Author/s:
Qi, J;Zhang, R;Ramamohanarao, K;Wang, H;Wen, Z;Wu, D

Title:
Indexable online time series segmentation with error bound guarantee

Date:
2015-03-01

Citation:
Qi, J., Zhang, R., Ramamohanarao, K., Wang, H., Wen, Z. & Wu, D. (2015). Indexable online time series segmentation with error bound guarantee. World Wide Web, 18 (2), pp.359-401. https://doi.org/10.1007/s11280-013-0256-y.

Persistent Link:
https://hdl.handle.net/11343/54984

World Wide Web manuscript No.
(will be inserted by the editor)

Indexable Online Time Series Segmentation with Error Bound Guarantee

Jianzhong Qi · Rui Zhang · Kotagiri Ramamohanarao · Hongzhi Wang · Zeyi Wen · Dan Wu

Received: date / Accepted: date

Abstract The volume of time series stream data grows rapidly in various applications. To reduce the storage, transmission and processing costs of time series data, segmentation and approximation is a common approach. In this paper, we propose a novel online segmentation algorithm that approximates time series by a set of different types of candidate functions (polynomials of different orders, exponential functions, etc.) and adaptively chooses the most compact one as the pattern of the time series changes. We call this algorithm the Adaptive Approximation (AA) algorithm. The AA algorithm incrementally narrows the feasible coefficient spaces (FCS) of candidate functions in coefficient coordinate systems to make each segment as long as possible given an error bound on each data point. We propose an algorithm called the FCS algorithm for the incremental computation of the feasible coefficient spaces. We further propose a mapping based index for similarity searches on the approximated time series. Experimental results show that our AA algorithm generates more compact approximations of the time series with lower average errors than the state-of-the-art algorithm, and our indexing method processes similarity searches on the approximated time series efficiently.

Keywords Time Series · Approximation · Indexing · Similarity search

1 Introduction

A time series is a sequence of data points where each data point is associated with a timestamp. There are rapidly increasing research interests in the management of time series data due to its importance in a variety of applications such as Internet traffic management [17, 39],

J. Qi, R. Zhang, K. Ramamohanarao and Z. Wen
Department of Computing and Information Systems
University of Melbourne, Australia
Tel.: +61-3-83441332
Fax: +61-3-83441345
E-mail: {jianzhong.qi, rui.zhang}@unimelb.edu.au, rao@csse.unimelb.edu.au, zeyi.wen@unimelb.edu.au

H. Wang and D. Wu
School of Computer Science and Technology
Harbin Institute of Technology, China
E-mail: wangzh@hit.edu.cn, wudan.hit@gmail.com

transaction management [29,36], telecommunications [9], and finance [28]. These applications need to record the change of certain values (counts of network packets, temperature, foreign exchange rates, etc.) over time and the recorded values form time series that grow with high speed and continuously. For example:

- In telecommunication, AT&T's call-detail time series contains roughly 300 million calls per day generating approximately 7GBs data each day [13].
- In financial markets, the data vectors like Reuters transmit more than 275,000 prices per day for foreign exchange spot rates alone [10].

Due to the rapid and continuous growth of data in the above applications, we usually cannot afford to store the entire time series due to the huge volume [3]. This poses a new challenge in data storage, transmission, and processing. Therefore, the need for more compact representations of time series data is compelling. Another important characteristic of the above applications is that the data points in the time series may arrive (or be generated) continually. These applications require continuously monitoring the data and analyzing them in almost real time. Therefore, we need to process the data points and provide answers on the fly, i.e., the algorithms need to be "online".

A common approach to address the problem of the large data volume is *segmentation*, which provides more compact representations of time series by dividing time series data into segments and using a high level representation to approximate each segment. The highly compact segmentation can reduce both the space and the computational cost of storing and transmitting such data, and also reduce the workload of data processing (e.g., more efficient in mining the sequence) [38]. Therefore, in this paper, we try to find *a highly compact segmentation scheme that each segment can be approximated by a high level representation given an error bound on each data point and the amount of information (i.e., the number of parameters) used to represent the time series is minimized*. Furthermore, we require online algorithms in order to accommodate the continuous nature of the data generated in the applications described earlier.

Piecewise Linear Approximation (PLA) [24,27,35] has been one of the most widely used segmentation methods for many practical applications because of its simplicity. PLA divides a time series into segments and uses a linear function to approximate each segment. Nonetheless, linear functions may not always be the best choice to approximate a time series due to the different kinds of patterns of the time series. Therefore, *Piecewise Polynomial Approximation (PPA)* [15,26] is introduced to approximate the time series with polynomial patterns more properly. PPA uses polynomial functions instead of linear functions to approximate segments.

However, the goal of most current PLA and PPA methods [15,24,26,35] is to minimize the *holistic* approximation error (e.g., the Euclidean distance between the approximation and the original time series) given a certain amount of information (e.g., the number of segments [35] or the maximum error in a segment [24]), where the best approximation result is the one with the lowest holistic error. This goal is different from that of our work: minimizing the amount of information used to represent the time series given a certain error bound on each data point, where the best approximation result should be the one using the smallest amount of information. Therefore, these methods do not satisfy the requirement of our problem and cannot be used to solve our problem.

A recent study [27] that has the same problem setting as ours uses the *Feasible Space Window (FSW)* method to find the farthest segmenting point of each segment. Here, the *feasible space (FS)* is a space where any straight line can approximate all the data points read so far with a given error bound on each data point. As Fig. 1(a) shows, the area between the

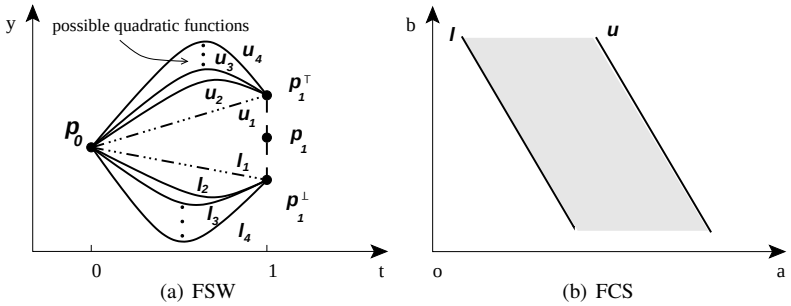


Fig. 1 FSW vs FCS

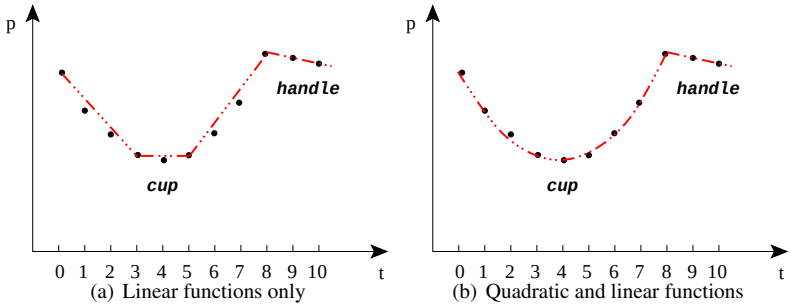


Fig. 2 Cup and handle in stock price time series

boundaries u_1 and l_1 is the feasible space for the approximation of p_0 and p_1. The feasible space is incrementally narrowed when new data points arrive continuously and eventually turns into an empty set at a certain data point so that the previous data point will be the farthest data point that can be approximated by a straight line. Thereby, FSW can make each approximation line as long as possible and minimize the amount of information used. Since FSW is the state-of-the-art for the problem addressed in this paper, we use FSW as the baseline in the experimental study and detail it in Section 3.2.

FSW approximates time series by linear functions only. However, in many real world situations, the patterns of the time series do not follow a constant rule. Using only one type of functions may not yield the best compaction. Take stock price time series as an example. A typical stock price pattern called *Cup and Handle* [33] has two parts: a "cup" and a "handle" as shown in Fig. 2. The cup is a round bottom and it is followed by the handle part which is a straight line. Fig. 2(a) shows how this time series is approximated using only linear functions and Fig. 2(b) shows how this time series is approximated using quadratic and linear functions. Based on an equation used to calculate the number of parameters (i.e., Equation 2 in Section 3.1), the segmentation in Fig. 2(a) uses nine parameters to represent the time series while the segmentation in Fig. 2(b) only uses six parameters. Therefore, using multiple types of functions can yield more compact approximation.

Motivated by these observations, we propose an online time series segmentation algorithm which approximates time series by a set of different types of functions (such as poly-

nomials of different orders, exponential functions, etc.) and adaptively chooses the most compact one as the pattern of the time series changes. We call this algorithm the *Adaptive Approximation (AA)* algorithm and refer to the functions used to approximate the time series as the *candidate functions*. To achieve our algorithm, we need to solve the following two subproblems: (i) for a candidate function, how to determine the values of its coefficients so that it can approximate as many data points as possible given an error bound on each data point; (ii) from a set of candidate functions, how to determine the one that generates the most compact approximation of a certain part of a time series (a sub-sequence).

Although feasible space is an intriguing idea for determining the compact approximation of time series data, it is difficult to apply the FSW algorithm to non-linear functions. The idea of FSW is to use a starting point and a next data point to determine the boundaries of the feasible space of the approximation on the fly. Nonetheless, most non-linear functions have more than two coefficients and their approximation boundaries can not be determined by only two data points. As shown in Fig. 1(a), two data points p_0 and p_1 can not uniquely determine the upper and lower boundaries of the feasible space for the quadratic function.

We propose a novel method to address this challenge: instead of finding the boundaries of feasible space as FSW does, we find the boundaries of the feasible values for the functions' coefficients – using two data points, we can uniquely identify such boundaries in the coefficient coordinate system. Then we can determine a space where each point is a feasible set of values for the coefficients of the candidate function. To distinguish our feasible space from the feasible space in FSW, we call our feasible space the *Feasible Coefficient Space (FCS)* and the algorithm used to generate FCS the *FCS algorithm*. Take a quadratic function $y = at^2 + bt + c$ as an example. A FCS (grey region in Fig. 1(b)) in a coefficient space using coefficients a and b as axes is determined through uniquely obtaining the upper boundary u by p_0 and $p_1^{\top}$ and the lower boundary l by p_0 and $p_1^{\perp}$. Note that the coefficient c in the function is required so that the function can approximate a time series that has a non-zero data point at time stamp 0, i.e., c is the time series' data point value when $t = 0$.

Based on the FCS algorithm, we propose an adaptive mechanism to determine the most compact candidate function for each part of a time series (a sub-sequence). Specifically, given a starting point, we continuously use the corresponding FCS algorithm of each candidate function to segment and approximate the time series until a data point (denoted by n_p) where each candidate function has obtained at least one segment. Then we compute the numbers of parameters used by the candidate functions to represent the sub-sequence ending at n_p and choose the candidate function with the smallest number of coefficients used as the approximation function of this sub-sequence.

We further study similarity searches on the approximated time series. Computing the similarity between two approximated time series requires "*decompressing*" (i.e., computing the approximation function value for every data point) the two time series, which is too costly considering the large amount of time series and data points. Indexing is a common way to reduce the search cost. However, existing indexing techniques [8, 22] treat every sub-sequence as a dimension and do not apply. This is because, after approximation by the AA algorithm, every time series has been segmented into a different set of sub-sequences. Any two sub-sequences from two time series may not be represented by the same type of function or with the same starting and ending points. There is no obvious way to define a consistent multi-dimensional space on all approximated time series based on the sub-sequences. To address this problem, we propose a mapping based indexing scheme. This scheme maps every approximated time series into a γ-dimensional space by computing the similarity between the approximated time series and γ *reference time series*. Here, γ is a predefined small integer and the reference time series are chosen in a way that guarantees

fast similarity computation between them and any approximated time series. We then index the similarity values with an R-tree [18] and use it to achieve efficient similarity searches on the approximated time series.

In summary, we make the following contributions in this paper.

- We propose an online time series segmentation algorithm called the *Adaptive Approximation (AA)* algorithm which approximates time series by a set of candidate functions (e.g., polynomials of different orders, exponential functions, etc.) and adaptively chooses the most compact one as the pattern of the time series changes.
- We propose a novel algorithm called the *Feasible Coefficient Space (FCS)* algorithm that can efficiently find the farthest segmenting data point for non-linear candidate functions with more than two coefficients (e.g., m^{th}-order polynomials where m is larger than 1). It addresses the drawback of the FSW algorithm, which can only find the farthest segmenting data point for the functions with two coefficients. We also analyze the complexities of the FCS algorithms for various candidate functions.
- We propose a mapping based indexing scheme to process similarity queries on the approximated time series. This indexing scheme overcomes the inefficiency of computing the similarity between two approximation time series by precomputing and indexing the similarity values between the approximated time series and a small set of *reference time series*, which are chosen to guarantee fast similarity computation between them and any approximated time series.
- We perform an extensive experimental study using both synthetic and real datasets. The results validate the effectiveness of our AA algorithm. It outperforms the state-of-the-art algorithm, FSW, in terms of the compression ratio. At the same time, the AA algorithm usually results in much lower actual errors than those introduced by the FSW algorithm given the same error bound. The results also show the high efficiency of our indexing scheme. It outperforms a scan based method in processing kNN queries by at least an order of magnitude.

This paper is an extended version of our earlier paper [41]. There we proposed the AA algorithm and the FCS algorithm for online time series segmentation and approximation. In this paper, we extend our work by investigating efficient similarity queries on the approximated time series. The challenge here is that, after approximation, every time series has been segmented into a different set of sub-sequences. Any two sub-sequences from two time series may not be represented by the same type of function or with the same starting and ending points. Existing indexes that treat every sub-sequence as a dimension to define a consistent multi-dimensional space on all approximated time series based on the sub-sequences do not apply. To address this challenge, we propose a mapping based indexing scheme that maps every approximated time series into a low dimensional space based on the similarity between the approximated time series and a small set of reference time series. We then index the similarity values with an R-tree to process similarity searches on the approximated time series efficiently. We propose a way to choose the reference time series that guarantees fast similarity computation between the reference time series and any approximated time series.

The rest of this paper is organized as follows. We first review related work in Section 2. Then we provide the preliminaries in Section 3. We present the FCS algorithms in Section 4 and the AA algorithm in Section 5. We study indexing the approximated time series for similarity searches in Section 6. In Section 7, we report the results of our experimental study. We conclude the paper and discuss future work in Section 8.

2 Related Work

2.1 Time Series Data Reduction

General lossless data compression techniques such as Huffman coding can yield data reduction, but they do not exploit the property that two consecutive values of a time series are close. Therefore, they cannot achieve compaction rates as high as those of segmentation methods, which are customized to the nature of time series.

