  PREFIX, sorted by distance

for i to size(Kbest)
   let CANDIDATE be the sub-sequence of the
Original_Sequence pointed to by Kbest_i
   update(CANDIDATE, QLONG, best-so-far)
 if any items in Kbest have a distance greater
                                 than best-so-far
        remove those items from Kbest
        best_full_length_match := CANDIDATE
        done := true
 endif
endfor

if done
   return best_full_length_match
else
   long_query_search(Original_Sequence,QLONG,2*K,l)
```

**Table 5:** Algorithm for searching for queries longer than *l*

We find the K best matches to the prefix (where K is a small integer in the range of approximately 2 to 5) and use the information returned as a guide to searching the original data, which 'knows' about queries of arbitrary lengths. If any of the matches between the full-length query and the original data has a distance less than the distance between the prefix its $i^{th}$ best match ($1< i \leq K$), then we are guaranteed that the best match overall does not have a prefix greater than i. If that is not the case, we simply double the value of K and try again.

In Section 4.3 we perform experimental validation of this method.

### 4.2 Dealing Queries Shorter Than *l*

Dealing with a query **QSHORT**, shorter that *l* is easier, and requires only one minor modification to the indexing scheme. As before we place a template grid over the query. But this time we use a slightly different function to build the vector *R*.

$$R_k = \sum_i \begin{cases} 0, \; if \;\; QkYR_i > length(QSHORT) \\ QkYR_i - QkYL_i, \quad otherwise \end{cases}$$

We can then use the vector *R* to evaluate the function min_dis as before. Normally, the bin-ordering technique produces a unique ordering of the bins. Using a shorter query will tend to produce several bins tied for first place, several bins tied for second place etc.

When finding the distance between **QSHORT** and an item in the bin, we first truncate the item to length *l*. We can then use the in-bin pruning technique unmodified. The in-bin pruning technique exploits the triangle inequality by assuring:

$$best\text{-}so\text{-}far \leq DS(\mathbf{Q},\mathbf{A}) - DS(\mathbf{A},\mathbf{B}) \Rightarrow \mathbf{B} \text{ can be pruned}$$

Knowing $DS(\mathbf{Q},\mathbf{B}) > best\text{-}so\text{-}far$ allows us to remove **B**, unexamined, from the search space. We can still use this rule with our shorter query because $DS(trunc(\mathbf{Q}),trunc(\mathbf{A})) \leq DS(\mathbf{Q},\mathbf{A})$ for any amount of truncation.

The same reasoning allows us to use the whole-bin pruning technique unmodified, although both whole-bin pruning and in-bin pruning become less efficient as the query becomes shorter, because they become increasingly conservative. Below we show experimental confirmation of this.

### 4.3 Experimental results

We performed additional experiments to test the effect of using queries that were longer or shorter than *l*, the index length. We indexed the dataset Photon, with *l* = 2,000 (as measured using the original datasets granularity). We ran queries of different lengths, from 50 to 200 percent of *l*, in 25% increments. Figure 6 shows the results. The top line shows that sequential scanning has a time complexity linear in the length of the query. The bottom line, in contrast, shows that our scheme works best for the query

length for which the index was built. As the query gets longer or shorter the performance degrades toward that of sequential scanning.

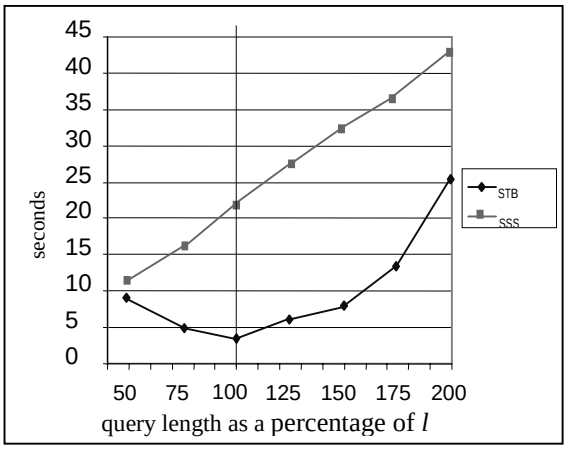



**Figure 6:** The degradation of performance as the query length deviates from *l*.

## 5 Discussion

As noted by Faloutsos et al. [8], there are several highly desirable properties for any indexing scheme:

- It should be much faster than sequential scanning.
