Eurographics Conference on Visualization (EuroVis) 2022
R. Borgo, G. E. Marai, and T. Schreck
(Guest Editors)

*Volume 41 (2022), Number 3*

# Streaming Approach to In Situ Selection of Key Time Steps for Time-Varying Volume Data

Mengxi Wu[1], Yi-Jen Chiang[1†], and Christopher Musco[1‡]

[1]Department of Computer Science and Engineering, Tandon School of Engineering, New York University, USA

**Abstract**

Key time steps selection*, i.e., selecting a subset of most representative time steps, is essential for effective and efficient scientific visualization of large time-varying volume data. In particular, as computer simulations continue to grow in size and complexity, they often generate output that exceeds both the available storage capacity and bandwidth for transferring results to storage, making it indispensable to save only a subset of time steps. At the same time, this subset must be chosen so that it is highly representative, to facilitate post-processing and reconstruction with high fidelity. The key time steps selection problem is especially challenging in the* in situ *setting, where we can only process data in* one pass *in an* online streaming *fashion, using a small amount of main memory and fast computation. In this paper, we formulate the problem as that of* optimal piece-wise linear interpolation*. We first apply a method from numerical linear algebra to compute linear interpolation solutions and their errors in an* online streaming *fashion. Using that method as a building block, we can obtain a global optimal solution for the piece-wise linear interpolation problem via a standard dynamic programming (DP) algorithm. However, this approach needs to process the time steps in multiple passes and is too slow for the* in situ *setting. To address this issue, we introduce a novel approximation algorithm, which processes time steps in* one pass *in an* online streaming *fashion, with very efficient computing time and main memory space both in theory and in practice. The algorithm is suitable for the* in situ *setting. Moreover, we prove that our algorithm, which is based on a greedy update rule, has* strong theoretical guarantees *on the approximation quality and the number of time steps stored. To the best of our knowledge, this is the first algorithm suitable for* in situ *key time steps selection with such theoretical guarantees, and is the main contribution of this paper. Experiments demonstrate the efficacy of our new techniques.*

**Keywords:** Algorithms, Temporal Data, Scalar Field Data, Large-Scale Data Techniques, Key Time Steps Selection.

## 1. Introduction

As data sizes continue to grow, scientific visualization of time-varying volume data remains a constant challenge. An essential research question toward this challenge is that of *key time steps selection*, which requires selecting a subset of most representative time steps from a time series of volumes. The selected subset of time steps can be considered a summary or succinct representation for the whole time series, enabling fast analysis and visualization of the time-varying dataset. Previously, most approaches for key time steps selection (e.g., [TLS12,FE17,ZC18,PXvO*19]; see Sec. 2) are designed for the *post-simulation* setting, namely, after the simulation is done and the resulting data is stored. In this setting, a computationally expensive preprocessing phase is used to select key time steps, typically in *multiple passes*. Such time steps are then used in a run-time phase for efficient data exploration and visualization.

An even more pressing need, however, is to solve the key time steps selection problem in the *in situ* setting. This setting is motivated by the reality that current computer simulations often generate output that exceeds both the available storage capacity and bandwidth for transferring simulation output to storage. Accordingly, it becomes necessary to select key time steps *on the fly*, at the same time as time series data is generated. In particular, we must process the time-varying volume data in *one pass* in an *online streaming* fashion; moreover, we can only use a small amount of main memory and computation time. It is important to recognize that in situ key time steps selection is extremely challenging, especially if we hope to select a highly representative subset of time steps to facilitate post-processing and reconstruction with high fidelity. (In the in situ community, this is related to "triggers"(e.g., [KMLC20]): having an inspection routine that studies the current state of a simulation and decides whether or not to "fire", which cues an additional action — visualization, analysis, or saving data to storage ("store"); the problem considers "store".)

In this paper, we take on this challenge, by developing a novel algorithm for key time steps selection that is suitable for the *in situ* setting. We adopt a common approach for formalizing the

† Corresponding author; supported in part by NSF grant CCF-2008768.
‡ Supported in part by NSF grant CCF-2045590.

key time steps selection problem via *reconstruction error*: a set of time steps is considered "good" or "representative" if they can be used to accurately reconstruct the entire time series of volume data [KMLC20, ZC18]. Accordingly, the task of optimal key time steps selection reduces to finding a set of time steps that are best able to reconstruct the time series. In a previous state-of-the-art approach [ZC18], the selected representative time steps are *from the original data*, and any skipped time steps are reconstructed from two consecutive selected time steps $i$ and $j$ by linear interpolation, with the goal of minimizing the total reconstruction error. In other words, selecting time steps corresponds to trying to "fit" the time series by a $k$ piece-wise linear function (for a given integer $k > 0$) to minimize the total "fitting error", with the restriction that the two endpoints of each of the $k$ segments be *from the original data*. We thus call the problem *restricted key time steps selection*. We call the corresponding linear interpolation where the two endpoints are *from the original data* (e.g., time steps $i$ and $j$ above) the *restricted linear interpolation*. We can remove such restriction, however, and the problem becomes *general key time steps selection*, where the two endpoints of each segment can be *anywhere* (and similarly for *linear interpolation* by which we mean the *unrestricted* version from now on). Clearly, the optimal solution for the general problem has lower fitting error than that for the restricted problem, since the optimal solution for the latter is just a special-case solution for the general problem.

In this paper, we work on the *general key time steps selection* problem, by formulating it as that of *optimal piece-wise linear least squares interpolation*; this is exactly the same setting and error metric used in the previous in situ work [MLF*16]. We apply a method from the numerical linear algebra literature (e.g., [GLPW16, BDM*20]) to compute linear interpolation solutions and their errors in an *online streaming* fashion. Using this method as a building block, we can obtain a global optimal solution for the piece-wise linear interpolation problem via a standard dynamic programming (DP) algorithm. This approach improves over the current state of the art, which also uses a DP based solution to solve the restricted key time steps selection [ZC18], in running time and I/O cost, with similar reconstruction error (see Secs. 2, 3.3.2 and the experiments in Sec. 4 for details). However, our DP approach needs to process the time steps in multiple passes and is still too slow for the *in situ* setting. To address this issue, we introduce a novel greedy approximation algorithm, which processes time steps in *one pass* in an *online streaming* fashion, keeping only $O(1)$ time steps in main memory[†], with total computation time and I/O cost both linear in the time-varying data size — all being *optimal*. The algorithm is suitable for the *in situ* setting. Moreover, we prove that our algorithm has *strong theoretical guarantees* on the approximation quality and the number of time steps stored. To the best of our knowledge, this is the first algorithm suitable for *in situ* key time steps selection with such theoretical guarantees. Experiments demonstrate the efficacy of our new techniques.

[†] This space complexity holds given knowledge of an optimal parameter for the algorithm. We remove that assumption in Sec. 3.4.2 with only a logarithmic increase in space complexity, which in practice is just a small constant (Sec. 4).