Lossy data reduction methods such as *Discrete Fourier Transform (DFT)* [37] focus on the global patterns of time series instead of individual data point. In these methods, errors on individual data points vary widely (i.e., unbounded) and unpredictably. Some recent studies (e.g., [16]) provide error bounds on individual data points, but these methods have to know the whole time series and work on the data offline.

Linear segmentation is another widely used lossy data reduction method due to its simplicity. It approximate time series through *Piecewise Linear Approximation (PLA)* [24], which divides a time series into segments and uses a linear function to approximate each segment. Linear segmentation based methods can be categorized into two classes: offline segmentation and online segmentation. Offline segmentation methods, such as Top-down/Bottom-up algorithm [24] and evolutionary computation [14], need to obtain the whole time series before processing it. Online segmentation methods process each data point on the fly. Since online segmentation methods need to process data in almost real time and continuously, they have strong requirement on the efficiency of the algorithm.

An optimal solution for the linear segmentation is proposed by Bellman [4]: given a number of segment k and a time series whose length is n, an optimal PLA result that minimizes the holistic approximation error (i.e., the Euclidean distance between the approximation and the original time series) is found by dynamic programming with a cost of $O(kn^2)$. To obtain this optimal result on the fly, we need to continuously rerun the dynamic programming algorithm whenever a new data point arrives, which is very expensive for applications whose data volume is large. As a result, greedy methods [2, 24, 35] are proposed.

The *Sliding Window (SW)* algorithm [2] is a classic online segmentation algorithm. It uses the first data point of a time series as the starting data point of a segment and tries to put the next data point into this segment. The straight line that connects the current data point and the starting data point is used to approximate the current segment. Every time a new data point arrives, the approximation error needs to be calculated again based on the vertical deviation between all data points and the approximation line. Once the approximation error of the current segment exceeds a given error bound, the current segment ends (at the previous data point) and a new segment starts. The above process repeats until the end of the time series is reached. Subsequent studies have proposed some improvements to reduce the complexity of SW. For example, Keogh et al. [24] present a method called SWAB which combines the SW algorithm with a Bottom-Up mechanism. Palpanas et al. [35] report a technique to reduce the complexity of SWAB and SW to linear time.

The goal of the above-mentioned studies [2, 4, 24, 35] is to minimize the holistic approximation error (e.g., the Euclidean distance between the approximation and the original time series) given a certain amount of information (e.g., the number of segments). This goal is different from ours, i.e., minimizing the amount of information used to represent the time series given an error bound on each data point (i.e., the number of parameters used to represent the time series), where the best approximation result should be the one with the lowest amount of information used. Therefore, these methods do not satisfy the requirement of our problem and cannot be used to solve our problem.

Techniques	Abbreviation
Discrete Wavelet Transform [37]	DWT
Piecewise Aggregate Approximation [23]	PAA
Piecewise Linear Approximation [8]	PLA
Adaptive Piecewise Constant Approximation [22]	APCA
Chebyshev Polynomials [6]	CP
Euclidean distance [1,12]	
Dynamic Time Warping [5]	DTW
L_p norm [42]	
Longest Common Subsequence [40]	LCSS
Edit distance [7]	
Weighted sub-trajectories similarity [25]	
Match rewards and gap penalties [30]	

Table 1 Techniques used in time series similarity search

A recent work with the same problem setting as ours proposes an online PLA segmentation method named *Feasible Space Window (FSW)* [27], which uses the *Feasible space (FS)* to find the farthest segmenting point to make each segment as long as possible given an error bound on each data point. The FSW algorithm is the state-of-the-art for the problem addressed in this paper. Therefore, we use the FSW algorithm as the baseline in the experimental study and will detail it in Section 3.2.

Lemire [26] and Fuchs et al. [15] introduce two Piecewise Polynomial Approximation (*PPA*) methods to approximate time series data by polynomial functions. These two studies aim to minimize the total approximation error given a certain number of information, i.e., the model complexity and the set of polynomial function orders $\{0, 1, 2, ..., k\}$, respectively. In contrast, we aim to minimize the amount of information given a certain error bound on each data point. Moreover, our method is more generic in terms of candidate functions, i.e, besides polynomial functions, we also use other kinds of functions (e.g., exponential functions, etc.) as the candidate functions.

O'Rourke [34] proposes to fit straight lines between data ranges through transferring the problem into the coefficient space. There are two differences between this work and ours: First, this early work only solves the problem in the case of straight lines, which is the same as what FSW does. Fitting non-linear curves (especially, curves of high-order polynomials) into a sequence of data ranges is the major challenge addressed by our FCS algorithm. Second, the method of this work is also different from ours. Specifically, this work directly constructs the coefficient space and obtains the boundaries based on the given approximation function without any preprocessing such that its resultant coefficient space is always one dimension higher than ours and hence the computation cost is higher.

2.2 Time Series Similarity Search

Studies of similarity searches on time series data can be roughly classified into two groups: similarity search with different data reduction techniques and similarity search on different similarity metrics, as summarized in Table 1. Our study falls in the first group.

Similarity search with different data reduction techniques: Most of the data reduction techniques reviewed above have their corresponding studies for similarity searches. For example, DFT is used in similarity search by Rafiei and Mendelzon [37]. For piecewise approximation based data reduction techniques, Keogh et al. [23] propose *Piecewise Aggregate Approximation (PAA)* to achieve fast similarity searches. Chen et al. [8] propose a

Symbol	Meaning
δ	A given error bound on each data point
P	A time series
F	A set of candidate functions
p_i	The i^{th} data point in time series
p_{start}	The starting point
p_{next}	The next coming data point
p_{t_e}	The data point at which the FCS becomes empty
n_p	The number of parameters used to represent a sub-sequence
n_{tp}	The number of parameters used to represent a time series
n_s	The number of segments
n_c	The number of coefficients of a function
$f_j(t)$	The j^{th} approximation function
u	An upper boundary line
l	A lower boundary line
hp	A hyperplane
hf	A hyperface
hp_m	An m-dimensional hyperplane
hf_m	An m-dimensional hyperface
hp_u	An upper boundary hyperplane
hp_l	A lower boundary hyperplane

Table 2 Frequently used symbols

method to index time series approximated by PLA. In addition, *Adaptive Piecewise Constant Approximation (APCA)* [22] divides a time series into segments and represent each segment by its mean and length. CP [6] represents a time series using the coefficients of *Chebyshev polynomials*. These studies are based on time series approximation techniques that are different from ours and hence their similarity search approaches do not apply. In the wider area of similarity search on spatio-temporal data, recent studies [31,32,43,44] focus on indexing techniques for moving objects.

Similarity search on different similarity metrics: Early studies in the 1990s [1,12] used *Euclidean distance* as the similarity metric. Since then various similarity metrics have been studied. For example, Berndt and Clifford [5] studied the *Dynamic Time Warping (DTW)* metric; Yi and Faloutsos [42] studied L_p norm metric; Vlachos et al. [40] studied the distance based on *Longest Common Subsequence (LCSS)*; Chen and Ng [7] studied *Edit distance*. In more recent years, the similarity metrics studied have become more and more sophisticated. Lee et al. [25] proposed a weighted similarity metric on sub-trajectories that combined three types of distance, i.e., perpendicular distance, parallel distance, and angle distance. Moose and Patel [30] proposed a time series similarity evaluation framework that can include match rewards and gap penalties in similarity searches more freely. For a detailed comparison of the various similarity metrics, readers are referred to [11]. In this study we use Euclidean distance as the similarity metric because it is a basic similarity metric and works well in many problems [11].

3 Preliminaries

We first provide a formal definition of our problem in Section 3.1 and then explain the state-of-the-art algorithm for this problem, the Feasible Space Window algorithm (FSW), in Section 3.2. We summarize the symbols frequently used in the discussion in Table 2.

3.1 Problem Statement

Given a time series $P = (p_1, p_2, \ldots, p_n)$, an error bound δ and a set of candidate functions F, our problem is to divide P into k continuous segments $S_1, S_2, \ldots, S_k$:

$$S_1 = (p_1, p_2, \ldots, p_{c_1}),$$

$$S_2 = (p_{c_1}, p_{c_1+1}, \ldots, p_{c_2}),$$

$$\ldots$$

$$S_k = (p_{c_{k-1}}, p_{c_{k-1}+1}, \ldots, p_{c_k}),$$

where $c_i \in [1..n]$, $c_i < c_{i+1}$ and $c_k = n$. Here, the segments satisfy that (i) each segment S_j is approximated by a candidate function $f_j(t)$ in F with the error bound δ on each data point, formally,

$$\tilde{p}_i = \begin{cases} f_1(i) \ i = 1, \ldots, c_1, \\ f_2(i) \ i = c_1, \ldots, c_2, \\ \quad \ldots \\ f_k(i) \ i = c_{k-1}, \ldots, c_k, \end{cases} \tag{1}$$

satisfying

$$distance(\tilde{p}_i - p_i) \leq \delta;$$

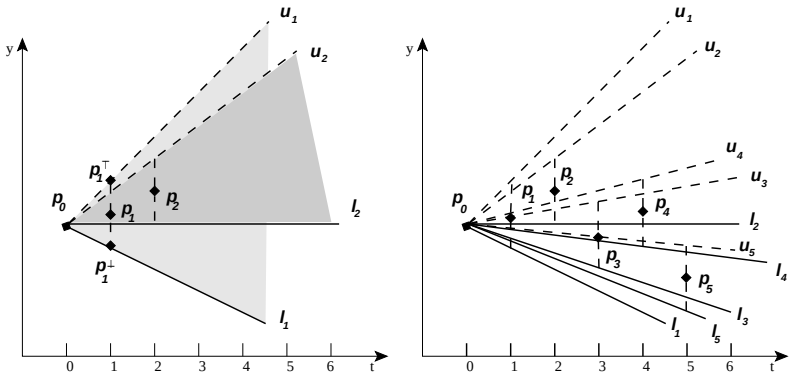
and (ii) the total number of parameters used to represent P (denoted as n_{tp}) is minimized.

Intuitively, in order to represent the approximation result of a time series, not only the values of coefficients of the approximation functions but also some other parameters, such as, the value of the starting point and the timestamp of each segmenting point, should be recorded as the approximation parameters of a time series. In this paper, in order to achieve smooth approximation, which is an important property desired in subsequent mining phases of the time series, we require the endpoint of the approximation function for the current segment to be the starting point of the approximation function for the next segment.

Thereby, the number of parameters needed to represent the first segment, which is approximated by a function with n_{c_1} coefficients, is $n_{c_1} + 2$, including $n_{c_1} - 1$ coefficient values, a value of the starting point (which is used to derived the $n_{c_1}^{th}$ coefficient value), a function type value, and a time timestamp of the first segmenting point. Since the following segments do not need the value of the starting point any more, each of them only needs $n_{c_i} + 1$ parameters. Formally,

$$n_{tp} = 1 + \sum_{i=1}^{k} (n_{c_i} + 1), \tag{2}$$

where n_{c_i} is the number of coefficients of the i^{th} approximation function.



(a) Feasible space (b) Determining a segment

Fig. 3 Example of FSW algorithm

3.2 Feasible Space Window

Liu et al. [27] propose the Feasible Space Window (FSW) algorithm to achieve compact segmentation by PLA. This algorithm finds the farthest segmenting point of each segment with an error bound guarantee on each data point through a concept called *Feasible Space* (FS). FS is an area in the data value space of a time series where any straight line in this area can approximate each data point of the corresponding segment within a given error bound.

Fig. 3(a) shows an example of the FS. Suppose the error bound is δ, and p_0 is the starting data point of a time series which is also required to be the starting point of the approximation line (i.e., the line that approximates the data points). When we read the second data point $p_1(t_1, y_1)$, we know that the y-coordinate of the approximation line at timestamp t_1 must be between the points $p_1^\top$ and $p_1^\perp$ which are the upper and lower boundary points of p_1. Here, $|\overline{p_1^\top, p_1}| = |\overline{p_1, p_1^\perp}| = \delta$. Therefore, any line between the upper line u_1 and the lower line l_1 satisfies the error bound requirement for p_1, and the region (the light grey region in the figure) between these two lines is the FS after reading the data point p_1.

The FS is incrementally updated when new data points are read. For example in Fig. 3(a), we read the next data point p_2 and similarly obtain two boundary lines u_2 and l_2. The area between u_2 and l_2 is the FS for p_2. The intersecting area of this FS and the previous FS (the dark grey region in the figure) becomes the current FS, which is the region for any approximation line that can satisfy the error bound requirement for both p_1 and p_2.

This FS update process repeats until the FS becomes empty at t_e, which means we cannot approximate any more following data points (including the current data point) by a straight line within the error bound. Hence, the previous data point p_{t_e-1} will be the endpoint of the current segment and also the new starting data point of the next segment.

Fig. 3(b) shows the process of determining a segment by the FSW algorithm. After p_2, we read the next two data points p_3 and p_4 and update the new FS to be $[u_3, l_2]$. After we read p_5, the new FS becomes empty because the lowest upper line u_5 is below the highest lower line l_2. Therefore, the previous data point p_4 is the segmenting point and the line connecting p_0 and p_4 is the approximation result of data points between p_0 and p_4. We use p_4 as the starting data point of the next segment and repeat this FS update process until the end of the time series is reached. As we can see, the FSW algorithm provides the linear approximation as each data point is read. Thus, it is an online algorithm.

4 Feasible Coefficient Space

Feasible space is an intriguing idea for determining a compact approximation of time series. However, it is difficult to apply the FSW algorithm to non-linear functions. The idea of FSW is to use a starting data point and a next data point to determine the boundaries of the feasible space. Nonetheless, most non-linear functions have more than two coefficients and their approximation boundaries cannot be determined by only two data points.

We propose the *FCS algorithm* to address this problem: instead of finding the boundaries of the feasible space, we find the boundaries of the feasible values for the functions' coefficients – using two data points, we can uniquely identify such boundaries in the coefficient coordinate system. Then we determine a space called the *Feasible Coefficient Space (FCS)* where each point is a feasible set of values for the coefficients of a candidate function.

Given a time series P, an error bound δ and a candidate function $f_j(t)$, the FCS algorithm approximates P as follows. When the first next data point p_{next} arrives, we derive two inequalities based on p_{start} (the starting data point), p_{next} and δ to determine two boundaries for the FCS of $f_j(t)$. Then we read another next data point to form two new boundaries and intersect them with the existing FCS to obtain a new FCS. The FCS is incrementally narrowed while the data points arrive continuously and finally becomes empty at a certain data point p_{t_e}, which means $f_j(t)$ cannot approximate any more following data points (including p_{t_e}) with a given error bound on each data point. Therefore, we take the previous data point p_{t_e-1} as the segmenting point of the current segment and start a new segment (also with p_{t_e-1}). The above process repeats until the time series is finished.