- The method should require little space overhead.
- The method should be able to handle queries of various lengths.
- The method should be dynamic, which is to say it should allow insertions and deletions without requiring the index to be rebuilt.
- It should be correct, i.e. there should be no false dismissals (although probabilistic matching may be acceptable in some domains).

Our indexing scheme has all the above properties, however the last item requires careful qualification. Our indexing scheme is guaranteed to produce no false dismissals with respect to the segmented representation; however, we cannot make the same claim with respect to the original data. Because such an event requires pathological conditions, and was never observed during any of our experiments, we do not feel this to be an important limitation. However, we will briefly discuss an extension to our representation that would allow us to modify our indexing scheme to guarantee no false dismissals with respect to the original data.

The segment representation of the data can be extended from a 4-tuple to a 5-tuple, where the extra element is the residual error from approximating the line [15] i.e.

$$\mathbf{A} \equiv \{A_{XL}, A_{XR}, A_{YL}, A_{YR}, A_{\varepsilon}\}$$

The distance measures stored in the index can be modified to include a confidence bound *cb*, which is simply the sum of all the residual error terms from all the segments being compared. An initial run of the indexing scheme can be used to find **BM**, the best (segmented) match to a query. This best match can be use to index the raw data, and the actual Euclidean distance between the query and the raw data BM, can be computed. The index is revisited and the set of all sequences such that $D(Q,BM) > DS(\mathbf{Q},\mathbf{C}_i) - cb_i$ are returned. We are guaranteed that the raw data analogues of these sequences contain the best match with respect to the original data.

Up to this point we have deliberately neglected to discuss how the parameter *b* is chosen; we will now repair this omission. To begin it may be helpful to consider the two extremes. If $b = 1$, all sequences will fall into one of two bins. This is unacceptable in terms of space complexity. Even if we could afford the time and space to construct two huge bins, search within the first bin cannot fully take advantage of our in-bin pruning technique, which relies on the assumption that a reasonably good match will be found early. We therefore we may be forced to examine every candidate sequence in the first bin.

If *b* is large enough, our bin ordering and whole-bin pruning techniques become maximally efficient, however the time taken to order the bins grows dramatically. Essentially we just pushed the complexity of sequential scanning to a different part of the algorithm.

The value of *b* affects the vector *R*. As *b* get large the sum $\Sigma|R_i|$ tends to grow. In general this is useful because the larger this sum the more efficient whole bin pruning is. The problem with having too large a value for *b* is that a single segment is likely to span multiple windows (as in figure 7.1). We call these windows redundant because they could be replaced by a single window without affecting the value of $\Sigma|R_i|$. Having redundant windows increases overhead without producing any speedup.

As we decrease the value of *b* to alleviate this problem, we are faced with another problem. The number of windows that contain a single segment (covered segments) tends to decrease, while the number of windows which

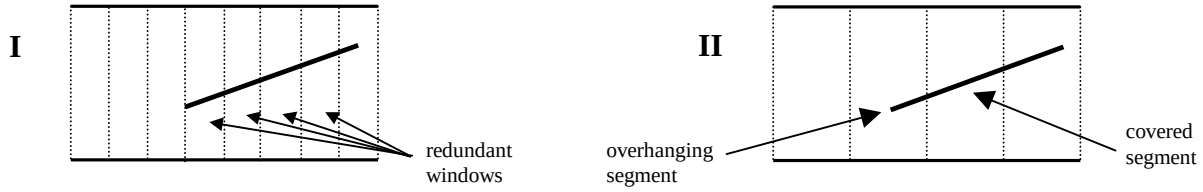



**Figure 7: I**) If *b* is too large we have the problem of redundant windows, i.e. multiple windows which could be replaced by a single window without affecting the sum $\Sigma|R_i|$. **II**) An illustration of the terms "overhanging segment" and "covered segment".

contain at least two segments (overhanging segments) tends to increase (see Figure 7 for an illustration of these terms). In general, overhanging segments are bad, because their contribution to $R_I$ may be cancelled by the other segment(s) in the same window which have the opposite slope.