**Contributions:** The contributions of this paper are as follows.
**(1)** By formulating the general key time steps selection problem into that of optimal piece-wise linear interpolation and applying a method in numerical linear algebra, we obtain an online streaming approach for computing linear interpolation solutions and their errors, which is a building block of our DP and greedy approaches.
**(2)** Based on the building block of **(1)** and standard dynamic programming (DP), we obtain a global optimal solution for the general key time steps selection problem, which improves over the previous state-of-the-art DP method in [ZC18].
**(3)** We devise a novel greedy, online streaming algorithm for the general key time steps selection problem. It is very efficient in computing time and main memory usage, both in theory and in practice, and is the first algorithm suitable for *in situ* key time steps selection with strong theoretical guarantees on the approximation quality and the number of resulting segments. This is our main contribution.

## 2. Previous Work

A closely related problem in video processing is key frame selection from videos. In [LK02], a method based on dynamic programming is used to select key frames by maximizing an energy function. There are many other results in this rich literature, and we refer to [HXL*11, Section II.B] for an excellent survey. Another closely related problem is selecting visualization parameters such as viewpoint, lighting, and so on. The results along this line include [BS05, Gum02, LHV06, MAB*97, CM10]. There has also been work on isosurface-topology analysis for selecting critical isovalues and time steps [SB06, TFO09]. In addition, entropy-based information metrics have been used for isovalue selection [WLS13] and for streamline generation [XLS10]. See the book [CFV*16] for an extensive survey on information-theoretic visualization techniques. For research on *in situ* analysis and visualization, we refer to [BAA*16, HWG*20] and the references therein.

There is also a rich body of work on key time steps selection for time-varying volume data. One group of methods focuses on providing an overview of the data that allows a user to visually decide which time steps to select [WS09a, LS08, AM07, AFM06]. It is difficult to quantify the performance of such methods and they are of course labor intensive; automatic techniques are more desirable for large-scale data. In [AFM06], similar time steps are grouped using a greedy method and one time step is selected from each group. The approach in [WYM08] computes *importance curves* based on the mutual information between adjacent time steps, and selects those time steps that are most dissimilar from their predecessors. Other related methods include those based on the Time Activity Curve approach (TAC) [WFMF02, FMHC07, WS09a, WS09b, LS09b, LS09a] and the *TransGraph* [GW11]. In [FE17], a technique based on minimum-cost flow is given for adaptive time steps selection. The approach initially selects time steps from random samples of regular partitions of the whole time series, and progressively considers additional time steps by random importance sampling.

The methods mentioned so far for key time steps selection are all based on local considerations and have no optimality guarantees. Moreover, they are all for the *restricted* problem. Existing approaches that can produce globally optimal solutions are based

on dynamic programming (DP). The *dynamic time warping (DTW)* method [TLS12] is an in-core DP technique to solve the *restricted* problem optimally, where *constant* interpolation is used for reconstruction. In [ZC18], the *restricted* problem is considered, where restricted linear interpolation is used for reconstruction. In that work, an accurate DP method is given to produce optimal solutions. Then, since the computing time and I/O cost are both high, an approximate method based on multi-pass DP is devised, which achieves optimal I/O cost and a significant run-time speed-up, with the solution quality close to optimal in the experiments (but there is no theoretical bound on the approximation quality). The classical Bellman's DP algorithm [Bel61] can also be used to find the optimal $k$-segmentation of a sequence minimizing the error of a piece-wise *constant* interpolation. As a comparison, our DP is standard and similar to those of [Bel61] and [ZC18] (the accurate one), but achieves improvements over the latter via a streaming method for *linear* interpolation. Comparing to the approximate method of [ZC18], our greedy approach runs significantly faster, has theoretical guarantees while performing well in practice, and is suitable for the *in situ* setting.

The methods discussed so far are for post-simulation. In contrast, the in situ community considers the problem as "triggers" in the *in situ* setting. Work on domain-specific triggers include [SBP*15]; see also references in [KMLC20]. For domain-agnostic triggers, the approaches typically use some metric to measure the variation between consecutive time steps, and make a "store" decision if the variation is high. In [YHSN19] Kernel Density Estimation (KDE) and Kullback-Leibler (KL) divergence are used, and in [LWM*18, KMLC20] entropy is used as the metric. Also, as mentioned before, [MLF*16] works on the *general* key time steps selection problem and uses the sum-of-squared errors as the error metric (exactly the same as ours); it gives a greedy algorithm to construct a sequential piece-wise linear regression model. However, it is not clear how the input parameters of the algorithm control the resulting error and the number of segments. In addition, like all the in situ results above, the method does not come with theoretical guarantees. On the other hand, our greedy algorithm has theoretical guarantees on how the input parameters are related to the approximation quality and the number of resulting segments.

Similar guarantees can also be obtained via an existing coreset based approach for segmenting streaming data [FRVR14] from the machine learning literature. This approach provides a technique for our (general) problem and, in fact, was the starting point of our work. However, the method is complex, and has several limitations: (1) at a minimum the coreset approach needs $O(k(\log T)/\varepsilon^2)$ space to select $k$ segments with solution quality $(1+\varepsilon)\cdot \text{Opt}$ where $\varepsilon \in (0,1)$, Opt is the optimal error and $T$ the number of time steps in the dataset; (2) to implement the method in an online streaming fashion (i.e., for the in situ setting) it must be combined with the high-overhead "merge-and-reduce" technique from the streaming algorithms literature [GIMS20, AHPV04], which adds an additional multiplicative $\log^3(T)$ factor to the space complexity. On the contrary, our Basic Greedy method (Sec. 3.4.1) obtains roughly the same guarantees (see Corollary 3 for details), is easy to implement, and only keeps $O(1)$ time steps in main memory, which is optimal. There has also been work on approximation algorithms for piecewise linear ( [ADLS16]) and polynomial ( [LSX21]) regression in the *non-streaming setting*. However, it is not clear how to adapt these methods to our in situ setting.

Finally, a deep learning approach for key time steps selection for *multivariate* data is recently given in [PXvO*19].

## 3. Our Approach

Given a time-varying volume dataset of $T$ time steps, each with data size $N$ (i.e., the number of scalar values in the volume per time step), we formulate the (general) key time steps selection problem into that of *optimal piece-wise linear interpolation* with $k \leq T$ pieces: partition the $T$ time steps into $k$ ranges where for each range we perform linear least squares interpolation, such that the total interpolation error from all ranges is minimized. Each of the $k$ ranges is also called a "segment" since we use the linear interpolation result, a line segment, to reconstruct any time step within the range.