In this paper, we focus on the FCS algorithm for a few types of commonly used functions (polynomial functions of different orders, exponential functions), although our method can be extended to other types of functions (e.g., logarithmic functions) straightforwardly.

4.1 Second-order Polynomials

In this subsection, we present the FCS algorithm for the second-order polynomial function (i.e., the quadratic function). A second-order polynomial function is in the form of Equation (3) where a, b and c are coefficients of the function:

$$y = at^2 + bt + c. \tag{3}$$

According to our problem definition, the first data point $p_0(t_0, y_0)$ of the time series must be on the approximation curve. Hence,

$$y_0 = at_0^2 + bt_0 + c. \tag{4}$$

When the second data point $p_1(t_1, y_1)$ arrives, if we approximate this data point by the quadratic function, then the approximation value of y_1 on the curve is

$$\tilde{y}_1 = at_1^2 + bt_1 + c. \tag{5}$$

Combining Equations (4) and (5), we have

$$\tilde{y}_1 = y_0 + a(t_1^2 - t_0^2) + b(t_1 - t_0). \tag{6}$$

Since we require that the approximation error of each data point does not exceed a predefined error bound δ, $\tilde{y}_1$ must fall in the interval $[y_1 - \delta, y_1 + \delta]$. Therefore, we have:

$$y_0 + a(t_1^2 - t_0^2) + b(t_1 - t_0) \leq y_1 + \delta; \tag{7}$$

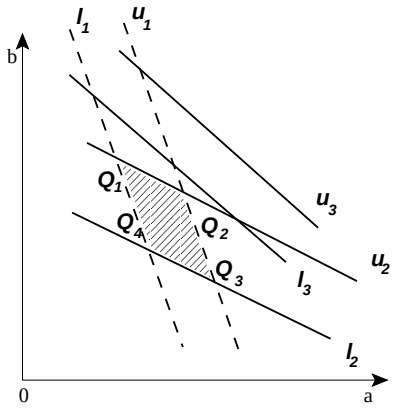


Fig. 4 Feasible coefficient space for quadratic functions

$$y_0 + a(t_1^2 - t_0^2) + b(t_1 - t_0) \geq y_1 - \delta. \tag{8}$$

Using the above inequalities, we can construct a 2-dimensional FCS in the coefficient coordinate system of the quadratic function with axes a and b (c is omitted when we combine Equations (4) and (5) and can be obtained by $c = y_0 - at_0^2 - bt_0$ based on a and b.).

As shown in Fig. 4, Inequalities (7) and (8) describe two parallel lines l_1 and u_1 indicating the lower and upper boundaries, between which is the current FCS, where each point is a feasible set of the coefficients' values of the quadratic function. When we read a new data point $p_2(t_2, y_2)$, we try to incorporate it in the current segment through similarly obtaining two other parallel lines l_2 and u_2, between which is the FCS satisfying the error bound requirement for p_0 and p_2. If l_2 and u_2 are not parallel to the previous lines l_1 and u_1, the intersection of the previous FCS (area between l_1 and u_1) and the current FCS (area between l_2 and u_2) is the new FCS (the shaded polygon area in Fig. 4).

If this new FCS is not empty, we continue to obtain the boundaries l_3 and u_3 for another point $p_3(t_3, y_3)$ and narrow the FCS (a polygon) by these two lines. This process is repeated until the FCS becomes empty at t_e when we cannot incorporate the new data point p_{t_e} in the current segment and approximate this segment by a quadratic function with the given error bound on each data point. At this moment, we use the previous data point p_{t_e-1} as the endpoint of the current segment as well as the starting point of the next segment, and then continue the above process.

Algorithm: The FCS algorithm for quadratic functions is named *FCSP2* and the pseudo-code is shown in Algorithm 1, where the current FCS (a convex polygon), the starting data point, the next data point and the error bound are denoted by g, p_{start}, p_{next}, and δ, respectively. Whenever a new p_{next} arrives, Inequalities (7) and (8) define two lines u and l, which are the upper and lower boundaries of the FCS for p_{next}, respectively. To obtain the intersection of this FCS and g, for each edge of g, we calculate the intersecting points with l. Since g is convex, at most two edges of g intersect with l and g is divided into two parts by l. We remove the lower part of g and the other part is the new g (*the lower and upper parts will be defined in Section 4.2*). Similarly, we use u to divide the newly computed g and cut the upper part. Note that g is still a convex after it is cut by u and l, and the resultant polygon g' is the new FCS after processing the data point p_{next}.

Complexity Analysis: In algorithm FCSP2, the most frequently executed operation is the computation of intersecting points between two generated boundary lines (l and u) and

Algorithm 1: FCSP2(g, p_{start}, p_{next}, δ)

```
// g : the current polygon; p_start : the starting data point;
// p_next : the next data point; δ : the error bound;
Construct two lines l and u by p_start and p_next according to Inequalities (7) and (8);
if g is empty then
    g' ← the space between l and u;
else
    I ← ∅;
    for each edge e_i of g do
        I ← I∪ the intersection points between e_i and l;
    if I ≠ ∅ or g is lower than l then
        Cut off the part lower than l from g based on I;
    I ← ∅;
    for each edge e_j of g do
        I ← I∪ the intersection points between e_j and u;
    if I ≠ ∅ or g is higher than u then
        Cut off the part higher than u from g based on I;
    g' ← the remained part of g;
Return g'
```

the current FCS, which takes constant time. Suppose we have obtained $n + 1$ data points before we get p_{next}, which means we have already generated n pairs of lines. In the worst case, these lines could make up a polygon with $2n$ edges. The number of intersection computation is $2n$ for either l or u and $4n$ in all. Therefore, the worst case computational cost of the FCSP2 for one particular data point is

$$C_2 = 4n \in O(n).$$

Further, based on the proof in [34], the amortized complexity of FCSP2 per data point is only $O(1)$.

4.2 M^{th}-order Polynomials

In this subsection, we present the FCS algorithm of m^{th}-order polynomials where $m \geq 3$. Firstly, we consider the case of $m = 3$. The 3^{rd}-order polynomial function is also called the cubic function which is in the form of Equation (9) where a, b, c and d are the coefficients.

$$y = at^3 + bt^2 + ct + d \tag{9}$$

Similar to the case of quadratic functions, we use the starting data point $p_0(t_0, y_0)$ and the approximate value of the following data point $p_1(t_1, y_1)$ to obtain a pair of equations:

$$y_0 = at_0^3 + bt_0^2 + ct_0 + d, \tag{10}$$

$$\tilde{y}_1 = at_1^3 + bt_1^2 + ct_1 + d. \tag{11}$$

Combining Equations (10) and (11), we have

$$\tilde{y}_1 = y_0 + a(t_1^3 - t_0^3) + b(t_1^2 - t_0^2) + c(t_1 - t_0). \tag{12}$$

According to our problem definition, the approximate value $\tilde{y}_1$ should fall into the interval $[y_1 - \delta, y_1 + \delta]$. Thus, we have

$$y_0 + a(t_1^3 - t_0^3) + b(t_1^2 - t_0^2) + c(t_1 - t_0) \leq y_1 + \delta, \tag{13}$$

$$y_0 + a(t_1^3 - t_0^3) + b(t_1^2 - t_0^2) + c(t_1 - t_0) \geq y_1 - \delta. \tag{14}$$

In the 3^{rd}-order coefficient coordinate system with axes a, b and c, Inequalities (13) and (14) describe two bounding planes and the space between these bounding planes is a 3-dimensional FCS. When we obtain a third data point $p_2(t_2, y_2)$, we use it to derive a new pair of bounding planes and use these planes to cut the previous FCS and generate a new FCS. In this case, the generated FCS is a 3-dimensional polyhedron. This process is repeated for the following data points to incrementally update the FCS until the FCS becomes empty at time t_e. We then segment the time series at the previous data point p_{t_e-1} and start a new round of approximation (also at p_{t_e-1}).

We generalize the quadratic and cubic polynomials to the case of m^{th}-order polynomials, which are in the following form:

$$y = a_m t^m + a_{m-1} t^{m-1} + ... + a_1 t + a_0. \tag{15}$$

Given the starting data point and the current data point, we can construct two $(m-1)$-dimensional hyperplanes in the m^{th}-order coefficient coordinate system as the boundaries of the m-dimensional FCS through the following inequalities:

$$\begin{aligned} \tilde{y}_1 = y_0 + a_m(t_1^m - t_0^m) + a_{m-1}(t_1^{m-1} - t_0^{m-1}) \\ +... + a_1(t_1 - t_0) \geq y_1 - \delta, \end{aligned} \tag{16}$$

$$\begin{aligned} \tilde{y}_1 = y_0 + a_m(t_1^m - t_0^m) + a_{m-1}(t_1^{m-1} - t_0^{m-1}) \\ +... + a_1(t_1 - t_0) \leq y_1 + \delta. \end{aligned} \tag{17}$$

The hyperhedron between these two boundaries is an m-dimensional FCS. When a new data point is read, we generate a new pair of $(m-1)$-dimensional hyperplanes and use them to cut the current m-dimensional hyperhedron (the current FCS) and obtain the intersecting hyperhedron as the new FCS.

Algorithm: When $m > 3$, the FCS of m^{th}-order polynomials will become a high-dimensional hyperhedron. We present an algorithm to generate this high-dimensional FCS (denoted as *FCSPm*) in Algorithm 2. In this algorithm, we use h, h', $hyperfaces$ and $hyperplanes$ to denote the current FCS, the new FCS, the faces of hyperhedron and high-dimensional planes generated by inequalities, respectively. The starting data point, the next data point and the error bound are defined and denoted similarly to their counterparts in Algorithm 1. When the new data point arrives, Inequalities (16) and (17) determine two hyperplanes denoted as hp_l and hp_u which are used to update the current FCS. The update process is similar to that of the FCSP2 algorithm.

To cut a m-dimensional FCS, we need determine a part of $(m-1)$-dimensional hyperface is the lower or upper part regarding to a $(m-1)$-dimensional hyperplane. For example, in the case of quadratic functions ($m = 2$), we need to determine the relative lower and upper parts of an edge (1-dimensional hyperface) regarding a line (1-dimensional hyperplane). Given a $(m-1)$-dimensional hyperplane in a m-dimensional coefficient coordinate system

Algorithm 2: FCSPm(h, p_{start}, p_{next}, δ)

```
// h : the current (m-dimensional) hyperhedron, p_start : the starting data point;
// p_next : the next data point; δ : the error bound;
Construct two (m − 1)-dimensional hyperplanes hp_l and hp_u by p_start and p_next according to
  Inequalities (16) and (17);
if h is empty then
    h' ← the space between hp_l and hp_u;
else
    I ← ∅;
    for each (m − 1)-dimensional hyperface hp of h do
        I ← I∪ the (m − 2)-dimensional intersection hyperface between hp_l and hp;
    if I ≠ ∅ or h is lower than hp_l then
        Cut off the part lower than hp_l from h based on I;
    I ← ∅;
    for each (m − 1)-dimensional hyperface hp of h do
        I ← I∪ the (m − 2)-dimensional intersection hyperface between hp_u and hp;
    if I ≠ ∅ or h is higher than hp_u then
        Cut off the part higher than hp_u from h based on I;
    h' ← the remained part of h;
Return h'
```

cutting the whole space into two parts, we define that a part of a $(m-1)$-dimensional hyperface is a lower (or upper) part regarding to this $(m-1)$-dimensional hyperplane if this part of the hyperface is contained in the lower (or upper) part of the space regarding the same $(m-1)$-dimensional hyperplane. The lower and upper part of the space is defined according to a selected coefficient axis called *pilot axis*: if a_m is the pilot axis, we define the upper part of the space to be the part containing $+\infty$ along the a_m axis, and the other part of this space is the lower part of space. For example, in Fig. 4, choosing axis b as the pilot axis, the edge Q_1Q_4 is the upper part regarding the line l_2 because this edge is contained in the upper part of the space.

Complexity Analysis: For the case of m^{th}-order polynomials, suppose we have obtained $n+1$ data points before we get p_{next} and constructed n pairs of $(m-1)$-dimensional hyperplanes. In the worst case, these $(m-1)$-dimensional hyperplanes result in a m-dimensional hyperhedron with $2n$ $(m-1)$-dimensional hyperfaces. Each $(m-1)$-dimensional hyperface is described by $2(n-1)$ $(m-2)$-dimensional hyperfaces and, similarly, each $(m-2)$-dimensional hyperfaces can be described by $2(n-2)$ $(m-3)$-dimensional hyperfaces if $m > 3$. This process keeps on going until it reaches the 1-dimensional hyperfaces (i.e., lines).

Calculating the intersection of two $(m-1)$-dimensional hyperplanes is denoted as $Cost_{m-1}$, which takes constant time. In FCSPm, the processes of using the lower $(m-1)$-dimensional hyperplane ($hp_{l_{m-1}}$) to cut one of the $(m-1)$-dimensional hyperfaces (hf_{m-1}) of h is as follows: Firstly, we calculate the intersection of $hp_{l_{m-1}}$ and hf_{m-1} resulting a $(m-2)$-dimensional hyperplane denoted by hp_{m-2} and the cost is suppose to be constant denoted as $Cost_{m-1}$. Then, for each $(m-2)$-dimensional hyperface hf_{m-2} describing the previous $(m-1)$-dimensional hyperface hf_{m-1}, we calculate the intersection of it and hp_{m-2}, which is a $(m-3)$-dimensional hyperplane. We further use this $(m-3)$-dimensional hyperplane to cut $(m-3)$-dimensional hyperfaces describing the hf_{m-2} and get a $(m-4)$-dimensional hyperfaces. This process is executed iteratively until it reaches the calculation

of the intersection of two lines and we have the total cost (denoted as C_m) as follows:

$$\begin{aligned} C_m &= 2nCost_{m-1} + 2n * 2(n-1)Cost_{m-2} + \\ &\dots \\ &+2n * 2(n-1) * ... * 2(n-(m-2))Cost_1, \end{aligned}$$

where $Cost_1$, $Cost_2$, ... and $Cost_{m-1}$ are different constant values. When $n >> m$, C_m is dominated by the last term, which has the worst case complexity of $O(n^{m-1})$. Although the complexity is polynomial, when m is large, the computation and implementation cost will also become very large. Therefore, we do not use polynomials with orders higher than two if it is not essential.