Ideally we would like to choose $b$ such that we minimize the ratio of overhanging segments to covered segments. The analysis is complicated by the fact that, apart from the leftmost segment, the segments do not generally line up with windows. Additionally, the segments can have arbitrary lengths. For simplicity, let us consider only the worst case, for segments of mean length μ. The length of overhanging segments is given by μ mod $l/b$ and the length of covered segments is given by μ div $l/b$. To guarantee we have no redundant bins we must have μ div $l/b < 2$, additionally we must have $\mu < l/b$ to ensure that we have at least one covered segment. So the problem is to find:

$$\arg\min\left(\frac{\mu \bmod l/b}{\mu \operatorname{div} l/b}\right), \mu < l/b < 2\mu$$

Given the constraints on the value of $l/b$, the denominator is a constant. So to minimize the expression we simply need to minimize μ mod $l/b$, which goes to zero as the ε in $l/b = 2(\mu-\varepsilon)$ goes to zero. Therefore we have $b = l/2(\mu-\varepsilon)$.

Experimental evaluation (omitted for brevity) confirms that this value is a good choice for $b$. Additionally, we found that the value is not too critical; we can vary $b$ by ± 15% without any noticeable effect.

## 6. Related work

Fast similarity search in the time series domain requires a high level representation of the data and an indexing scheme. Below we consider some of the alternative approaches proposed.

### 6.1 High Level Representations

In general, pattern matching on 'raw data' is not feasible because of the sheer volume of data. In addition, raw data may contain spikes, dropouts or other noise that could confuse the matching process. A variety of higher-level representations of times series have been proposed, most notably the Discrete Fourier Transform. This approach involves performing a Discrete Fourier Transform on the original time series, discarding all but the $K$ most informative coefficients, and then mapping these coefficients into $K$-dimensional space. The original work by Agrawal, Faloutsos and Swami [1] only allowed the comparison of two time series of equal length, but was extended by Faloutsos, et al. [7] to include subsequence matching.

An important feature of the various higher level representations is their ability to compress the data. Not only can this reduce disk storage requirements, but similarity searching over the compressed representation is likely to be much faster. The best representation is obviously data dependent. In highly spectral datasets, Discrete Fourier transforms work extremely well, but in datasets which have a weak spectral signature they must either produce a coarse representation or a low compression rate. Piece-wise linear segments have the opposite problem. In a highly spectral domain they must either produce a course representation or approximate the signal with many short segments, thus achieving a low compression rate. In general, across many real datasets, we have found that piece-wise linear segments do at least as well at approximating the data as Discrete Fourier transforms. To illustrate this we took a time series that visually seems to contain both spectral and linear elements, and produced approximations of it using DFT and piece-wise linear segments using the same compression rate for each. Figure 8 illustrates the results.

Others, including Shatkay and Zdonik [11] recognize that a piece-wise linear representation greatly reduces the required storage and search space for a time series, but they do not suggest a robust distance measure or indexing technique.

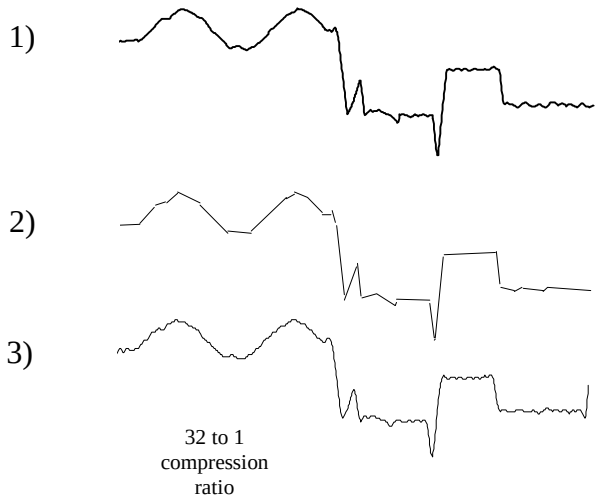



**Figure 8:** A comparison of the representational power of linear segments and Fourier transforms **1)** An inertial navigation sensor from Shuttle Mission STS-57 **2)** A piece-wise representation **3)** A discrete Fourier transform representation.

### 6.2 Indexing Techniques

Shasha and Wang [21] proposed a very general indexing method called Approximate Distance Map (ADM). ADM uses the triangular inequality combined with precomputed distances to prune the search space.

Agrawal et al. (1993) considered this method for time series, but noted that the space overhead is quadratic in the number of objects. In this paper we used a modified version of Shasha and Wang's indexing scheme as one of the elements of our overall approach. We are able to utilize a modification of this method in spite of its high space complexity, because our representation of the data combined with the declustering technique discussed above decreases the space complexity by a large constant factor. Although the space saving is only a constant factor, it is an extremely large constant. On the datasets used for experimental validation this constant ranged from 107,966 to 71,668,018.