We first present technical preliminaries from linear algebra in Sec. 3.1. Note that although we use matrices to represent our data, which would have prohibitively high dimensions given our typical volume-data sizes, we actually only maintain these matrices *implicitly* and are able to perform matrix operations in an *online streaming* fashion, keeping only $O(1)$ time steps (i.e., $O(1)$ volumes) in main memory. Such a streaming method for linear least squares interpolation is a key building block in our methods, and is described in Sec. 3.2. We then present our dynamic programming and greedy approaches in Secs. 3.3 and 3.4, respectively.

### 3.1. Technical Preliminaries

**Linear Algebra Notation.** We use $\mathbb{R}^N$ to denote the set of real-valued length $N$ vectors, and $\mathbb{R}^{M\times N}$ to denote the set of real-valued $M \times N$ matrices. For a vector $x$, $x_i$ denotes the $i^{\text{th}}$ entry and for a matrix $X$, $X_{ij}$ denotes the entry in the $i^{\text{th}}$ row and $j^{\text{th}}$ column. Let $X_{i:}$ denote the $i^{\text{th}}$ row of $X$ and let $X_{:j}$ denote the $j^{\text{th}}$ column. For a vector $x \in \mathbb{R}^N$, let $\|x\|_2$ denote the standard Euclidean norm $\sqrt{\sum_{i=1}^N x_i^2}$ and for a matrix $X \in \mathbb{R}^{M\times N}$, let $\|X\|_F$ denote the Frobenius norm $\|X\|_F = \sqrt{\sum_{i=1}^M \sum_{j=1}^N X_{ij}^2}$. We let $\vec{1}_N$ and $\vec{0}_N$ denote $N$-dimensional vectors containing all ones and all zeros, respectively. For a vector $x$ or matrix $X$, we let $x'$ and $X'$ denote the transpose[‡]. For an invertible square matrix $X$, we let $X^{-1}$ denote the inverse.

**Other Notation.** As is standard, we let $[a,b]$ denote a (closed) interval on the real line, which contains points $t$ with $a \leq t \leq b$. We let $(a,b)$ denote the open interval containing points $t$ with $a < t < b$, and e.g. $[a,b)$ denote the interval containing points $t$ with $a \leq t < b$.

**Least Squares Interpolation.** Consider equally spaced integer timesteps $[1,\ldots,T]$. Our methods apply unmodified to unequally spaced time steps (e.g., $[1,4,5,13,\ldots]$), but we assume equal spacing to simplify notation. Given $T$ data values that change over time, which can be represented in a vector $y \in \mathbb{R}^T$, the *linear least squares interpolation* problem is to find $m^*, b^* \in \mathbb{R}$ which minimize the sum-of-squares error, $m^*, b^* = \arg\min_{m,b} \sum_{i=1}^T (m\cdot i + b - y_i)^2$. This problem generalizes to vector-value data, like $N$-dimensional volumes.

[‡] Another standard notation is to use $x^T$ and $X^T$ for transpose. We use $x'$ and $X'$ instead because we already use $T$ for the total number of time steps.

In this case, we have a data matrix $Y \in \mathbb{R}^{T\times N}$, where every row $Y_{i:}$ is an $N$-dimensional vector, representing the $N$-dimensional volume data at the $i$-th time step. The least squares interpolation problem is to find $m^*, b^* \in \mathbb{R}^N$ satisfying:

$$m^*, b^* = \arg\min_{m,b} \sum_{i=1}^{T} \|m\cdot i + b - Y_{i:}\|_2^2 = \arg\min_{m,b} \|AZ - Y\|_F^2, \quad (1)$$

where $A$ is a $T\times 2$ matrix with its first column equal to $t = [1,\ldots,T]$ and its second column equal to $\vec{1}_T$, and $Z$ is a $2\times N$ matrix with its first row equal to $m$ and its second row equal to $b$. Note that $\sum_{i=1}^{T}\|m\cdot i + b - Y_{i:}\|_2^2 = \sum_{i=1}^{T}\sum_{j=1}^{N}(m_j\cdot i + b_j - Y_{ij})^2$, so solving Eq. (1) is equivalent to solving $N$ independent one dimensional linear interpolation problems, one for each data dimension. Let $\mathrm{Err}^*$ denote the optimal interpolation error $\mathrm{Err}^* = \min_{m,b}\|AZ-Y\|_F^2$. Recalling that $t = [1,\ldots,T]$, we use the notation:

$$[m^*, b^*, \mathrm{Err}^*] = \mathrm{BestLinearFit}(t, Y). \quad (2)$$

Our main methods for fitting piece-wise linear functions require a black-box access to an implementation of a BestLinearFit() function that returns the optimal parameters and squared error for a given data set. Since the least squares interpolation problem is a special case of a two-dimensional linear least squares regression, such a function can be implemented using the well known closed form solution (see e.g. [Str16]). We express that solution in matrix notation below.

**Fact 1** (Linear Least Squares Interpolation Solution)**.** *Consider time-steps $1,\ldots,T$ and data matrix $Y\in\mathbb{R}^{T\times N}$. Let $A\in\mathbb{R}^{T\times 2}$ have its first column equal to $t=[1,\ldots,T]$ and second column equal to $\vec{1}_T$. The optimal solution to Eq. (1) satisfies:*

$$\begin{bmatrix} m^* \\ b^* \end{bmatrix} = Z^* = (A'A)^{-1}A'Y. \quad (3)$$

Specifically, the result of $(A'A)^{-1}A'Y$ is a $2\times N$ matrix, and the optimal $m^*$ is its first row, while the optimal $b^*$ is its second row.

We formulate the key time steps selection problem by approximating our time series with a sequence of $k$ linear interpolants, i.e., a piece-wise linear function. An example of such a function with $k=5$ pieces is shown in Fig. 1, and the formal definition is below:

**Definition 1** (Piece-wise linear function)**.** *Consider time points $1,\ldots,T$. A piece-wise linear function $F:\{1,\ldots,T\}\to\mathbb{R}^N$ with $k$ pieces maps these points to $N$ dimensional vectors. $F$ is defined by a set $S_F$ of $k$ tuples:*

$$S_F = \{(s_1,m_1,b_1),\ldots,(s_k,m_k,b_k)\},$$

*where $\{s_1,\ldots,s_k\}$ is a subset of $\{1,\ldots,T\}$, and $1 = s_1 < \ldots < s_k \le T$. $m_i, b_i \in \mathbb{R}^N$ are slope and intercept coefficient vectors. For any time point $x\in\{1,\ldots,T\}$ we have:*

$$F(x) = \begin{cases} x\cdot m_1 + b_1 & \text{for } s_1 \le x \le s_2 - 1 \\ x\cdot m_2 + b_2 & \text{for } s_2 \le x \le s_3 - 1 \\ \quad\vdots & \\ x\cdot m_k + b_k & \text{for } s_k \le x \le T \end{cases} \quad (4)$$