4.3 Exponential Functions

In this subsection we present the FCS algorithm for exponential functions:

$$y = be^{at}.$$

It can be transformed into the linear approximation function using logarithmic rules. When $b > 0$, we take the natural logarithms of both sides of the equation and get

$$\ln(y) = \ln(b) + at. \tag{18}$$

Similarly, the starting data point $p_0(t_0, y_0)$ should be on the curve such that

$$y_0 = be^{at_0}, \tag{19}$$

namely,

$$\ln(y_0) = \ln(b) + at_0 \ (y > 0, b > 0). \tag{20}$$

When we get another data point $p_1(t_1, y_1)$, for the approximate value of y_1, we have

$$\ln(\tilde{y}_1) = \ln(b) + at_1 \ (\tilde{y}_1 > 0, b > 0). \tag{21}$$

Combining Equations (20) and (21), we obtain

$$\ln(\tilde{y}_1) = \ln(y_0) + a(t_1 - t_0). \tag{22}$$

We know that $\tilde{y}_1$ is within the error bound, i.e., $\tilde{y}_1 \in [y_1 - \delta, y_1 + \delta]$. Here we assume $y_1 - \delta > 0$ (if not, we iteratively multiply δ by 0.5 until it stands). Note that the natural logarithmic function is a monotonic increasing function with $(0, +\infty)$ as the definitional domain. Thus, we combine the previous two inequalities and obtain

$$\ln(y_0) + a(t_1 - t_0) \leq \ln(y_1 + \delta) \ (y_0, y_1 > 0), \tag{23}$$

$$\ln(y_0) + a(t_1 - t_0) \geq \ln(y_1 - \delta) \ (y_0, y_1 - \delta > 0). \tag{24}$$

Then we have

$$\frac{\ln(y_1 - \delta) - \ln(y_0)}{t_1 - t_0} \leq a \leq \frac{\ln(y_1 + \delta) - \ln(y_0)}{t_1 - t_0}. \tag{25}$$

This inequality defines a pair of boundary points for the coefficient a. Every time a new data point arrives, we obtain a new pair of such boundary points and incrementally update the feasible value range (a 1-dimensional FCS) for coefficient a until the FCS becomes empty. This is similar to the linear segmenting problem. The only difference is that, in the linear segmenting case, Inequality (25) becomes

$$\frac{y_1 - y_0 - \delta}{t_1 - t_0} \leq a \leq \frac{y_1 - y_0 + \delta}{t_1 - t_0}. \tag{26}$$

Therefore, the FCS algorithm used for the exponential function is the same as that for the linear function. Furthermore, the aforementioned FCS algorithm FCSPm is applicable for both the linear function and the exponential function because the linear function is a 1^{st}-order polynomial whose high-dimensional coefficients are zero.

Complexity Analysis: In the complexity aspect, the exponential function is also similar to the linear function. No matter how many data points have been processed, we only need to maintain two points to indicate the current feasible value range of coefficient a (1-dimensional FCS). Therefore, when the next data point arrives, we only need to compare the values of two pairs of boundary points to obtain the new FCS. Thus, the computation complexity of exponential function is $C_e = C_1 = O(1)$

5 Adaptive Approximation Algorithm

The objective of our method is to minimize the number of parameters used to represent the time series given an error bound on each data point. Suppose there is a given starting data point p_{start} and an appropriate FCS algorithm for each candidate function. The AA algorithm approximates a time series as follows: we read the data points one by one and approximate these points through the respective FCS algorithms of the candidate functions. For each candidate function, the corresponding FCS is incrementally narrowed while the data points arrive continuously and turns into empty at t_e, so we take p_{t_e-1} as the segmenting point of the current segment and the new starting data point of the next segment, and then we continue to read and approximate the following data points. This process is repeated by all the candidate functions until each candidate function has encountered at least one segmenting point. At this moment, for each candidate function, we compute the number of parameters used for approximating the current sub-sequence (denoted as n_p), then compare and choose the function with the smallest n_p as the approximation function of the current sub-sequence. We repeat the above process until the whole time series is processed.

Similar to Equation (2), the number of parameters used to represent a sub-sequence approximated by a candidate function $f_j(t)$ can be obtained by $n_p = n_{c_j} * n_{s_j}$, where n_{c_j} is the number of coefficients of $f_j(t)$ and n_{s_j} is the number of segments generated and approximated by $f_j(t)$. However, when we compute n_p at the data point p_t where each candidate function has encountered at least one segmenting point, p_{t-1} must be the segmenting point for some candidate functions but it may not be the segmenting point for other candidate functions. The latter type of functions can approximate more data points without increasing n_p. For fair comparison, we heuristically tune n_p for these functions to be $n_p = n_c * (n_s - 1) + n_c * \alpha$, where α ranges from 0 to 1.

We call the segments obtained by each candidate function the *local segments* of the corresponding function and the segments obtained by the final chosen approximation function *global segments* of the corresponding sub-sequence. Thereby, each data point belongs

to many different local segments (generated by different candidate functions) but only one global segment (generated by the chosen approximation function).

The algorithm complexity of our method depends on the candidate function with the highest complexity. For example, if we choose the quadratic function, the linear function and the exponential function as the candidate functions, then the complexity is dominated by the quadratic function and the worst case complexity and the amortized complexity of the AA algorithm will be $O(n)$ and $O(1)$, respectively. We provide the pseudo-code of the AA algorithm in Algorithm 3 and detail it in the following subsections.

5.1 Initialization

As Algorithm 3 shows, we use lists to store the function coefficients used to represent the time series: the function coefficients of the global segments of the time series and those of the local segments are stored in lists l_g and l_{l_j}, respectively, where j indicates that this list is used to store the coefficients of the local segments generated by the j^{th} candidate function (line 2). At start, we initialize the AA algorithm with the first data point p_0 of the time series P to be the first data point of the first global segment of P (denoted by p_{first}) and also the starting data point of the first local segment of each candidate function (denoted by p_{start_j}) (line 3). We set every list and the feasible coefficient space of every candidate function (denoted by FCS_j) to be empty (line 4). We also set two flags F_{rv} and F_{cf} to be false (line 5). Here, F_{cf} indicates whether there has been a candidate function chosen as the approximation function and F_{rv} indicates whether the chosen approximation function is no longer feasible for further approximation.

5.2 Finding the Segmenting Point

When the next data point p_{next} comes (lines 6 and 7), we firstly need to check the value of flag F_{cf}. If F_{cf} is false, it means there is no chosen approximation function (line 8). Hence, we iteratively update the FCS of each candidate function to choose an approximation function (lines 9 to 24). Otherwise, we simply use the chosen function to approximate the coming data point and only update its FCS (lines 25 to 32). Therefore, the process of finding the segmenting data point can be divided into two cases as follows:

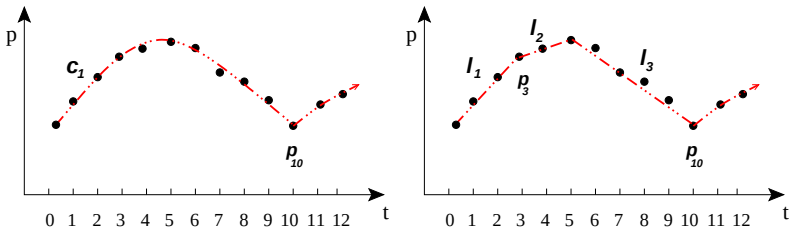
Case One: If F_{cf} is false, for each candidate function $f_j(t)$ in F, we firstly update its feasible coefficient space FCS_j through the corresponding FCS algorithm $FCSA_j$ (lines 10 and 11). If the FCS becomes empty after the update, then the previous data point p_{next-1} is the segmenting point of $f_j(t)$. Thus, we save the approximation parameters of this generated local segment into the local segment list l_{l_j}, update the starting data point of this function p_{start_j} to be the previous data point p_{next-1}, and then re-construct the FCS for p_{next} based on the new starting data point through re-invoking the corresponding FCS algorithm $FCSA_j$ (lines 12 to 15).

After we finish the FCS update of each candidate function, we further check whether all local segment lists are non-empty (line 16). If so, it means each candidate function has met at least one segmenting point and we need to choose an approximation function for the sub-sequence from p_{first} to p_{next-1}. we compute the number of parameters used by each candidate function to represent the sub-sequence (denoted as n_p) and choose the function with the smallest n_p as the approximation function $f_a(t)$ (lines 17 to 19).

Algorithm 3: AA(P, δ, F)

```
1  // P = (p_0, p_1, ...p_n, ...); δ : error bound; F = {f_1(t), f_2(t), ... f_j(t), ... f_m(t)};
2  l_g = l_{l_1} = l_{l_2} = ... = l_{l_m} = ∅; // coefficient lists of the approximation functions
3  p_first = p_start1 = p_start2 = ... = p_startm = p_0; // starting points of the approximation
   functions
4  FCS_1 = FCS_2 = ... = FCS_m = ∅; // feasible coefficient spaces of the approximation functions
5  F_rv = F_cf = false; // re-initialization flags
6  while P not completed do
7      Fetch the next data point p_next from P;
8      if F_cf = false then
9          // have not chosen the approximation function;
10         for each candidate function f_j(t) in F do
11             FCSA_j (f_j(t), FCS_j, p_start_j, p_next);
12             if FCS_j = ∅ then
13                 Append new local segment to l_{l_j};
14                 p_start_j ← p_{next−1};
15                 FCSA_j (f_j(t), FCS_j, p_start_j, p_next);
16         if all l_{l_j} ≠ ∅ then
17             for each f_j(t) in l_f do
18                 Calculate n_p for points between p_first and p_{next−1};
19             Choose the function with the smallest n_p as f_a(t);
20             if p_start_a ≠ p_{next−1} then
21                 F_cf ← true;
22             else
23                 F_rv ← true;
24                 Append l_{l_a} to l_g;
25     else
26         // have chosen the approximation function;
27         FCSA_a (f_a(t), FCS_a, p_start_a, p_next);
28         if FCS_a = ∅ then
29             Append new local segment to l_{l_a};
30             Append l_{l_a} to l_g;
31             F_cf ← false;
32             F_rv ← true;
33     if F_rv = true then
34         // re-initialization;
35         p_first ← p_{next−1};
36         for j from 1 to m do
37             l_{l_j} ← ∅;
38             p_start_j ← p_{next−1};
39             FCSA_j (f_j(t), FCS_j, p_start_j, p_next);
40         F_rv ← false;
41 Return l_g;
```

Then, we check whether the previous data point p_{next-1} is the segmenting point of $f_a(t)$. If not, we will use this function to approximate more subsequent data points until its FCS becomes empty, i.e., we change the value of F_{cf} to be true (lines 20 and 21). Otherwise, we directly append the approximation parameters in its local segment list l_{l_a} to the global segment list l_g and change F_{rv} to be true to do re-initialization (lines 22 to 24).



(a) Quadratic functions (b) Linear functions

Fig. 5 A running example

Case Two: If F_{cf} is true, then we only need to update the FCS of the chosen approximation function for the coming data point p_{next} and check whether its FCS becomes empty after the update (lines 25 to 28). If not, we continue to process the next data point. Otherwise, it indicates that the chosen approximation function has met a new segmenting point p_{next-1}. Hence, we add the approximation parameters of this newly generated segment to the local segment list of this function (denoted by l_{l_a}) and append the information in this list to the global segment list l_g (lines 29 and 30). Finally, we set F_{cf} to be false and F_{rv} to be true, respectively (lines 31 and 32).

5.3 Re-initialization

If F_{rv} becomes true after the processes as described in Section 5.2, the re-initialization step is invoked (lines 33 to 34): (i) we use the previous data point p_{next-1} as the starting data point of the next global segment p_{first} and the next local segment for each candidate function (line 35); (ii) for each candidate function, we set the corresponding local segment list to be empty and re-construct the FCS for p_{next} based on the new starting data point (lines 36 to 39).

We repeat the above process until the whole time series is completed. Given the same approximate error bound, the AA algorithm can achieve better extent of compactness through always finding the most compact candidate function, which has the smallest n_p, as the approximation function. As we can see, the approximation process only relies on the coming data point. Therefore, it is an online segmentation algorithm.

5.4 A Running Example

We illustrate the AA algorithm with a running example. Fig. 5 depicts the cases of approximating a time series by two candidate functions: the quadratic function and the linear function. Starting from the data point p_0, both functions incrementally read and approximate the coming data points one by one through their FCS algorithms. When p_4 comes, the FCS of the linear function becomes empty so p_3 is the segmenting point of the linear function. However, the FCS of the quadratic function is not empty at p_4. Thus, we only start a new local segment for the linear function. This means, we save p_3 in the local segment list for the linear function, change the starting data point to be p_3, re-construct the FCS based on p_4 and p_3, and continue to approximate the next data point p_5. The approximation process is repeated by both functions until p_{11} where the FCS of the quadratic function becomes empty. At this moment, both functions have had at least one segmenting point. Then we compute

the n_p (the number of parameters used to represent the sub-sequence) of both functions and obtain $n_p = 3$ for the quadratic function and $n_p = 6$ for the linear function. Since the quadratic function has a smaller n_p value, it is chosen as the approximation function for the sub-sequence between p_0 and p_{10}. For both candidate functions, the approximation restarts at p_{10} and the same process repeats until the end of the time series.

6 Similarity Search on Approximated Time Series

Given a set of n time series $\tilde{\mathcal{P}} = \{\tilde{P}_1, \tilde{P}_2, ..., \tilde{P}_n\}$ and a query time series $\tilde{Q}$, each of which has been approximated by the AA algorithm, we study two basic types of similarity queries on $\tilde{\mathcal{P}}$, i.e., range queries and k nearest neighbour (kNN) queries.

Definition 1 (Range Query on Approximated Time Series) *Given a query time series $\tilde{Q}$ and a real number as the query range r, the range query $rq(\tilde{Q}, r)$ on approximated time series $\tilde{\mathcal{P}}$ returns every approximated time series whose distance to $\tilde{Q}$ is less than or equal to r. Formally,*

$$rq(\tilde{Q}, r) = \{\tilde{P} \in \tilde{\mathcal{P}} | dist(\tilde{P}, \tilde{Q}) \leq r\}.$$

Definition 2 (kNN Query on Approximated Time Series) *Given a query time series $\tilde{Q}$ and a query parameter k, the kNN query $knn(\tilde{Q}, k)$ on approximated time series $\tilde{\mathcal{P}}$ returns a subset $\tilde{\mathcal{P}}'$ of $\tilde{\mathcal{P}}$ with k approximated time series, where each time series has a distance to $\tilde{Q}$ that is less than or equal to any time series in $\tilde{\mathcal{P}} \setminus \tilde{\mathcal{P}}'$. Formally,*

$$knn(\tilde{Q}, k) = \{\tilde{P} \in \tilde{\mathcal{P}}' | |\tilde{\mathcal{P}}'| = k, \forall \tilde{P}' \in \tilde{\mathcal{P}} \setminus \tilde{\mathcal{P}}', dist(\tilde{P}, \tilde{Q}) \leq dist(\tilde{P}', \tilde{Q})\}.$$

In the definitions, $\tilde{P} = (< f_1, t_{s1} >, < f_2, t_{s2} >, ..., < f_i, t_{si} >, ..., < f_d, t_{sd} >)$ denotes an approximated time series generated by AA from a time series $P = (p_1, p_2, ..., p_N)$, where f_i and t_{si} denote the approximation function and starting timestamp of the i^{th} sub-sequence of P. Function $dist(\tilde{P}, \tilde{Q})$ returns the distance between two approximated time series. In this study we use Euclidean distance since it is a basic similarity metric and it works well in many problems [11]. For simplicity we assume that the approximated time series are generated from time series of the same number of data points N, although our analysis and proposed methods also apply on time series of different lengths.