A general indexing technique that has been successfully applied to different kinds of multimedia objects is R-tree indexing (and its many variants and extensions [10]). The basic idea is to map the objects in question into Euclidean space such that the distance between two objects in Euclidean space is closely related to the distance between those objects using the chosen distance measure. Similarity search then corresponds to a range search over Euclidean space. We considered using this technique to index the piecewise linear representation, but chose not to do so for two reasons:

1) In order to guarantee that the method does not allow false dismissals, we must ensure that the distance in Euclidean space between two objects is less than or equal to the distance between the objects using the chosen distance function. In order to guarantee this using the segmented representation, we must either have a very large number of dimensions or come up with a "pessimistic" Euclidean measure (i.e. use the square root of the actual measure as [25]. The problem with the former is that the performance of indexing trees deteriorates for high dimensionalities. The problem with the latter is that it introduces many false hits that must be pruned by a separate technique.
2) Although we only consider the indexing problem in this paper. We are ultimately interested in supporting a variety of datamining algorithms. For example [6] discuss a method for rule induction from time series. The method is demonstrated on small datasets and does not scale well. They suggest "the hierarchical piecewise linear representation may provide a computationally efficient way to increase the expressive power of the underlying signal". In fact, we not only believe this to be the case, but we also intend to show in future work that the data structures created by our indexing scheme will allow more accurate and robust rules to be discovered.

Another class of indexing structures explicitly utilizes dimensionality reduction (DR). The idea is that the data is mapped to a lower dimension and the lower dimension representation is then indexed. Various dimensionality reduction techniques are used, including SVD (Kanth et. al 1998) and Fastmap (Faloutsos & Lin, 1995) and the KL-transform (Fukunaga 1990). We can consider the piece-wise linear structure a dimensionally reduced form of the original data, however it is not clear if DR techniques can be made to work with our representation. First, the segments are generally of different lengths, so it may be difficult to define exactly what constitutes a dimension. We could consider forcing all segments to be the same length, but if we choose a relatively short fixed-length we gain little benefit from dimensionality reduction. Choosing a longer fixed-length means that some data sequences may be very coarsely represented. This will result in the problem of false dismissals.

## 7. Conclusion

We have introduced an indexing scheme that allows indexing of time series represented by piece-wise linear segments. A more general contribution is demonstrating how Shasha and Wang's ADM scheme can have its high space complexity mitigated by a good representation of the data, combined with a technique that clusters (bins) similar data together before the precalculation of distance measures.

Future work includes extending the representation to allow a weighted representation (where different parts of the query are allowed to have differing relative importance when finding matches) and more generalized distance measures. Additionally, we hope the speedup obtained by STB-indexing will allow us to support a variety of time series datamining algorithms that scale poorly to large datasets, for example the rule induction algorithm proposed by Das et al (1998).

## 8. References

[1] Agrawal, R., Faloutsos, C., & Swami, A. (1993). Efficient similarity search in sequence databases. *Proc. of the 4th Conference on Foundations of Data Organization and Algorithms*, Chicago, October.

[2] Agrawal, R., Lin, K. I., Sawhney, H. S., & Shim, K. (1995). Fast similarity search in the presence of noise, scaling, and translation in times-series databases. In *VLDB*, September.

[3] Box, G. P., & Jenkins, G.M (1970). Time series analysis, forecasting and control. San Francisco, Ca. Holden-Day.

[4] Buckhard, W. A, Keller, R. M. (1973). Some approaches to best-match file searching. Commun. ACM 16, 4 April, pp. 230-236.

[5] Cheng, Y. C., & Lu, S. Y. (1982). Waveform correlation using tree matching. *IEEE Conf. PRIP*.

[6] Das, G. Lin, K. Mannila, H. Renganathan, G. & Smyth, P.(1998). Rule Discovery from Time Series. In *Proceedings of the 3rd International Conference of Knowledge Discovery and Data Mining*. pp 16-22, AAAI Press.

[7] Faloutsos, C., & Lin, K. (1995). Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. ACM SIGMOD Conf.*, pp 163-174.

[8] Faloutsos, C., Ranganathan, M., & Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD Conf.*, Minneapolis, May.