**Problem 2** (Optimal $k$ piece-wise linear interpolation)**.** *Let $\mathcal{F}_k$ be the set of all $k$ piece-wise linear functions for $N$ dimensional data and a specified integer $k$. Given a volume data $Y\in\mathbb{R}^{T\times N}$, the goal is to find an optimal $k$ piece-wise linear approximation to the data. I.e., to return $F^*$ satisfying:*

$$F^* = \arg\min_{F\in\mathcal{F}_k} \mathrm{Cost}(F) \quad \textit{where} \quad \mathrm{Cost}(F) = \sum_{i=1}^{T}\|F(i) - Y_{i:}\|_2^2.$$

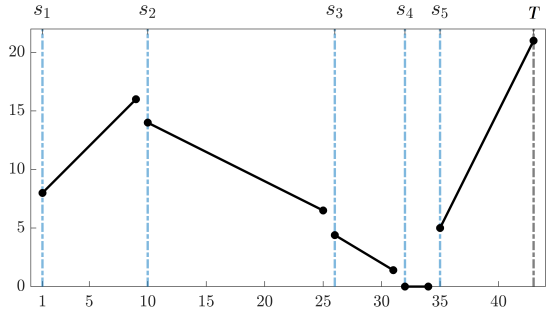


**Figure 1:** *An example of a piece-wise linear function (see Definition 1) with $k=5$ and dimension $N=1$. Note that we always have $s_1 = 1$.*

Note that solving Problem 2 amounts to finding $k$ optimal time steps $s_1^*, s_2^*, \ldots, s_k^*$ to start each of the $k$ segments in $F$, with $s_1^*$ always equal to 1. Once these time steps are chosen, the optimal slope and intercept parameters $m_1,\ldots,m_k$ and $b_1,\ldots,b_k$ can be obtained directly from Fact 1: for the segment with starting point $s_i$, we solve the interpolation problem optimally for our dataset restricted to time points that lie in the interval $[s_i, s_{i+1}-1]$ using an online least squares interpolation method, described below.

### 3.2. Building Block: Online Streaming Method for Linear Least Square Interpolation

As mentioned in Sec. 3, a key building block of our methods is to compute linear least square interpolation (see Fact 1) in an *online streaming* fashion while keeping only $O(1)$ time steps in main memory. To this end, we will apply a method from numerical linear algebra (e.g., [GLPW16, BDM*20]) to perform streaming linear algebra operations in the "row update" or "row arrival" model.

**Computing the Optimal Solution $Z^*$**

Refer to the solution in Fact 1, and recall that the $i$-th time step corresponds to the $i^{\text{th}}$ row of matrices $A$ and $Y$. Thus we want to compute the optimal solution $Z^* = (A'A)^{-1}A'Y$ (see Eq. (3)), where the rows of $A$ and $Y$ arrive one by one. In the process, even though $A$ and $Y$ grow conceptually, we will only use a fixed amount of memory space to maintain $A'A$ (a $2\times 2$ matrix) and $A'Y$ (a $2\times N$ matrix), and update them each time a new row arrives. We can then easily compute the inverse of $A'A$ in $O(1)$ time, and multiply the result by $A'Y$ in $O(N)$ time to obtain the current $Z^*$. In the following, we discuss how to update $A'A$ and $A'Y$ when a new row arrives.

Consider a simple example for $A'A$ when $A$ grows from two rows to three rows (and $A'$ grows from two columns to three columns):

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 3 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \end{bmatrix}.$$

This shows that we only need to multiply the last/new column of $A'$ with the last/new row of $A$, whose result is a $2\times 2$ matrix, and add this result to the original $A'A$ to get the new $A'A$. This is true in general when we go from time step $i$ to $i+1$ for $i\geq 2$. In the same way, we can update $A'Y$ by multiplying the new column of $A'$ with the new row of $Y$ (the result being a $2\times N$ matrix), and adding this result to the current $A'Y$, also a $2\times N$ matrix. Overall, to update the optimal solution $Z^*$, we only need $O(N)$ computing time per time step, using $O(N)$ space (equivalent to $O(1)$ time steps of volume).

In addition to the optimal $Z^*$, we also need to compute the corresponding minimum error, obtained by plugging $Z^*$ into Eq. (1):

$$\text{Err}^* = \|AZ^* - Y\|_F^2 . \tag{5}$$

**Computing the Optimal Error Err$^*$**
Let $tr(M) = \sum_i M_{ii}$ be the *trace* of a square matrix $M$ (the sum of its diagonal entries), and note that

$$\|X\|_F^2 = \sum_i \sum_j X_{ij}^2 = \sum_j \|X_{:j}\|_2^2 = tr(X'X). \tag{6}$$

Expressing Eq. (5) in terms of trace and recalling that $Z^* = (A'A)^{-1}A'Y$ and thus $(Z^*)' = Y'A(A'A)^{-1}$, we have

$$\begin{aligned}\text{Err}^* &= tr((AZ^*-Y)'(AZ^*-Y)) = tr(((Z^*)'A'-Y')(AZ^*-Y))\\ &= tr((Z^*)'A'AZ^* - Y'AZ^* - (Z^*)'A'Y + Y'Y)\\ &= tr(Y'A(A'A)^{-1}(A'A)(A'A)^{-1}A'Y - Y'A(A'A)^{-1}A'Y\\ &\quad - Y'A(A'A)^{-1}A'Y + Y'Y) = tr(Y'Y - Y'A(A'A)^{-1}A'Y)\end{aligned}$$

Applying linearity of the trace we have:

$$\text{Err}^* = tr(Y'Y) - tr(Y'A(A'A)^{-1}(A'Y)). \tag{7}$$

Now we discuss how to compute Err$^*$ in an online streaming fashion. By Eq. (6), we have $tr(Y'Y) = \sum_i \sum_j Y_{ij}^2$, so this term can be easily computed in an online way — each time a new row of $Y$ arrives, we just square each entry of this row and add these squares to $tr(Y'Y)$ to update it. This takes $O(N)$ time since each row of $Y$ is a length $N$ vector for one time step of the volume data.

For the second term in Eq. (7), $tr(Y'A(A'A)^{-1}(A'Y))$, notice that using the above method for updating $Z^*$, we already have $A'Y$ and $(A'A)^{-1}$ available. Letting $W = A'Y$ and $M = Y'A(A'A)^{-1}(A'Y) = W'(A'A)^{-1}W$, our task is to compute $tr(M) = \sum_i M_{ii}$. Observe that the $i^{th}$ diagonal entry of $M$, $M_{ii}$, is obtained by $M_{ii} = (W_{:i})'(A'A)^{-1}W_{:i}$, where $W_{:i}$ is the $i^{th}$ column of $W$. Recall that $W$ is a $2\times N$ matrix, so its $i^{th}$ column is just a *length-2* vector. Also $(A'A)^{-1}$ is just a $2\times 2$ matrix. Therefore using $M_{ii} = (W_{:i})'(A'A)^{-1}W_{:i}$ we can compute each diagonal entry $M_{ii}$ in $O(1)$ time, and computing $tr(M) = \sum_i M_{ii}$ takes $O(N)$ time.