A *scan based solution* to the queries is to first *"decompress"* every approximated time series (i.e., compute a value for each data point from the approximation functions), and then compute its distance to the query time series by comparing each pair of data points of the two time series. By this method processing a rang query on a set of n approximated time series takes $O(nN)$ time, and processing a kNN query takes $O(nN \log k)$ time, where "$\log k$" is for maintaining a heap for the kNN found so far. Considering the large amount of time series data being generated continuously, the scan based method is too slow.

To improve query efficiency, indexing is a natural choice. In previous studies [8, 22], a commonly used approach is to view the original time series as points in an N-dimensional space, and the approximated time series as points in a βd-dimensional space, where d denotes the number of sub-sequences in an approximated time series and β denotes the number of parameters used to represent a sub-sequence. Then effectively, the approximation process becomes a mapping process that maps an m-dimensional point to a relatively low βd-dimensional point, and a multi-dimensional index is used to index the approximated time series in the βd-dimensional space. Note that this method requires each original time series to be segmented into the same number (d) of equi-length sub-sequences, and each sub-sequence to be represented by the same number (β) of parameters.

To apply the above approach, an inherent difficulty is that our AA algorithm segments different time series into different sets of sub-sequences. Any two sub-sequences from two time series may not be represented by the same type of function or with the same starting and ending points. There is no obvious way to define a consistent βd-dimensional space for all time series approximated by the AA algorithm.

To overcome this difficulty, we propose an alternative mapping approach as follows. This approach involves a small set of γ predefined *reference time series*. It uses the distances of an approximated time series $\tilde{P}$ to the reference time series as the coordinates of the approximated time series in a γ-dimensional space. As a result, we map the approximated time series into a γ-dimensional space and can index them with a traditional multi-dimensional index (we use an R-tree due to its popularity). When a query is issued, we first map the query to the γ-dimensional space, and then perform a search on the multi-dimensional index to find the query result. This way we can take advantage of the pruning capability of the multi-dimensional index and improve query processing efficiently.

Next we elaborate how we choose the reference time series, map the approximated time series, and process queries using the index.

6.1 Choosing the Reference Time Series

When choosing the reference time series there are two main goals to achieve:

- The distance from the reference time series to any approximated time series should be computed efficiently, since the distance will be computed frequently for a large number of approximated time series.
- The distance between the reference time series themselves should be large enough so that approximated time series that are close to different reference time series can be differentiated by their distances to different reference time series.

First we consider the distance computation efficiency. If a random time series is chosen, then for distance computation we will have to decompress every approximated time series, which takes too much time. To avoid the decompression, we propose to use a time series with a constant value at each data point. Let $R = (v, v, ..., v)$ be such a time series, where v is a constant value. Then for an approximated time series $\tilde{P}$,

$$\tilde{P} = (< f_1, t_{s1} >, < f_2, t_{s2} >, ..., < f_i, t_{si} >, ..., < f_d, t_{sd} >),$$

we can define a segmentation $\tilde{R}$ for R that matches $\tilde{P}$:

$$\tilde{R} = (< f'_1, t_{s1} >, < f'_2, t_{s2} >, ..., < f'_i, t_{si} >, ..., < f'_d, t_{sd} >).$$

Here, for $i \in [1..d]$, $f'_i(t) = v$.

Then $dist(\tilde{P}, \tilde{R})$ can be computed on the *pairs of sub-sequences*, whose number is usually much smaller than the number of *pairs of data points*:

$$dist(\tilde{P}, \tilde{R}) = \sqrt{\sum_{i=1}^{d} dist^2(< f_i, t_{si} >, < f'_i, t_{si} >)}. \tag{27}$$

Here, $dist^2(< f_i, t_{si} >, < f'_i, t_{si} >)$ denotes the distance square of the i^{th} pair of sub-sequences.

$$dist^2(< f_i, t_{si} >, < f'_i, t_{si} >) = \sum_{t=t_{si}}^{t_{ei}} (f_i(t) - v)^2. \tag{28}$$

In the equation, t_{ei} denotes the ending point of the i^{th} sub-sequence. We observe that, since $t_{si}, t_{si}+1, t_{si}+2, ..., t_{ei}-1, t_{ei}$ is an arithmetic sequence, $(f_i(t)-v)^2 = f_i^2(t) - 2vf_i(t) + v^2, t \in [t_{si}...t_{ei}]$ will be a combination of a set of *well-defined sequences*, each of which has an equation for fast sum computation. The sum in Equation (28) can be computed as a combination of the sums of these well-defined sequences, which avoids decompressing the sub-sequences. Next we explain how to compute the sum for the different types of functions used in the AA algorithm including m^{th} order polynomials ($m = 1, 2$) and exponential functions.

1^{th} *order polynomial (linear function):* If $f_i(t)$ is a linear function $\tilde{y} = at + b$, then,

$$\begin{array}{l} dist^2(< f_i, t_{si} >, < f'_i, t_{si} >) \\ = \sum_{t=t_{si}}^{t_{ei}} (at + b - v)^2 \\ = a^2 \sum_{t=t_{si}}^{t_{ei}} t^2 + 2a(b-v) \sum_{t=t_{si}}^{t_{ei}} t + \sum_{t=t_{si}}^{t_{ei}} (b-v)^2. \end{array} \tag{29}$$

In the equation, $t^2, t, (b-v)^2, t \in [t_{si}...t_{ei}]$ are well-defined sequences. Their sums can be computed efficiently with basic arithmetic computation, e.g., $\sum_{t=t_{si}}^{t_{ei}} t = \frac{(t_{si}+t_{ei})(t_{ei}-t_{si}+1)}{2}$.

2^{th} *order polynomial (quadratic function):* If $f_i(t)$ is a quadratic function $\tilde{y} = at^2 + bt + c$, then,

$$\begin{array}{l} dist^2(< f_i, t_{si} >, < f'_i, t_{si} >) \\ = \sum_{t=t_{si}}^{t_{ei}} (at^2 + bt + c - v)^2 \\ = a^2 \sum_{t=t_{si}}^{t_{ei}} t^4 + 2ab \sum_{t=t_{si}}^{t_{ei}} t^3 + [b^2 - 2a(c-v)] \sum_{t=t_{si}}^{t_{ei}} t^2 \\ \quad -2b(c-v) \sum_{t=t_{si}}^{t_{ei}} t + \sum_{t=t_{si}}^{t_{ei}} (c-v)^2. \end{array} \tag{30}$$

Here, $t^4, t^3, t^2, t, (c-v)^2, t \in [t_{si}...t_{ei}]$ are well-defined sequences. Note that when m increases, the sums of the well-defined sequences become more complex and the computation efficiency decreases. When $m > 2$ the computation becomes very inefficient. This is similar to the AA algorithm. Therefore, we do not consider polynomials of higher orders.

Exponential function: If $f_i(t)$ is an exponential function $\tilde{y} = be^{at}$, then,

$$\begin{array}{l} dist^2(< f_i, t_{si} >, < f'_i, t_{si} >) \\ = \sum_{t=t_{si}}^{t_{ei}} (be^{at} - v)^2 \\ = b^2 \sum_{t=t_{si}}^{t_{ei}} e^{2at} - 2bv \sum_{t=t_{si}}^{t_{ei}} e^{at} + \sum_{t=t_{si}}^{t_{ei}} v^2. \end{array} \tag{31}$$

In the equation, $e^{2at}, e^{at}, t \in [t_{si}...t_{ei}]$ are geometric sequences and their sums can be computed efficiently.

Next we consider distance between the reference time series. Due to the "*curse of dimensionality*", distance-to-reference-point based high-dimensional indexes (e.g., [20]) work better when the data are clustered and the reference points are the cluster centers, so that data points belonging to different clusters can be identified efficiently based on their different distances to the reference points. However, clustering is a very expensive operation, and as time series data are being generated continuously, we may need to re-cluster the data frequently to maintain high-quality clusters, which may be infeasible due to the high cost. Therefore, in this study, instead of the cluster centers, we heuristically choose the reference time series as follows.

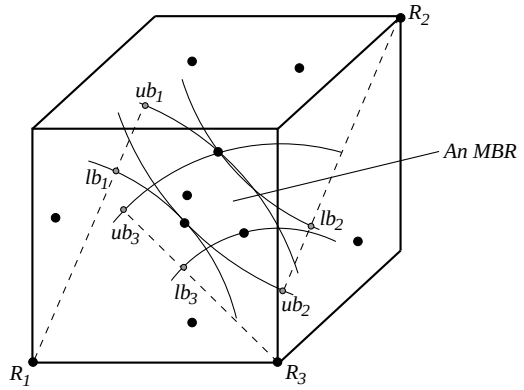


Fig. 6 The three reference time series

We first determine the number of reference time series. Let γ be the number. Then every approximated time series will be mapped to an γ-dimensional space. We propose to use $\gamma = 3$ due to the fact that if $\gamma > 3$, then the approximated time series will be mapped to a relatively high dimensional space, which requires dimension reduction again for indexing. Specifically, we choose the following 3 reference time series to achieve large distances between the reference time series and hence good pruning performance in query processing: $R_1 = (v_{min}, v_{min}, ..., v_{min})$, $R_2 = (v_{max}, v_{max}, ..., v_{max})$ and $R_3 = (v_{max}, v_{min}, ..., v_{min})$, where v_{min} and v_{max} denote the minimum and maximum values of all data points in the time series dataset. Note that in R_3, the first element is different from the rest of the elements. For computing the distance to R_3, the distance contributed by the first element needs to be computed separately. These 3 reference time series represent 3 corners of the N-dimensional data domain that the approximated time series data lie in, as shown in Fig. 6 (assuming that $N = 3$). In our experiments, we will show that an index based on these reference time series achieves good query processing efficiency and it outperforms the scan based approach constantly.

After the mapping, each approximated time series is represented by a 3-element tuple. We use an R-tree to index all the 3-element tuples due to its popularity, although any traditional multi-dimensional index could be used. In the R-tree, the MBR of a node, denoted by $< lb_1, ub_1, lb_2, ub_2, lb_3, ub_3 >$, bounds the distances to the 3 reference time series of all approximated time series indexed in the node, as shown in Fig. 6.

6.2 Processing Queries

When a query is issued, we first map the query time series to a 3-dimensional point. Then this 3-dimensional point is used to perform a search on the R-tree index, which returns a set of approximated time series as the candidate query answers. We check each candidate query answer against the query time series by first decompressing the two time series and then computing the exact distance to identify the final query result.

Range queries: For a range query we also need to map the query range r into the 3-dimensional space. Given the query time series $\tilde{Q}$, the query range r defines a hyper-sphere $\mathcal{S}$ centred at $\tilde{Q}$, as shown in Fig. 7. We need a query MBR in the 3-dimensional space that encloses $\mathcal{S}$ to guarantee no false dismissal. As shown in the figure, on $\mathcal{S}$, the closest and

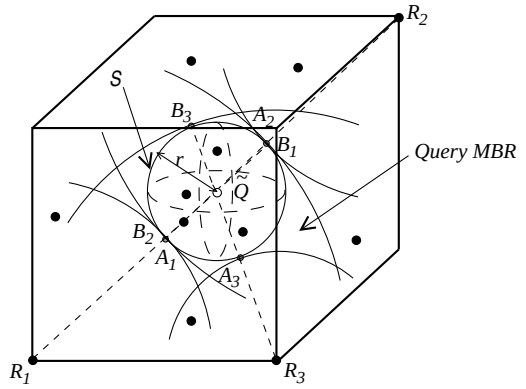


Fig. 7 Query range mapping

Algorithm 4: RQ($\tilde{\mathcal{P}}$, $R_{\tilde{\mathcal{P}}}$, $\tilde{Q}$, r)

```
// P̃: the approximated time series dataset; R_P̃: an R-tree on the mapped data points of P̃;
// Q̃: the approximated query time series; r: the query range;
Query MBR M ← map Q̃, r;
rq(Q̃, r) ← range_query(R_P̃, M); // filtering
for each P̃ ∈ rq(Q̃, r) do
    // refinement
    if dist(P̃, Q̃) > r then
        Remove P̃ from rq(Q̃, r);
Return rq(Q̃, r);
```

farthest points to a reference point R_1 are the two points where the line $R_1\tilde{Q}$ intersects $\mathcal{S}$ (the proof is straightforward and hence omitted), denoted by A_1 and B_1, respectively. Similarly, we can determine the closest and farthest points to R_2 and R_3, as denoted by A_2, B_2, A_3 and B_3, respectively. Further, we can get the distances from these 6 points to the 3 reference points as $dist(\tilde{Q}, R_1) - r$, $dist(\tilde{Q}, R_1) + r$, $dist(\tilde{Q}, R_2) - r$, $dist(\tilde{Q}, R_2) + r$, $dist(\tilde{Q}, R_3) - r$, and $dist(\tilde{Q}, R_2) + r$. The 6 distances form the mapped query range (an MBR) for a range query on the R-tree to retrieve the candidate query answers. As can be seen from Fig. 7, this query range guarantee no false dismissal straightforwardly, but it also introduces some false positives, which will be removed in the refinement stage. The above range query processing procedure is summarized in Algorithm 4, where $R_{\tilde{\mathcal{P}}}$ denotes the R-tree to index the mapped approximated time series.

Complexity Analysis: Let d be the number of sub-sequences in $\tilde{Q}$. Then mapping $\tilde{Q}$ to the 3-dimensional space takes $O(d)$ time. Obtaining the mapped query range from the mapped query time series takes $O(1)$ time, and performing a range query on an R-tree takes $O(\log n)$ time on average, where n denotes the number of approximated time series. Let $\theta \in [0, 100\%]$ be the percentage of approximated time series returned by the range query as the candidate query answers. Then checking the candidate query answers in the refinement stage takes $O(\theta nN)$ time. As a result, processing a range query takes $O(d + 1 + \log n + \theta nN)$ time. The superiority of the proposed algorithm over the straightforward solution relies on the value of θ, which in return is determined by the clustering property of the

approximated time series. As our experiments will show, θ is usually below 40%, which means the proposed algorithm saves at least 60% of the query processing time.

kNN queries: For a kNN query, we first map the query $\tilde{Q}$ to a point q in the 3-dimensional space and then perform a best-first traversal on the R-tree using an adapted version of the best-first R-tree kNN search algorithm [19], as shown in Algorithm 5. The adaptation is in that a kNN of q found during the traversal must be checked against $\tilde{Q}$ directly to filter false positives. The reason is that, the best-first kNN search algorithm relies on the distances between the *data points* and q in the 3-dimensional space, which does not equal to the distance between the *approximated time series* and $\tilde{Q}$ in the original space.