[9] Fukunaga. K. (1990). Introduction to Statistical Pattern Recognition. Academic Press, second edition.

[10] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf*., pp 47-57.

[11] Hagit, S., & Zdonik, S. (1996). Approximate queries and representations for large data sequences. *Proc. 12th IEEE International Conference on Data Engineering*. pp 546-553, New Orleans, Louisiana, February.

[12] Hayes, M. (1997). The Dow Jones-Irwin Guide to Stock Market Cycles. Dow Jones-Irwin, Homewood, Illinois.

[13] Hellerstein. J. M., Papadimitriou, C. H., & Koutsoupias, E. (1997). Towards an Analysis of Indexing Schemes. *Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, May.

[14] Kanth, K.V., Agrawal, D., & Singh, A.(1998). Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *Proc. ACM SIGMOD Conf*., pp. 166-176

[15] Keogh, E. (1997). Fast similarity search in the presence of longitudinal scaling in time series databases. In *Proceedings of the 9th International Conference on Tools with Artificial Intelligence*. pp 578-584. IEEE Press.

[16] Keogh, E., & Pazzani, M. (1998). An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. *Proceedings of the 4rd International Conference of Knowledge Discovery and Data Mining*. pp 239-241, AAAI Press.

[17] Keogh, E., & Smyth, P. (1997). A probabilistic approach to fast pattern matching in time series databases. In *Proceedings of the 3rd International Conference of Knowledge Discovery and Data Mining*. pp 24-20, AAAI Press.

[18] Ng, M.K., & Huang, Z. (1997). Temporal data mining with a case study as astronomical data analysis. *Lecture Notes in Computer Sciences*. Springer. pp. 2-18.

[19] Pavlidis, T., & Horowitz, S., (1974). Segmentation of plane curves. *IEEE Transactions on Computers*, Vol. C-23, No 8, August.

[20] Scargle, J. (1998). Studies in astronomical time series analysis: v. Bayesian blocks, a new method to analyze structure in photon counting data. Astrophysical Journal, Vol. 504.

[21] Shasha, D., & Wang, T. L., (1990). New techniques for best-match retrieval. ACM Transactions on Information Systems, Vol. 8, No 2 April 1990, pp. 140-158.

[22] Shaw, S. W. & DeFigueiredo, R. J. P. (1990). Structural processing of waveforms as trees. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 38 No 2 February.

[23] Smeaton, A. F., & Van Rijsbergen, C. J. (1981). The nearest neighbor problem in information retrieval: An algorithm using upperbounds. *ACM SIGIR Forum 16*, pp 83-87.

[24] Refiei, D., & Mendelzon, A. (1997). Similarity-Based queries for time series data. In *Proc. ACM SIGMOD Conf.*, pp. 13-25, Tucson, Arizona.

[25] Yi, B,K., Jagadish, H., & Faloutsos, C. (1998). Efficient retrieval of similar time sequences under time warping. *IEEEE International Conference on Data Engineering*. pp 201-208 Orlando, Florida, Feb.

## Appendix A

One of the advantages of using piecewise linear segments is that it possible to define different distance measures customized for different domains. For this paper however, we have used Euclidean distance for simplicity and to facilitate comparisons to other systems.

The distance measure *DS* is designed to closely approximate the distance measure defined on the raw data. In fact, if the segmented representation approximates the raw data A and B with a residual error of ε, we have:

$$D(A,B) - \varepsilon \le DS(\mathbf{A},\mathbf{B}) \le D(A,B) + \varepsilon$$

As ε goes to zero this reduces to $DS(\mathbf{A},\mathbf{B}) = D(A,B)$. On most real datasets ε is nonzero, but very small.

It is convenient for notational purposes to assume that the endpoints of the two sequences being compared are aligned in the X-axis. In general, with real data, this is not the case. So we will have to process the sequences first, by breaking some segments at a point which corresponds to an endpoint in the other sequence. Additionally, if any of the segments in one sequence cross a segment in the other sequence we break both segments at the crossing point. This can be accomplished in O(*K*). Figure A.a illustrates this process. Each pair of segments that are aligned in the X-axis is called a slice. Below we show how to measure the Euclidean distance between the pair of segments in a slice. The overall distance between two sequences is simply the square root of the summation over all such slices.