In summary, we can compute the optimal solution $Z^*$ and its corresponding minimum interpolation error Err$^*$ in an online streaming fashion using $O(N)$ computing time per time step, keeping only $O(1)$ time steps (with space $O(N)$) in main memory. These bounds are *optimal* since $Z^*$ itself takes $O(N)$ space to represent.

### 3.3. Accurate Approach via Dynamic Programming

In this section, we present our accurate dynamic programming approach to obtain globally optimal solutions to the general piece-wise linear interpolation problem. This approach follows rather directly from the streaming least squares interpolation method discussed in Sec. 3.2: we present it as a baseline to compare against our faster greedy approximation algorithm, which is our main technical contribution, and because it outperforms the previous method that seeks a globally optimal solution [ZC18].

Recall that $N$ is the volume data size per time step, and $T$ is the total number of time steps in the whole time series. There are two phases which we discuss below: we first perform initial computation to obtain least square interpolation errors for each time range, which then facilitates dynamic programming in the second phase.

#### 3.3.1. Initial Computation

In this first phase, we compute, for each possible time range $[i,j]$ where $1\leq i<j\leq T$ for integers $i,j$, the linear least square interpolation error in this range, denoted by $e(i,j)$. We use the online streaming method discussed in Sec. 3.2 to carry out the task, where $e(i,j)$ is the optimal Err$^*$ for the time range $[i,j]$. In order to maintain low space complexity (i.e., to use just $O(N)$ working memory), we perform the computation in multiple passes. In pass 1, we compute $e(1,2), e(1,3), \cdots, e(1,T)$, in that order, where each $e(,)$ value is obtained by an incremental update using $O(N)$ time (see Sec. 3.2). Thus we complete the pass in $O(NT)$ time. In pass 2, we compute $e(2,3), e(2,4), \cdots, e(2,T)$ in the same way. In general, in pass $i$ (where $i = 1,2,\cdots,T-1$) we compute $e(i,i+1),\cdots,e(i,T)$, in $O(NT)$ time. The total time for all passes is thus $O(NT^2)$. In the out-of-core setting, each pass goes through the whole time series once, with I/O cost $O(NT/B)$ where $B$ is the number of items fitting in one disk block. Therefore the total I/O cost from all passes is $O(NT^2/B)$.

#### 3.3.2. Dynamic Programming

Now we are ready to perform dynamic programming in the second phase. Our goal is to partition the time range $[1,T]$ into $k$ segments such that the sum of the linear least square interpolation errors from all $k$ segments is minimized, for a given $k\in[1,T]$. For a general subproblem, we define a cost function $L(i,k)$ to be the minimum total error of partitioning the time range $[1,i]$ into $k$ segments. Then we have the following recurrence:

$$L(i,k) = \min_{k\leq p\leq i-1} \{L(p-1,k-1) + e(p,i)\} \tag{8}$$

In the base case, we have $L(i,1) = e(1,i)$, i.e., the optimal error of partitioning the range $[1,i]$ into one segment is just $e(1,i)$ for all $i$.

We can solve the recurrence by memorizing the subproblem results computed before. The correctness of dynamic programming can be proved since the problem has the properties of optimal substructure and overlapping subproblems. As given in Eq. (8), to make $k$ partitions in the range $[1,i]$ optimally, we try each valid position of $p$ ($k\leq p\leq i-1$) such that the last segment is for range $[p,i]$ (whose error is $e(p,i)$) and for the remaining range $[1,p-1]$ we partition it into $k-1$ segments optimally (with error $L(p-1,k-1)$). We then take the minimum among the errors from all choices of $p$.

For each value of $k = 2,\cdots,T$, we compute $L(i,k)$ for $i = 2,\cdots,T$ and store its corresponding $p$ that realizes the minimum. The running time for this dynamic programming phase is thus $O(T^3)$. Together

with the initial computation in the first phase, our accurate approach takes $O(NT^2+T^3)$ time. Note that typically $N$ is much bigger than $T$, so the running time is dominated by $O(NT^2)$. In the out-of-core setting, only the first phase needs I/O operations; from Sec. 3.3.1, the total I/O cost is $O(NT^2/B)$.

As a comparison, the DP method of [ZC18] takes $O(NT^3+T^3)$ time, and in the out-of-core setting the I/O cost is $O(NT^3/B)$. The additional $T$ factor in comparison to our method arises from the fact that [ZC18] solves the *restricted* piece-wise linear interpolation problem, which is not amenable to an incremental approach for computing the error of each segment in the initial computation phase. The approximate out-of-core DP method of [ZC18] takes $O(t^2TN+T^3)$ time and optimal $O(TN/B)$ I/O cost, where $t$ is an input parameter for the number of time steps kept in-core. However, the accuracy of the approximate method decreases as $t$ decreases, and there are no theoretical guarantees on this trade-off.

### 3.4. Approximate Greedy Algorithm

In this section, we present our novel greedy algorithm, which is suitable for the *online streaming* and *in situ* settings. It is faster and more I/O efficient than our accurate DP method by a factor of $T$, with runtime just $O(NT)$ and I/O cost $O(NT/B)$. The tradeoff is that the greedy method provides an approximate instead of an exact solution, but we prove strong theoretical bounds on the approximation error, showing that it is small for *any* time-varying volume dataset. The pseudocode is presented below in Algorithm 1 (Sec. 3.4.1). We call this method the "basic" greedy algorithm to contrast it with our "final" greedy algorithm, which is presented in Sec. 3.4.2. The methods are nearly identical, except that the basic algorithm assumes knowledge of a threshold parameter $E$, which is not known ahead of time in practice. The final algorithm handles this issue with a search procedure that allows for automatic tuning of the parameter. We present the basic greedy algorithm first for clarity.

#### 3.4.1. Basic Greedy Algorithm

The method constructs a piece-wise function $\tilde{F}$ from left to right, which is represented as a list of tuples $S_{\tilde{F}} = \{(\tilde{s}_1,\tilde{m}_1,\tilde{b}_1),\ldots,(\tilde{s}_q,\tilde{m}_q,\tilde{b}_q)\}$. The variable $s$ maintains the start point for the current segment being constructed. New data points are then sequentially added to the segment, and every time a new point $j$ is added, we compute the optimal linear interpolation for the segment (Line 4 in Algorithm 1). This can be done using the streaming approach from Sec. 3.2. If the error of the optimal interpolation reaches a fixed error threshold $E$ (we will discuss how to set the value of $E$ shortly), we remove the last point added to the segment, add the segment to $S_{\tilde{F}}$, and restart a new segment. We continue until all $T$ data points have been processed and added to segments.

Note that the streaming method in Sec. 3.2 can be used to implement the BestLinearFit() building block needed in Algorithm 1. So clearly Algorithm 1 has running time and I/O cost both *linear* in the dataset size ($O(NT)$ and $O(NT/B)$ respectively), while keeping only $O(1)$ time steps in main memory. All bounds are optimal.