Meanwhile, we observe that the distance between a data point and q is a lower bound of the distance between the approximated time series represented this data point and $\tilde{Q}$. Intuitively, two approximated time series that are closer should have smaller difference in their distances to the reference time series. A strict proof is as follows.

Theorem 1 *Let $\tilde{P}$, $\tilde{Q}$ be two approximated time series and p, q be the two data points that $\tilde{P}$, $\tilde{Q}$ are mapped to in the 3-dimensional space, respectively. We have $dist(\tilde{P}, \tilde{Q}) > \frac{1}{\sqrt{3}} dist(p, q)$.*

Proof According to triangular inequality,

$$dist(\tilde{P}, \tilde{Q}) > |dist(\tilde{P}, \tilde{R_1}) - dist(\tilde{Q}, \tilde{R_1})|$$
$$dist(\tilde{P}, \tilde{Q}) > |dist(\tilde{P}, \tilde{R_2}) - dist(\tilde{Q}, \tilde{R_2})|$$
$$dist(\tilde{P}, \tilde{Q}) > |dist(\tilde{P}, \tilde{R_3}) - dist(\tilde{Q}, \tilde{R_3})|.$$

Thus,

$$\begin{aligned} 3dist^2(\tilde{P}, \tilde{Q}) > {} & [dist(\tilde{P}, \tilde{R_1}) - dist(\tilde{Q}, \tilde{R_1})]^2 + [dist(\tilde{P}, \tilde{R_2}) - dist(\tilde{Q}, \tilde{R_2})]^2 \\ & + [dist(\tilde{P}, \tilde{R_3}) - dist(\tilde{Q}, \tilde{R_3})]^2. \end{aligned} \tag{32}$$

By definition,

$$p = \langle dist(\tilde{P}, \tilde{R_1}), dist(\tilde{P}, \tilde{R_2}), dist(\tilde{P}, \tilde{R_3}) \rangle,$$
$$q = \langle dist(\tilde{Q}, \tilde{R_1}), dist(\tilde{Q}, \tilde{R_2}), dist(\tilde{Q}, \tilde{R_3}) \rangle.$$

Thus,

$$\begin{aligned} dist^2(p, q) = {} & [dist(\tilde{P}, \tilde{R_1}) - dist(\tilde{Q}, \tilde{R_1})]^2 + [(dist(\tilde{P}, \tilde{R_2}) - dist(\tilde{Q}, \tilde{R_2})]^2 \\ & + [dist(\tilde{P}, \tilde{R_3}) - dist(\tilde{Q}, \tilde{R_3})]^2. \end{aligned} \tag{33}$$

Combining Inequality (32) and Equation (33), we have, $dist(\tilde{P}, \tilde{Q}) > \frac{1}{\sqrt{3}} dist(p, q)$.

Based on the theorem, we can compute a distance lower bound between a data point and q to prioritize the data points for the traversal. Once a data point traversed has a distance lower bound to q that is larger than or equal to the distance between $\tilde{Q}$ and its currently found k^{th} NN, we can early terminate the traversal and return the current kNN.

Complexity Analysis: Mapping $\tilde{Q}$ to the 3-dimensional space takes $O(d)$ time. A best-first search on the R-tree takes $O(n \log n)$ time on average. Let ωn be the number of approximated time series checked during the traversal. Then the checking takes $O(\omega n N)$ time, and using a heap of size k to identify the kNN from these ωn time series takes $O(\omega n \log k)$ time. As a result, processing a kNN query takes $O(d + n \log n + \omega n N + \omega n \log k)$ time in total. As our experiments will show, ω is usually very small and our indexed based algorithm can achieve very high query processing efficiency. On average our algorithm saves 90% of the query processing time when compared with the scan based method.

Algorithm 5: KNNQ($\tilde{\mathcal{P}}$, $R_{\tilde{\mathcal{P}}}$, $\tilde{Q}$, k)

```
// P̃: the approximated time series dataset; R_P̃: an R-tree on the mapped data points of P̃;
// Q̃: the approximated query time series; k: the query parameter;
A sized k max-heap H ← ∅;
Query point q ← map Q̃;
A priority query pq ← R_P̃.root.entries;
while not pq.empty() do
    e ← pq.dequeue();
    if distancelowerbound(e, q) ≥ dist(P̃, H.top) then
        break;
    if e.childnode then
        // e is a non-leaf node entry;
        pq.enqueue(e.childnode.entries);
    else
        // e is a leaf node entry;
        if dist(e.P̃, Q̃) < dist(P̃, H.top) then
            Remove H.top;
            Insert e.P̃ into H;
knn(Q̃, k) ← H;
Return knn(Q̃, k);
```

7 Experimental Study

In this section, we first evaluate the compression performance of our time series approximation algorithm in Section 7.1, and then evaluate the query efficiency of our approximated time series indexing scheme in Section 7.2. All experiments are performed on a desktop computer with a 3.20 GHz 6-core Intel® Xeon® CPU and 6 GB memory. Both synthetic and real datasets are used in the experiments.

7.1 Time Series Approximation

We compare our AA algorithm with the state-of-the-art algorithm FSW [27]. We use the linear function, the quadratic function and the exponential function as the candidate functions in our implementation.

Our goal is to obtain a more compact approximation with a user-specified error bound on each data point. Therefore, we measure the performance by the *compression ratio*, which is defined as the size (in terms of bytes) of a time series P divided by the size of its corresponding approximated time series $\tilde{P}$:

$$\text{Compression ratio} = \frac{sizeof(P)}{sizeof(\tilde{P})}$$

Here, $sizeof(P)$ is computed as the number of data points m in P multiplied by the size of a data point. We use an 8-byte double precision floating point number for every data point. Hence, $sizeof(P) = 8m$. The value of $sizeof(\tilde{P})$ is computed as follows. As discussed in Section 3.1, a function of n_{c_i} coefficients used to approximate a segment is represented by $n_{c_i} - 1$ coefficients (8-byte double precision floating point numbers) plus the starting

timestamp of the segment (a 4-byte integer). In addition, $\tilde{P}$ needs to store the first data point in P. Thus, for FSW, $sizeof(\tilde{P}) = 8 + \sum_{i=1}^{k_{fsw}}(8(n_{c_i} - 1) + 4)$, where k_{fsw} denotes the number of segments in $\tilde{P}$ approximated by FSW. AA uses multiple types of functions in the approximation. Therefore, it also needs to store the function types. In the experiments, we use 3 function types and hence AA needs 2 extra bits (0.25 bytes) for every function. Thus, for AA, $sizeof(\tilde{P}) = 8 + \sum_{i=1}^{k_{aa}}(8(n_{c_i} - 1) + 4 + 0.25)$, where k_{aa} denotes the number of segments in $\tilde{P}$ approximated by AA.

We also measure the *average (absolute) approximation error*, which indicates how accurate the approximation is. It is defined as the sum of individual *absolute* approximation errors on the data points divided by the number of data points.

The error bounds are expressed as *relative* values in comparison to the maximum value of the data points of the time series.

7.1.1 Synthetic Datasets

In this subsection, we evaluate the AA algorithms using four synthetic time series datasets: a *linear time series dataset*, a *quadratic time series dataset*, a *exponential time series dataset* and a *mixed time series dataset*. Each dataset is generated by first generating a function of random valued coefficients and then using the function to generate a random number (between 1 and 80) of data points. We repeat the generation until we have obtained 1000 data points in each dataset. For the linear, quadratic and exponential time series datasets, the functions are linear, quadratic and exponential synthetic functions, respectively. For the mix time series dataset, each function is randomly chosen among the aforementioned three kinds of synthetic functions. The definitions of these *synthetic functions* are as follow:

(1) Linear Function

$$f(t) = at + b + \epsilon,$$

where the values of the coefficients a and b are randomly chosen from $[-10, 10]$ and $[-20, 20]$, respectively.

(2) Quadratic Function

$$f(t) = at^2 + bt + c + \epsilon,$$

where the values of the coefficients a, b and c are randomly chosen from $[-1, 1]$, $[-10, 10]$ and $[-20, 20]$, respectively.

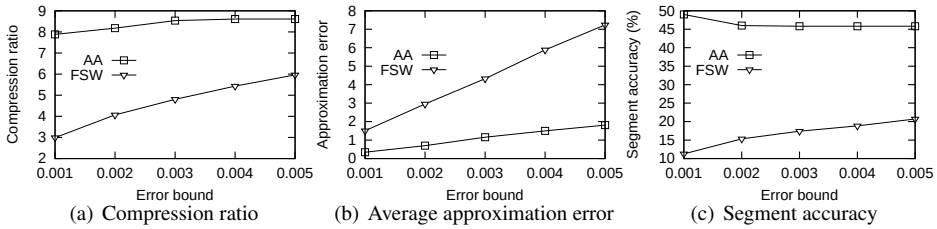
(3) Exponential Function

$$f(t) = be^{at} + \epsilon,$$

where the values of the coefficients a and b are randomly chosen from $[0.05, 1]$ and $[10, 20]$, respectively.

In these functions, ϵ is a random noise in the range of $[-0.5, 0.5]$.

To evaluate the effect of the error bound on the compression ratio and the average approximation error, we use five different error bound values: 0.001, 0.002, 0.003, 0.004 and 0.005. We record the compression ratio and the average approximation error of both the AA algorithm and the FSW algorithms. In addition, we record the "*Segment Accuracy*", which is the percentage of segments in the approximated time series that share the same starting points with those in the original time series generated by the synthetic functions. This measurement indicates the capability of an approximation algorithm to identify the *turning points* of a time series, i.e., the timestamps where data point values change trends.



(a) Compression ratio (b) Average approximation error (c) Segment accuracy

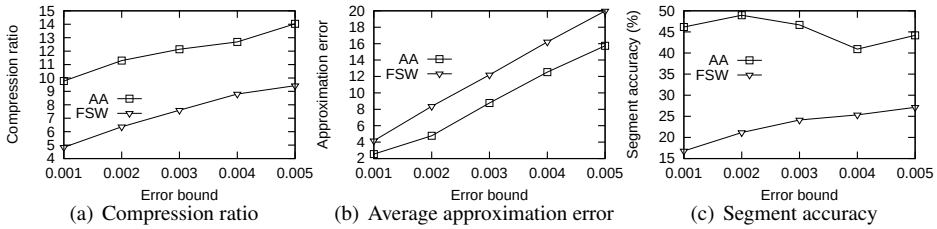
Fig. 8 Error bound: quadratic time series

Quadratic time series: Fig. 8 shows the experimental result of the quadratic time series dataset. We observe that the AA algorithm outperforms the FSW algorithm in all measurements for all error bound values. In terms of compression ratio, the improvement factor of the AA algorithm over the FSW algorithm is more than twice when the error bound is 0.001, while the approximation error produced by the AA algorithm is only 25% of that by the FSW algorithm. Meanwhile, the segment accuracy of AA is close to 50%, which is more than twice of that of FSW. The reason is that, for the dataset with quadratic patterns, the AA algorithm can accommodate these patterns and approximate the time series through adaptively choosing quadratic functions while the FSW algorithm can only approximate the time series by linear functions. Furthermore, when the error bound becomes larger, the average approximation error of the FSW algorithm increase dramatically with a relatively slow rise in the compression ratio. In comparison, the AA algorithm keeps steady in all three measurements because it has already accommodated to the patterns of the time series very well since a small error bound. This shows the robustness of the AA algorithm.

Note that the 50% segment accuracy of AA indicates that about half of its segments have the same starting points as those generated by the synthetic functions. This effectively means that the AA algorithm has successfully identify almost all the turning points in the original time series. The reason is as follows. In the original time series, two consecutive *synthetic segments* are generated by different synthetic functions. There is usually a large gap between the ending point of a synthetic segment and the starting point of its following synthetic segment, and the time series starts a new trend at a new synthetic segment. The AA algorithm requires smooth segmentation, i.e., the ending point of a segment must be the starting point of its following segment. As a result, the AA algorithm needs a "smoothing" segment to bridge the gap between the ending point and the starting point of two consecutive synthetic segments. These smoothing segments will not have the same starting points as those of the synthetic segments. Meanwhile, the AA algorithm needs to generate segments to approximate the synthetic segments themselves, which can have the same starting points as those of the synthetic segments. As the results show, the AA algorithm usually needs to generates only 1 segment to approximate each synthetic segment plus 1 segment to smooth the approximation between two segments, and hence achieves near 50% segment accuracy.

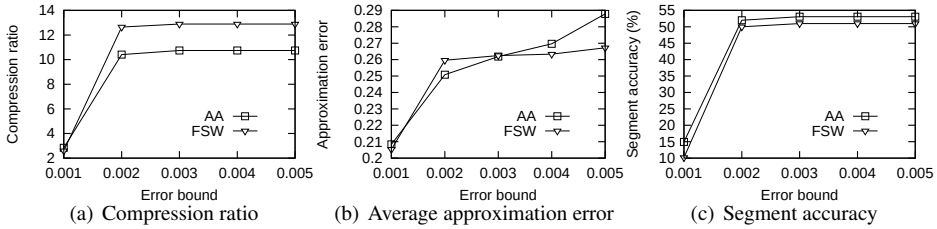
Exponential time series: Fig. 9 shows the experimental result of the exponential time series dataset. The comparative performance of the AA algorithm and the FSW algorithm is similar to that of the quadratic time series datasets. The compression ratio and the segment accuracy of the AA algorithm is twice as large as that of the FSW algorithm on average and the approximation error of the AA algorithm is smaller than that of the FSW algorithm.