The intuition behind the distance measure is to imagine that the two segments in a slice have been replaced with raw data, sampled at original rate. Figure A.b illustrates this idea, with phantom lines drawn between each corresponding pair of virtual datapoints. We need to sum the squares of the lengths of all such lines. Rather than actually perform a summation, we use an equivalent closed form equation.

Let *L* be the absolute difference in Y-axis values between the left endpoints of the two segments. Let *R* be the absolute difference in Y-axis values between the right endpoints of the two segments (note that either or both of *L* and *R* could be zero). Let *x* be the length of the slice, as measured in the units of the original time series (i.e., if the original time series had one observed value for each day, then *x* should be measured in days). Let *c* be min(*L*,*R*) and let $\Delta = (\max(L,R) - c\ ) / x$.

The sum of squares between the two segments in the i[th] slice is: $sumofsquares_i = (x+1)\left(c^2 + c\Delta x + \frac{1}{6}\Delta^2 x(2x+1)\right)$

And the Euclidean distance between two sequences is:

$$DS(A,B) = \sqrt{\sum_{i \in slice} sumofsquares_i}$$

Note that we can perform various optimizations on the above. If the slice was created because two segments crossed then $c = 0$ and the terms $c^2$ and $c\Delta x$ do not need to be calculated. Additionally, *x* is always a positive integer less than the overall length of the sequences been compared, so we can precalculate $(x+1)$ and $x(2x+1)$ for all possible values and use table lookup rather than repeatedly recalculating the same expressions.

For completeness we include the derivation of *sumofsquares*. The key observation is that all $x+1$ "virtual" lines differ in length from their neighbors by Δ. The shortest one has a length of *c*, its neighbor has a length of $c + 1\Delta$ , the next one has a length of $c + 2\Delta$ etc. We want to calculate the sum of squares between the two segments in a slice:

$$sumofsquares = \sum_{i=1}^{x+1} (virtual\ lines)^2$$

Given the above, we can rewrite this as:

$$\sum_{i=1}^{x+1} (c + (i-1)\Delta)^2$$

And expand into:

$$\sum_{i=1}^{x+1} c^2 + 2c(i-1)\Delta + ((i-1)\Delta)^2$$

We can break this into 3 summations, the first of which can be written as a product, because it's the summation of a constant:

$$(x+1)c^2 + \sum_{i=1}^{x+1} 2c(i-1)\Delta \quad + \sum_{i=1}^{x+1} ((i-1)\Delta)^2$$

Pull constants out of the summations:

$$(x+1)c^2 + 2c\Delta \sum_{i=1}^{x+1} (i-1) \quad + \Delta^2 \sum_{i=1}^{x+1} (i-1)^2$$

Shift the bounds of the summations. We can ignore i and $i^2$ when i = 0.

$$(x+1)c^2 + 2c\Delta \sum_{i=0}^{x} i \quad + \Delta^2 \sum_{i=0}^{x} i^2$$

Substitute identities:

$$(x+1)c^2 + 2c\Delta \frac{x(x+1)}{2} \quad + \quad \Delta^2 \frac{x(x+1)(2x+1)}{6}$$

Factor out (x + 1) and simplify:

$$(x+1)\left(c^2 + c\Delta x + \frac{1}{6}\Delta^2 x(2x+1)\right)$$

Which completes the derivation.

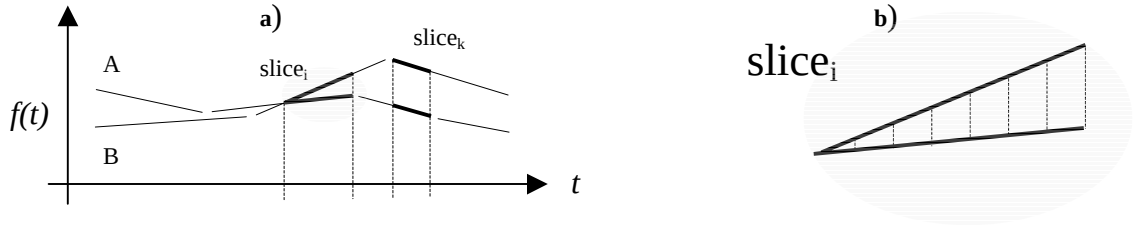



**Figure A: a)** Two examples of slices. **b)** A "zoom-in" of a $slice_i$ with "phantom" lines placed in-between the two segments using the same granularity as the original time series.