Our approach is based on the fact that the total error of $\tilde{F}$ is equal

**Algorithm 1** Basic Greedy Algorithm

**Input:** Time steps $[1,\ldots,T]=t$, data matrix $Y\in\mathbb{R}^{T\times N}$, threshold parameter $E>0$.
**Output:** Near optimal piece-wise linear approximation $\tilde{F}$ for $t,Y$, with pieces $S_{\tilde{F}}=\{(s_1,m_1,b_1),\ldots,(s_q,m_q,b_q)\}$.

1: Initialize $S_{\tilde{F}}\leftarrow\{\}$.
2: Initialize $s\leftarrow 1$, $\hat{m}\leftarrow\vec{0}_N$, $\hat{b}\leftarrow Y_{1:}$.
▷ $s$ is the start point of our current segment.
3: **for** $j=2,\ldots,T$ **do**
4: $[m^*,b^*,\mathrm{Err}^*]\leftarrow\mathrm{BestLinearFit}([s,\ldots,j],[Y_{s:},\ldots,Y_{j:}])$.
▷ Recall from Eq. (2) for BestLinearFit().
5: **if** $\mathrm{Err}^*\geq E$ **then**
6: $S_{\tilde{F}}\leftarrow S_{\tilde{F}}\cup\{(s,\hat{m},\hat{b})\}$. ▷ End segment if error is too large and start a new segment at time step $j$.
7: $s\leftarrow j$, $\hat{m}\leftarrow\vec{0}_N$, $\hat{b}\leftarrow Y_{j:}$.
8: **else**
9: $\hat{m}\leftarrow m^*$, $\hat{b}\leftarrow b^*$
10: **end if**
11: **end for**
12: $S_{\tilde{F}}\leftarrow S_{\tilde{F}}\cup\{(s,\hat{m},\hat{b})\}$. ▷ Include the last segment.
13: **return** $S_{\tilde{F}}$

to the sum-of-squared errors for each segment in $\tilde{F}$. Specifically:

$$\mathrm{Cost}(\tilde{F})=\sum_{j=1}^{q}\mathrm{Cost}_j(\tilde{F}) \qquad \text{where}$$

$$\mathrm{Cost}_j(\tilde{F})=\sum_{i=s_j}^{s_{j+1}-1}\left\|m_j\cdot i+b_j-Y_{i:}\right\|_2^2 \qquad \text{is the error for segment } j.$$

By starting a new segment whenever the current segment error reaches $E$, Algorithm 1 ensures that each segment in $S_{\tilde{F}}$ has error $\mathrm{Cost}_j(\tilde{F})<E$. However, it is not clear if 1) the algorithm produces a solution with a small number of segments (ideally not much larger than our target number $k$) and 2) that the *total* error is small. These concerns compete with each other: if we set $E$ to be small, we start a new segment more often, which leads to good error, but a solution with many segments. On the other hand, a large value of $E$ leads to a solution with few segments, but possibly a high error. Nevertheless, we are able to prove that there is a "sweet spot" for Algorithm 1 where we obtain both an accurate solution and one that does not have too many segments. Our major bounds are given below.

**Theorem 2.** *Let $\tilde{F}$ be the piece-wise linear function returned by Algorithm 1 (represented by $S_{\tilde{F}}$). For any $k$, let $F^*$ be the optimal interpolant with $k$ pieces, defined in Problem 2. Then $\tilde{F}$ has $\mathrm{Cost}(\tilde{F})\leq\mathrm{Cost}(F^*)+Ek$ and at most $q=k+\frac{2\cdot\mathrm{Cost}(F^*)}{E}$ segments.*

The above guarantee is natural: the cost of our approximate solution $\tilde{F}$ increases as $E$ increases, while the number of segments in the solution decreases. By setting $E$ appropriately, we can obtain a provable relative error approximation guarantee:

**Corollary 3** (Bi-criteria Approximation)**.** *For any accuracy parameter $0<\varepsilon\leq 1$, if Algorithm 1 is implemented with $E=\frac{\varepsilon\cdot\mathrm{Cost}(F^*)}{k}$ then it returns a piece-wise linear function $\tilde{F}$ with $q\leq\frac{3k}{\varepsilon}$ segments which satisfies:*

$$\mathrm{Cost}(\tilde{F})\leq(1+\varepsilon)\mathrm{Cost}(F^*).$$

Corollary 3 follows directly from Theorem 2 by substituting in the specified value of $E$. It gives a strong *bi-criteria* approximation guarantee: both the error obtained by $\tilde{F}$ and the number of segments contained in $\tilde{F}$ are near optimal, specifically not more than $(1+\varepsilon)$ and $\frac{3}{\varepsilon}$ times the optimal solution, $F^*$, respectively.

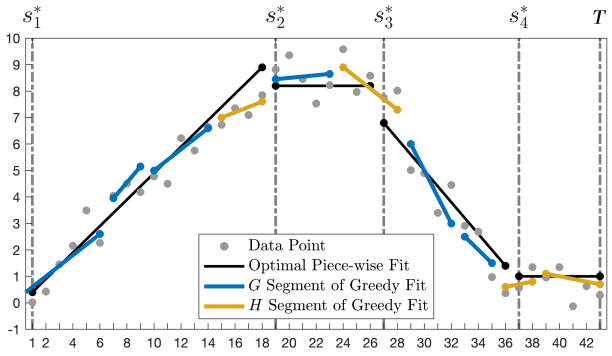


**Figure 2:** *An example k piece-wise linear interpolation problem in $N = 1$ dimension. We show the optimal solution $F^*$ in black, as well as a sample greedy solution obtained by Algorithm 1. The analysis of that algorithm crucially relies on dividing the solution into $G$ and $H$ segments, pictured in blue and orange. $H$ segments are those which either "cross" between two segments in the optimal solution, or terminate at the same time point as an optimal segment.*

**High-Level Insight and Proof Idea.** While the proof of Theorem 2 is non-trivial, we can get across the main points with a visualization that makes it clear why our greedy method performs well. Referring to Figure 2, consider dividing all of the segments in the solution returned by Algorithm 1 into two sets, $G$ and $H$. The $G$ segments are those that are "contained" entirely within one segment of the optimal solution $F^*$, possibly sharing a start time point. The $H$ segments are those that "cross" between optimal segments, or terminate at the same time point as an optimal segment. The main observation is that the *total cost* of all $G$ segments can be bounded by $\text{Cost}(F^*)$ – this is the case because splitting a segment into multiple sections and finding an optimal linear interpolation for each section can only have smaller error than if all data points in the segment are interpolated using the *same, single* linear function. The cost of $H$ segments is not as easy to bound, but fortunately there are only $k$ of them – exactly one for each of the $k$ optimal segments. Since every segment returned by the greedy algorithm has error $< E$ (or else Algorithm 1 would have started a new segment) we thus have that the $H$ segments contribute error at most $E \cdot k$. This gives the overall error bound in Theorem 2 of $\text{Cost}(\tilde{F}) \leq \text{Cost}(F^*) + Ek$.