Linear time series: The time series with linear patterns is the best case for the FSW algorithm. From Fig. 10, we observe that the AA algorithm is very close to the FSW algorithm in all three measurements. This is because the AA algorithm adapts to linear functions to approximate the linear time series such that its approximation results are mostly the same



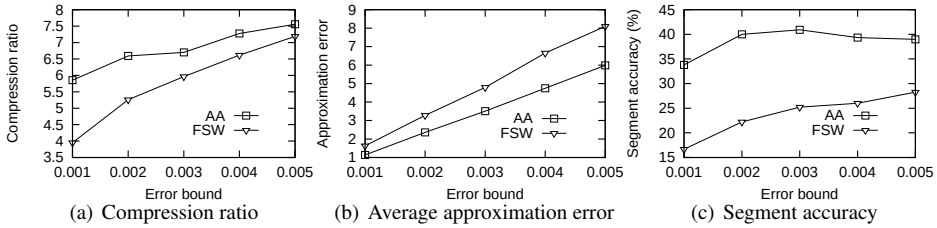
(a) Compression ratio (b) Average approximation error (c) Segment accuracy

Fig. 9 Error bound: exponential time series



(a) Compression ratio (b) Average approximation error (c) Segment accuracy

Fig. 10 Error bound: linear time series

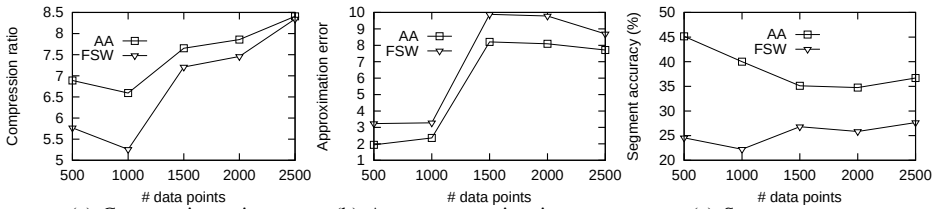


(a) Compression ratio (b) Average approximation error (c) Segment accuracy

Fig. 11 Error bound: mixed time series

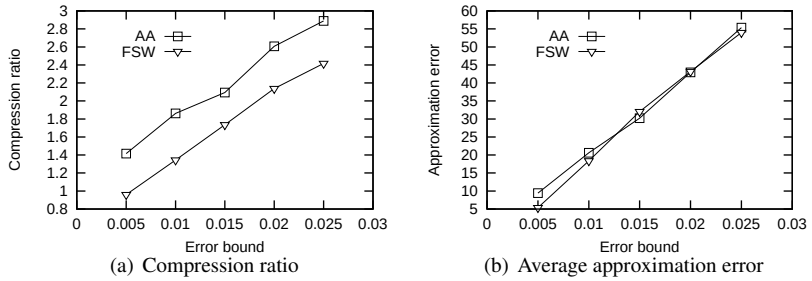
as those of the FSW algorithm. The slightly smaller compression ratio of the AA algorithm is due to the space used to store the types of the functions used in the approximation.

Mixed time series: In real world scenarios, time series data do not follow a constant pattern. We use the mixed time series dataset to simulate time series of varying patterns and evaluate the adaptive mechanism of the AA algorithm. As shown in Fig. 11, the compression ratio and segment accuracy of the AA algorithm are constantly larger than those of the FSW algorithm while the average approximation error of AA is smaller. The result confirms that the AA algorithm can adapt to the change of patterns far better than the FSW does through adaptively choosing appropriate approximation functions.



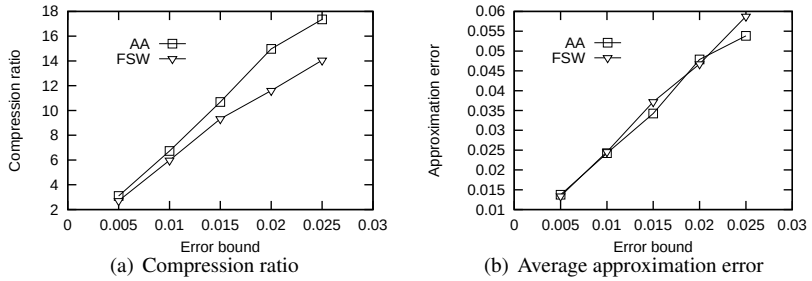
(a) Compression ratio (b) Average approximation error (c) Segment accuracy

Fig. 12 Length: mix time series



(a) Compression ratio (b) Average approximation error

Fig. 13 Error bound: speech time series



(a) Compression ratio (b) Average approximation error

Fig. 14 Error bound: memory time series

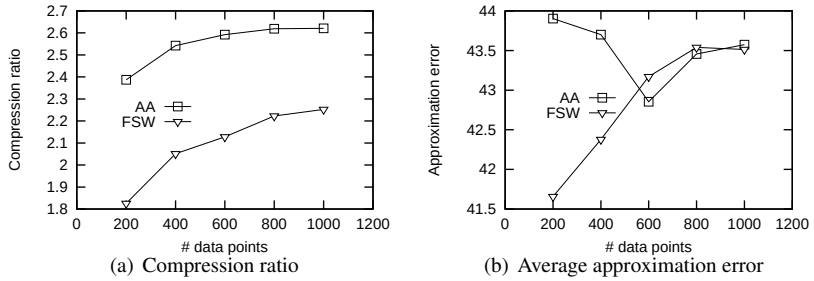
Time series length: We vary the lengths of the time series from 500 to 2500 data points and set the relative error bound at 0.002. The experimental result of the mixed time series is shown in Fig. 12. Again, the AA algorithm outperforms the FSW algorithm in all measurements constantly. This demonstrates that the AA algorithm scales well with the length of the time series. Results of other synthetic time series datasets show similar behavior and hence are omitted.

7.1.2 Real Datasets

In this subsection, we evaluate the algorithms using two real time series datasets: the *Memory dataset* and the *Speech dataset*, which have irregular patterns (e.g., neither polynomial nor exponential patterns), from the UCR Time Series Data Mining Archive [21].

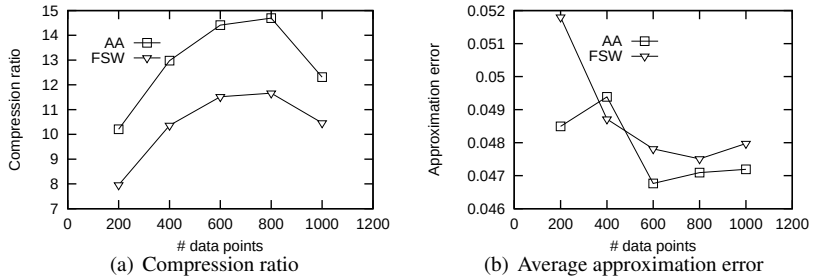
Error bound: To evaluate the effect of the error bound, we vary its value from 0.005 to 0.025 and keep the length of the time series at 500 data points. We show the experimental results of the Speech and Memory datasets in Figs. 13 and 14, respectively. For both datasets, the AA algorithm obtains larger compression ratio values than those of the FSW algorithm. This is because the AA algorithm can adapt to different patterns using different candidate functions. This finding asserts that the AA algorithm can achieve more compact approximation than the FSW algorithm does in practical workloads. The approximation errors of the two algorithms are similar because the patterns of these real datasets are irregular and no candidate function in the AA algorithm can perfectly accommodate these patterns.

Time series length: We also evaluate the algorithms when the lengths of the real time series are varied. Due to the limited data points in the real time series, we vary the lengths from 200 to 1000 data points. We set the error bound at 0.02. The experimental results are plotted in Figs. 15 and 16. We observe that: (i) the AA algorithm outperforms the FSW algorithm in terms of the compression ratio constantly, and (ii) the changes in the average



(a) Compression ratio (b) Average approximation error

Fig. 15 Length: speech time series



(a) Compression ratio (b) Average approximation error

Fig. 16 Length: memory time series

approximation error of both methods have quite different patterns, but the values of both methods are always similar (note the scale). Overall, for real datasets, the AA algorithm also scales well with the length of the time series.

7.1.3 Summary

Experiments in this section verified that the AA algorithm achieves more compact representation of time series than the FSW algorithm does for almost all the time series datasets except the linear case. When the time series follows linear patterns, the performance of the AA algorithm is very close to that of the FSW algorithm. Meanwhile, the average error produced by the AA algorithm is smaller than that by the FSW algorithm in most cases.

7.2 Similarity Search on Approximated Time Series

We now verify the effectiveness and efficiency of our mapping based indexing scheme in processing similarity searches.

In the experiments we use the exchange rate data from the Reserve Bank of Australia [1], which contains daily exchange rates of different currencies based on the Australian Dollar in the past 30 years. Due to reasons like introducing new currencies (e.g. the Euro (EUR)) or discontinuing existing currencies (e.g., the Deutsche Mark (DEM)), the data is not complete for every currency. We preprocess the data and extract a total of 308 time series, each of

[1] http://www.rba.gov.au/statistics/hist-exchange-rates/

Parameter	Values
Dataset cardinality	3080, **15400**, 30800, 154000, 308000
Error bound	0.005, 0.01, 0.015, **0.02**, 0.025
r	0.001%, 0.002%, 0.01%, **0.02** $\%$, 0.1%
k	**1**, 5, 10, 50, 100

Table 3 Experimental settings

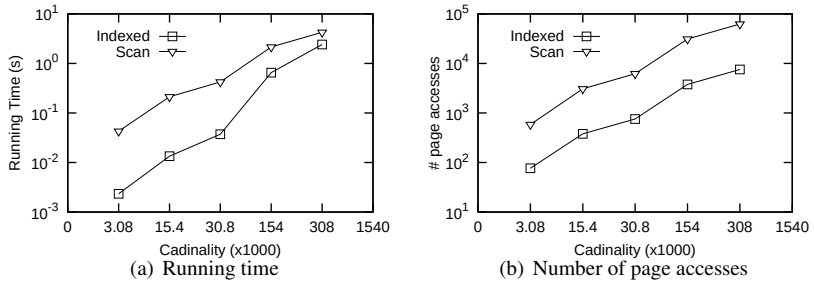
Currency (2010)	NN Currency (Year)	NN Normalized Currency (Year)
United States Dollar	Canada Dollar (2010)	UAED (2010)
Chinese Renminbi	Chinese Renminbi (1995)	Hong Kong Dollar (2010)
Japanese Yen	Japanese Yen (2004)	Singapore Dollar (2010)
Euro	United States Dollar (1987)	UK Pound Sterling (2003)
South Korean Won	Italian Lira (1988)	Canada Dollar (2010)
Singapore Dollar	New Zealand Dollar (2010)	New Zealand Dollar (2010)
Thai Baht	New Taiwan Dollar (2010)	Malaysian Ringgit (2010)
New Zealand Dollar	Singapore Dollar (2010)	Singapore Dollar (2010)
New Taiwan Dollar	Thai Baht (2010)	Singapore Dollar (2010)
Malaysian Ringgit	Malaysian Ringgit (2007)	Singapore Dollar (2010)
Indonesian Rupiah	Indonesian Rupiah (2008)	Chinese Renminbi (2010)
Vietnamese Dong	Vietnamese Dong (2011)	Hong Kong Dollar (2010)
United Arab Emirates Dirham (UAED)	Saudi Arabian Riyal (2010)	United States Dollar (2010)
Papua New Guinea Kina	Chinese Renminbi (1987)	Singapore Dollar (2010)
Hong Kong Dollar	South African Rand (2010)	Saudi Arabian Riyal (2010)
Canada Dollar	United States Dollar (2010)	Chinese Renminbi (2010)
South African Rand	Swedish Krona (2011)	Thai Baht (2010)
Saudi Arabian Riyal	UAED (2010)	United States Dollar (2010)
Swiss Franc	Canada Dollar (1999)	Swedish Krona (2010)
Swedish Krona	Chinese Renminbi (2010)	Chinese Renminbi (1991)

Table 4 Currencies with similar exchange rates or exchange rate changing patterns

which contains the exchange rates of a currency of 365 days. We call this the **Base** dataset and each time series in the dataset a **Base** time series. For scalability tests we generate synthetic datasets from the Base dataset by adding a random noise to each of the data points. The random noise range varies from 0.5% to 2.5% of the data domain. We generate synthetic datasets with the cardinality ranging from 3,080 (generating 10 synthetic time series from each Base time series) to 308,000 (generating 1000 synthetic time series from each Base time series). We apply the AA algorithm on the real and synthetic datasets to obtain datasets of approximated time series, varying the error bound from 0.005 to 0.025. In the experiments we vary the range query parameter r from 0.001% to 0.1% of the data domain and the kNN query parameter k from 1 to 100. When an R-tree is used, we set the page size at 4 KB. Table 3 summarizes the parameters, where values in bold denote the default values.

7.2.1 Validity

In this subsection we verify the validity of using Euclidean distance as the similarity metric. We first perform an NN query for every Base time series and check whether the two currencies represented by a time series and its NN have similar exchange rates. Then we normalize the time series by dividing the mean value of a time series at its each data point and perform NN queries on the normalized time series. By doing so our aim is to find the time series with similar value changing patterns.



(a) Running time (b) Number of page accesses

Fig. 17 Range query costs vs. dataset cardinality

Table 4 shows the NN results for 20 currencies' time series of year 2010. By examining the exchange rates of the currencies listed in the table, we find that the NN searches using Euclidean distance as the similarity metric successfully identify the currencies with similar exchange rates and/or exchange rate changing patterns in certain periods.

In terms of exchange rates, United States Dollar and Canada Dollar are similar (ranging between 0.8 and 1.0) in 2010. Similarly, Singapore Dollar and New Zealand Dollar have close exchange rates in 2010, and so are Thai Baht and New Taiwan Dollar. Some currencies have similar exchange rates to other currencies in different periods (e.g., Euro (2010) and United States Dollar (1987)). Others have similar exchange rates to themselves but in different periods (e.g., Chinese Renminbi (2010) and Chinese Renminbi (1995)).

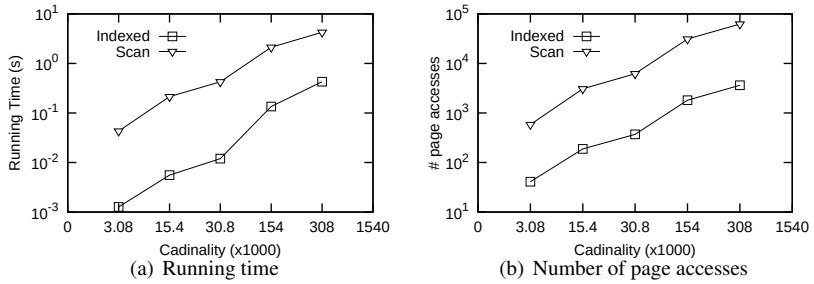
In terms of exchange rate changing patterns, most NN pairs appear in the same year, i.e., 90% (18 out of 20). Among them, some currencies have fixed exchange rates to some other currencies. For example, United Arab Emirates Dirham and Saudi Arabian Riyal both have fixed exchange rates to United States Dollar. For the other currencies, the similar exchange rate changing patterns in the same year are probably because some countries are closely related and have mutual economic influence on each other. The reasons for the 2 exceptions are to be explored. These findings may help reveal patterns in exchange rate variation and support business or financial decision marking for financial organizations and governments.