The second part of the proof bounds the number of segments returned by the greedy algorithm, which we want to claim is not too large. To do so, we again treat the $G$ and $H$ groups separately. There are $k$ segments in the $H$ group. Bounding the number of $G$ segments is a bit trickier. Roughly, we argue that every consecutive pair of $G$ segments has error $\geq E$ in the optimal global solution, since otherwise the greedy algorithm would not have terminated the first segment in the pair. Since the total cost of the $G$ segments under the optimal global solution is $\text{Cost}(F^*)$, it follows that there can only be at most $O(\text{Cost}(F^*)/E)$ $G$ segments. This leads to a bound on the total number of segments in Theorem 2. We give the formal proof in Appendix A (in the Supplementary Materials).

### 3.4.2. Final Greedy Algorithm

A potential limitation of Algorithm 1 is that, to obtain the provable guarantee of Corollary 3, we need to set $E = \frac{\varepsilon \cdot \text{Cost}(F^*)}{k}$, which is dependent on the optimal cost $\text{Cost}(F^*)$. Of course, since we are not able to actually compute $F^*$, we do not have access to this value, and estimating the value seems potentially as hard as the piece-wise interpolation problem we are trying to solve in the first place.

Fortunately, a simple approach is able to avoid this issue, with only a logarithmic overhead in space and runtime complexity.[§] First, observe that we do not need to set $E$ precisely. For example, if we choose a value slightly too large, a very close approximation guarantee still holds. Specifically, if we set $E = \sigma \cdot \frac{\varepsilon \cdot \text{Cost}(F^*)}{k}$ for some constant $\sigma > 1$, we will get the exact same guarantee on $q$ (the number of segments) as Corollary 3, but with error factor $(1+\sigma\varepsilon)$ instead of $(1+\varepsilon)$. So, even if $\sigma = 2$ (i.e., we set $E$ twice as large as ideal), the increase in error is marginal. On the other hand, if we set $E$ too small, e.g., to $E = \frac{1}{\sigma} \cdot \frac{\varepsilon \cdot \text{Cost}(F^*)}{k}$, we will only get *less error*, although we might select $3\sigma k/\varepsilon$ instead of $3k/\varepsilon$ segments.

This observation motives a simple "gridding" strategy for choosing the right value of $E$, with three key ideas:
**(1)** identify lower and upper bounds $E_{\min}$ and $E_{\max}$ on $E$;
**(2)** *in parallel*, compute solutions for a geometric grid of thresholds between those bounds;
**(3)** combine tasks in **(1)** and **(2)** so that everything is done in one pass, in an online streaming fashion.

We discuss **(2)** first. For a constant $\sigma > 1$ (e.g. 2 or 5), consider integer values $\lfloor \log_\sigma E_{\min} \rfloor, \lfloor \log_\sigma E_{\min} \rfloor + 1, \ldots, \lceil \log_\sigma E_{\max} \rceil$ and compute a solution using Algorithm 1 with thresholds:

$$\bar{E}_{\min} = \sigma^{\lfloor \log_\sigma E_{\min} \rfloor}, \sigma^{\lfloor \log_\sigma E_{\min} \rfloor + 1}, \ldots, \sigma^{\lceil \log_\sigma E_{\max} \rceil} = \bar{E}_{\max}.$$

This list of thresholds only contain at most $O(\log_\sigma(E_{\max}/E_{\min}))$ values but, at the same time, is guaranteed to contain one threshold within a multiplicative factor of $\sigma$ from the *ideal threshold* $E$. Note that $\bar{E}_{\min}$ and $\bar{E}_{\max}$ may not exactly equal to the lower and upper bounds $E_{\min}$ and $E_{\max}$, since we choose integer powers of $\sigma$ for our thresholds, but our range $[\bar{E}_{\min}, \bar{E}_{\max}]$ does cover $[E_{\min}, E_{\max}]$.

Now we discuss **(1)** and **(3)** together. For $E_{\max}$, it suffices to set $E_{\max} = \text{BestLinearFit}([1,\ldots,T], Y)$ – i.e., to the cost of the best 1-piece linear interpolation of the entire dataset. This is because the behavior of the greedy algorithm for $E_{\max} = \text{BestLinearFit}([1,\ldots,T], Y)$ and for *any higher threshold* will be identical: it will only create a single segment. So there is no point in using any thresholds above $\bar{E}_{\max}$. In the streaming setting, $\text{BestLinearFit}([1,\ldots,T], Y)$ can be computed incrementally, as discussed in Sec. 3.2. As more time steps are processed, this value monotonically increases, and so does our current estimated value for $\bar{E}_{\max}$. Naively, this may seem like an issue: whenever $\bar{E}_{\max}$ increases, we would need to restart from the beginning of the dataset to run the basic greedy approach with more thresholds. However,

[§] We can reduce the actual runtime by running in parallel; see below.

there is a trick to avoid restarting. Let $\bar{E}^j_{\max}$ and $\bar{E}^{j+1}_{\max}$ be our current estimates for $\bar{E}_{\max}$ at times $j$ and $j+1$. For any *new* thresholds between $\bar{E}^j_{\max}$ and $\bar{E}^{j+1}_{\max}$ (inclusive) the behavior of the greedy algorithm *up until step* $j$ is identical — again, it only generates *a single segment*. So, we can initialize the greedy algorithm for newly added thresholds above $\bar{E}^j_{\max}$ with a single segment from time steps 1 to $j$, with no need to return to those time steps.

The method for choosing $E_{\min}$ is similar. We process our data in an online way to maintain a monotonically decreasing estimate for $E_{\min}$ with the property that this minimum threshold should be the largest possible that would cause the greedy algorithm to construct what we call a "ZeroErrorSolution" that perfectly interpolates the time-varying volume data. Typically this solution will exactly contain $T/2$ segments, one for each pair of time steps, which can be fit perfectly with a linear function[¶]. Again, there is no point in considering thresholds less than $E_{\min}$, which will all generate identical zero error solutions. Moreover, as our estimate for $E_{\min}$ decreases, we can always initialize the greedy algorithm for new candidate thresholds using the current ZeroErrorSolution, avoiding the need to restart from the beginning of the dataset.

The complete pseudocode for this approach is given in Appendix B (in the Supplementary Materials). There, we moreover discuss an efficient implementation and analyze its runtime and memory footprint. In particular, its total I/O time for reading the input is the same as that of Basic Greedy, which is optimal.

**Remark 1.** As mentioned, compared to Basic Greedy, this method has an additional overhead of $O(\log_\sigma(E_{\max}/E_{\min}))$ in space and runtime, which is small even for $\sigma = 5$ (see experiments in Sec. 4).