7.2.2 Efficiency

Next we verify the query processing efficiency of the proposed mapping based index. We use 308 approximated time series generated from the Base dataset as the query time series and measure the average query processing time and number of page accesses per query. As existing techniques requires that every sub-sequence in the approximated time series is represented by the same type of function and with the same length, they do not apply in our approximated time series. Thus, we compare our index based query processing scheme (denoted by "Indexed") with the scan based query processing approach (denoted by "Scan") as described in Section 6.

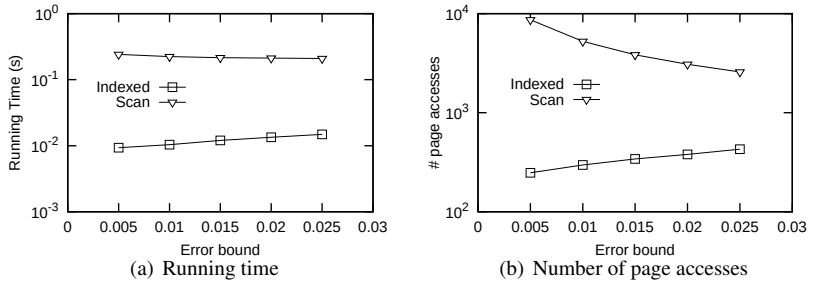
Cardinality: Figs. 17 and 18 show the query processing costs when the dataset cardinality is varied. From the figures we can see that the costs of both methods increase with the dataset cardinality, and that our proposed algorithm outperforms the baseline algorithm constantly in terms of both query processing time and number of page accesses constantly.

For range queries the costs of our indexed based method increases slightly faster than those of the scan method. This is because as the number of synthetically generated time series increases, the distribution of the time series becomes less clustered gradually. Even so,



(a) Running time (b) Number of page accesses

Fig. 18 kNN query costs vs. dataset cardinality



(a) Running time (b) Number of page accesses

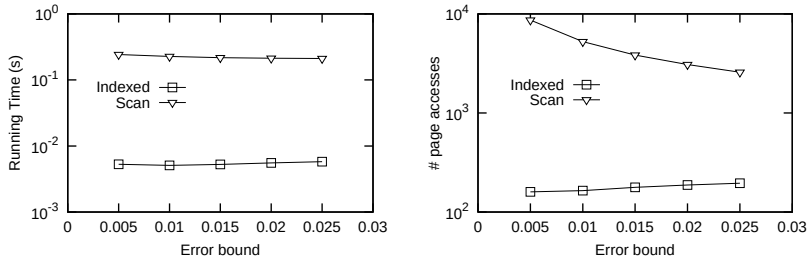
Fig. 19 Range query costs vs. approximation error bound

our indexed based method still outperforms the scan method significantly when the dataset gets large (please note the logarithmic scale). It reduces the query processing time by at least 44%, and the number of page accesses by at least 80%. Here, the save in page access is more than that in query processing time. This is because our indexing based method has a different processing time cost per page access from that of the scan method. Our method requires distance computations on the R-tree pages, while the scan method requires decompressing computations on the approximated time series data pages.

For kNN queries our proposed method outperforms the scan method by at least an order of magnitude in both measures. This demonstrates the pruning capability of our mapping based index. For a dataset of 308,000 time series our algorithm can provide the query answer in 0.43 seconds, while the scan method requires 4.3 seconds. In real applications, queries are issued in hundreds or even thousands simultaneously. The scan method is obviously too slow to process queries in such scales. On contrast, our method can process hundreds of kNN queries in just a few seconds on a commodity desktop computer. With an upgrade in the hardware, the query processing time of our method can be further reduced to within a second, which should be an acceptable time for the users.

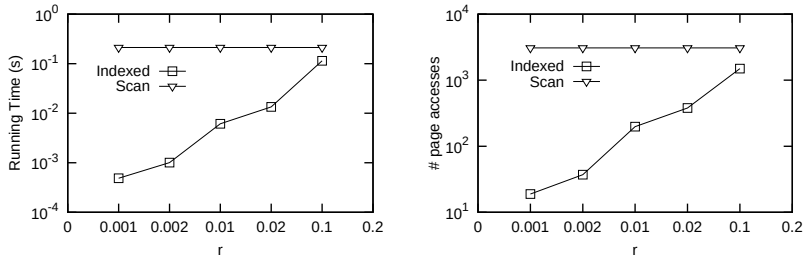
Error bound: Figs. 19 and 20 show the query processing costs when the approximated time series are generated with different approximation error bounds. When the error bound is at 0.025 (which is quite large for a meaningful approximation), our proposed method outperforms the scan method by more than an order of magnitude in terms of both query processing time and number of page accesses for range and kNN query processing.

We observe that when the approximation error bound increases the costs of our method increase slightly because the approximated time series get distributed more evenly. The number of page accesses of the scan method, on the other hand, drops gradually. This is because



(a) Running time (b) Number of page accesses

Fig. 20 kNN query costs vs. approximation error bound



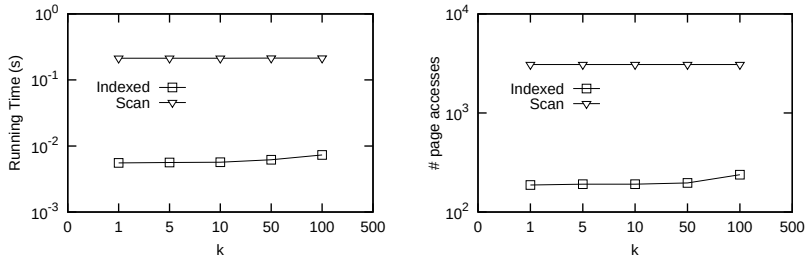
(a) Running time (b) Number of page accesses

Fig. 21 Range query costs vs. r

when the approximation error bound increases, a function can approximate a longer subsequence. It takes less functions to approximate a time series and the data file size decreases. Few page accesses are required for the scan method to read all the approximate time series. However, the query processing time of the scan method stays stable because the number of approximate time series and data points in each time series do not change. Similar amount of time is still needed to decompress and compare all the time series.

Range query parameter r: Fig. 21 shows the query processing costs when we vary r. From the figure we can see that our indexed based method outperforms the scan method under various values of r. As expected, when r increases the costs of our method increase gradually because the pruning capability of the index decreases. Meanwhile, the increase of r affects little of the scan method since it has to scan the whole dataset anyway. As a result, the performance of our method approaches the scan method gradually. However, until $r = 0.1\%(v_{max} - v_{min})$, which is a very large query range as the dimensionality is high, our proposed method still saves at least 56% of the query process costs, which demonstrates the robustness of our method.

kNN query parameter k: Fig. 22 shows the query processing costs when k is varied. Our method outperforms the scan method by at least an order of magnitude. This demonstrates the scalability of our index based method in processing kNN queries. Meanwhile, the query processing time of both methods has a slight increase when k increases, which is because a larger heap needs to be maintained for a larger k. Note that the processing time of a kNN query when k increases does not increase as much as that of a range query when r increases. This is because when k increases, the increase in query selectivity is far slower than that when the query range r increases due to the high dimensionality.



(a) Running time (b) Number of page accesses

Fig. 22 kNN query costs vs. k

8 Conclusions

We proposed an online segmentation algorithm that approximates time series by a set of candidate functions (polynomials of different orders, exponential functions, etc.) and adaptively chooses the most compact one as the pattern of the time series changes. We further proposed a novel method to efficiently generate the compact approximation of a time series in an online fashion for several types of candidate functions. This method incrementally narrows the feasible coefficient space of candidate functions in the coefficient coordinate system and find the farthest data point a segment can approximate given an error bound on each data point. To support efficient similarity searches on the approximated time series, we proposed an indexing scheme that indexes the similarity values between the approximated time series and a set of reference time series. We proposed an efficient method to compute the similarity values for indexing, which overcame the difficulty in computing the similarity between two approximated time series directly. Extensive experimental results show that our approximation algorithm achieves higher compression ratio than the state-of-the-art algorithm FSW does, and our indexing scheme outperforms a scan based search method by at least an order of magnitude in processing kNN queries and 40% in processing range queries.

Acknowledgements Rui Zhang is supported by the Australian Research Council's Future Fellow funding scheme (project number FT120100832).

References

1. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. In: FODO, pp. 69–84 (1993)
2. Appela, U., Brandta, A.V.: Adaptive sequential segmentation of piecewise stationary time series. In: Information Science, pp. 27–56 (1983)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS, pp. 1–16 (2002)
4. Bellman, R.: On the approximation of curves by line segments using dynamic programming. In: Commun. ACM, p. 284 (1961)
5. Berndt, D.J., Clifford, J.: Finding patterns in time series: A dynamic programming approach. In: Advances in Knowledge Discovery and Data Mining, pp. 229–248. American Association for Artificial Intelligence (1996)
6. Cai, Y., Ng, R.T.: Indexing spatio-temporal trajectories with chebyshev polynomials. In: SIGMOD, pp. 599–610 (2004)
7. Chen, L., Ng, R.T.: On the marriage of lp-norms and edit distance. In: VLDB, pp. 792–803 (2004)
8. Chen, Q., Chen, L., Lian, X., Liu, Y., Yu, J.X.: Indexable pla for efficient similarity search. In: VLDB, pp. 435–446 (2007)

9. Cortes, C., Fisher, K., Pregibon, D., Rogers, A., Smith, F.: Hancock: A language for extracting signatures from data streams. In: SIGKDD, pp. 9–17 (2000)
10. Dacorogna, M., Gencay, R., Muller, U.A., Pictet, O.V., Olsen, R.: An Introduction to High-Frequency Finance. Academic Press (2001)
11. Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., Keogh, E.J.: Querying and mining of time series data: experimental comparison of representations and distance measures. PVLDB **1**(2), 1542–1552 (2008)
12. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. In: SIGMOD, pp. 419–429 (1994)
13. Fisher, K., Gruber, R.: Pads: Processing arbitrary data streams. In: Workshop of AT&T Labs (June 2003)
14. Fu, A.C., Chung, F.L., Ng, V., Luk, R.: Evolutionary segmentation of financial time series into subsequences. In: Evolutionary Computation, pp. 426–430 (2001)
15. Fuchs, E., Gruber, T., Nitschke, J., Sick, B.: Online segmentation of time series based on polynomial least-squares approximations. IEEE Trans. Pattern Anal. Mach. Intell. **32**(12), 2232–2245 (2010)
16. Garofalakis, M., Kumar, A.: Deterministic wavelet thresholding for maximum-error metrics. In: PODS, pp. 166–176 (2004)
17. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.J.: Quicksand: Quick summary and analysis of network data. Tech. Rep. 2001-43, DIMACS (2001)
18. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
19. Hjaltason, G.R., Samet, H.: Ranking in spatial databases. In: SSD, pp. 83–95 (1995)
20. Jagadish, H.V., Ooi, B.C., Tan, K.L., Yu, C., Zhang, R.: idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. **30**(2), 364–397 (2005)
21. Keogh, E., Folias, T.: The UCR time series data mining archive. In: http://www.cs.ucr.edu/ eamonn/TSDMA (2002)
22. Keogh, E.J., Chakrabarti, K., Mehrotra, S., Pazzani, M.J.: Locally adaptive dimensionality reduction for indexing large time series databases. In: SIGMOD, pp. 151–162 (2001)
23. Keogh, E.J., Chakrabarti, K., Pazzani, M.J., Mehrotra, S.: Dimensionality reduction for fast similarity search in large time series databases. Knowl. Inf. Syst. **3**(3), 263–286 (2001)
24. Keogh, E.J., Chu, S., Hart, D., Pazzani, M.J.: An online algorithm for segmenting time series. In: ICDM, pp. 289–296 (2001)
25. Lee, J.G., Han, J., Whang, K.Y.: Trajectory clustering: a partition-and-group framework. In: SIGMOD, pp. 593–604 (2007)
26. Lemire, D.: A better alternative to piecewise linear time series segmentation. In: SIAM Data Mining, pp. 545–550 (2007)
27. Liu, X., Lin, Z., Wang, H.: Novel online methods for time series segmentation. TKDE **20**(12), 1616–1626 (2008)
28. Liu, X., Wu, X., Wang, H., Zhang, R., Bailey, J., Ramamohanarao, K.: Mining distribution change in stock order streams. In: ICDE, pp. 105–108 (2010)
29. Lomet, D.B., Hong, M., Nehme, R.V., Zhang, R.: Transaction time indexing with version compression. PVLDB **1**(1), 870–881 (2008)
30. Morse, M.D., Patel, J.M.: An efficient and accurate method for evaluating time series similarity. In: SIGMOD, pp. 569–580 (2007)
31. Nutanong, S., Tanin, E., Zhang, R.: Incremental evaluation of visible nearest neighbor queries. TKDE **22**(5), 665–681 (2010)
32. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: Analysis and evaluation of v*-*k*nn: an efficient algorithm for moving *k*nn queries. VLDB J. **19**(3), 307–332 (2010)
33. O'Neil, W.: How to Make Money in Stocks (4 edition). McGraw-Hill (2009)
34. O'Rourke, J.: An on-line algorithm for fitting straight lines between data ranges. Commun. ACM **24**, 574–578 (1981)
35. Palpanas, T., Vlachos, M., Keogh, E.J., Gunopulos, D.: Streaming time series summarization using user-defined amnesic functions. TKDE **20**(7), 992–1006 (2008)
36. Papazoglou, M.P.: Web services and business transactions. World Wide Web **6**(1), 49–91 (2003)
37. Rafiei, D., Mendelzon, A.O.: Efficient retrieval of similar time sequences using dft. In: FODO, pp. 249–257 (1998)
38. Shatkay, H.: Approximate queries and representations for large data sequences. In: ICDE, pp. 536–545 (1996)
39. Sullivan, M., Heybey, A.: Tribeca: A system for managing large databases of network traffic. In: USENIX Technical Conference, pp. 13–24 (1998)
40. Vlachos, M., Gunopulos, D., Kollios, G.: Discovering similar multidimensional trajectories. In: ICDE, pp. 673–684 (2002)

41. Xu, Z., Zhang, R., Ramamohanarao, K., Parampalli, U.: An adaptive algorithm for online time series segmentation with error bound guarantee. In: EDBT, pp. 192–203 (2012)
42. Yi, B.K., Faloutsos, C.: Fast time sequence indexing for arbitrary lp norms. In: VLDB, pp. 385–394 (2000)
43. Yu, C., Zhang, R., Huang, Y., Xiong, H.: High-dimensional knn joins with incremental updates. GeoInformatica **14**(1), 55–82 (2010)
44. Zhang, R., Jagadish, H.V., Dai, B.T., Ramamohanarao, K.: Optimized algorithms for predictive range and knn queries on moving objects. Inf. Syst. **35**(8), 911–932 (2010)