**Remark 2.** Final Greedy can be used in two ways: **(A)** In the one-pass in-situ setting, it takes as input a target number of segments $k$. During execution, different threads are used to run parallel copies of Basic Greedy with different thresholds, and any thread is terminated as soon as it accumulates more than $k$ segments. **(B)** If running two passes is possible (e.g., [ZC18]), a first pass run of Final Greedy can be used to record the error and segment start points (time steps $s_1, \cdots, s_k$) for each of the thresholds tried. This produces a "global view", i.e., a plot of error vs. number of segments (e.g., the curve of Final Greedy in Fig. 3(a)), so that the user can choose the *most desirable* error/number of segments. A second pass is then used to generate the full solution (i.e., the interpolation coefficients) corresponding to the computed segment start points.

## 4. Results

We have implemented our methods in C++ and run our experiments under the computing service of our institution's high-performance computing (HPC) cluster, which provided a computing environment with a 24-core 2.90 GHz Intel Xeon Platinum 8268 CPU, 192GB RAM, nVidia Tesla RTX8000 GPU, and Linux Ubuntu OS. Our C++ program calls the ArrayFire [Arr] library for matrix operations, for which the library integrates with CUDA for GPU support. The volume rendering images were produced using the VisIt [CBW*12] package. We used the test datasets listed in Table 1. They are real-world datasets from scientific applications. For all experiments, we do not report the I/O time to output the $k$ segments to disk, since such I/O time depends on the value of $k$, and for all methods compared, such I/O times would be the same for the same output size[‖].

**Table 1:** *Test Datasets.*

| Dataset | Size | Dimensions | TimeSteps | DataType |
|---|---|---|---|---|
| *Isabel* | 4.46 GB | 500x500x100 | 48 | Float |
| *Vortex* | 784 MB | 128x128x128 | 100 | Float |
| *TeraShake* | 23.7 GB | 750x375x100 | 227 | Float |
| *Radiation* | 27.4 GB | 600x248x248 | 200 | Float |
| *Radiation2* | 54.8 GB | 600x248x248 | 400 | Float |
| *Radiation4* | 109.6 GB | 600x248x248 | 800 | Float |
| *Radiation8* | 219.2 GB | 600x248x248 | 1600 | Float |

**Table 2:** *Results for small datasets. N: # grid points at each time step; T: # time steps in the dataset; Runtime: total runtime in* ***seconds****; I/O: I/O time in* ***seconds****; DP: dynamic programming time in* ***milliseconds****; e-time: time to compute* $e(,)$ *in* ***seconds****; Mem: memory footprint; Th: # threads in Final Greedy.*

| Dataset | Method | Runtime | I/O | DP | *e*-time | Mem |
|---|---|---|---|---|---|---|
| *Vortex* ($T$: 100) | AR-DP | 3957 | 5.48 | 1.25 | 3951 | 831MB |
| (784 MB) | Our DP | 164 | 5.72 | 1.38 | 158 | 881MB |
| $N = 2.1$M | Basic Greedy | 14 | 4.26 | N/A | N/A | 50.3MB |
| (Th: 9) | Final Greedy | 17.4 | 4.35 | N/A | N/A | 304MB |
| *Isabel* ($T$: 48) | AR-DP | 5528 | 6.72 | 0.15 | 5821 | 4.8GB |
| (4.46GB) | Our DP | 743 | 6.43 | 0.16 | 731 | 5.4GB |
| $N = 25$M | Basic Greedy | 44 | 6.88 | N/A | N/A | 600MB |
| (Th: 6) | Final Greedy | 86.52 | 6.26 | N/A | N/A | 2480MB |

### In-Core Data: Analysis of Efficiency

Recall that our dynamic programming approach (called **Our DP** here) provides globally optimal solutions to the *general* key time steps selection problem, while the accurate dynamic programming method of [ZC18] gives globally optimal solutions to the *restricted* problem; we call the latter **AR-DP** (denoting *accurate restricted* DP). They are basically in-core algorithms: first read and keep the entire dataset in main memory, then perform computation without the need for additional I/O. They both have two phases: initial computation to compute errors $e(i, j)$, and the phase of DP. We ran **Our DP** and **AR-DP** on the two smaller datasets, *Vortex* and *Isabel*; the results are shown in Table 2. As can be seen, their I/O time and DP time are basically the same, but we are much faster in the initial computation (the $e$-time) due to the online streaming method of Sec. 3.2. Therefore we are also much faster in the total time. In addition, we show the results of running our **Basic Greedy** and **Final Greedy** (with $\sigma = 5$ throughout all experiments) in Table 2. As seen, both our greedy methods are significantly faster than the DP methods. The runtime of Basic Greedy is the fastest, which is linear, and is not affected by the parameter value $E$, as expected. For Final Greedy compared to Basic Greedy, the I/O time is the same, the runtime is less than 2 times, and the memory footprint is about $(2 \cdot Th + 1)/3$ times where $Th$ is the number of threads (which is also $4s \cdot Th + 2s$ where $s$ is the size of one time step in the input), as analyzed in Appendix B (in the Supplementary Materials).

[¶] In rare cases, the ZeroErrorSolution may have segments with $> 2$ time steps, if some volume data does not change over time, or is perfectly colinear.

[‖] For Final Greedy, the threads can write in parallel on parallel disks.

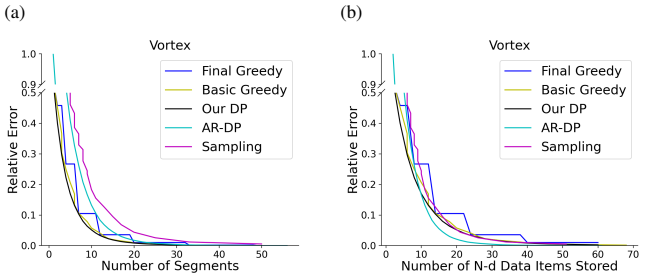


**Figure 3:** *Total error from various methods for the in-core dataset* Vortex*, where the x-axis is (a) the number of segments; (b) the number of N-dimensional data items stored. Relative error is obtained by dividing the (original) error by the maximum of the (original) errors among all methods.*

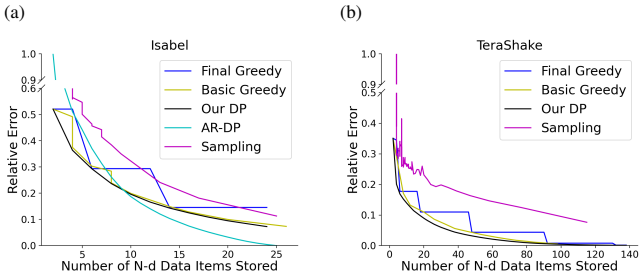


**Figure 4:** *Total error from various methods for (a) the in-core dataset* Isabel*; (b) the larger dataset* TeraShake*.*

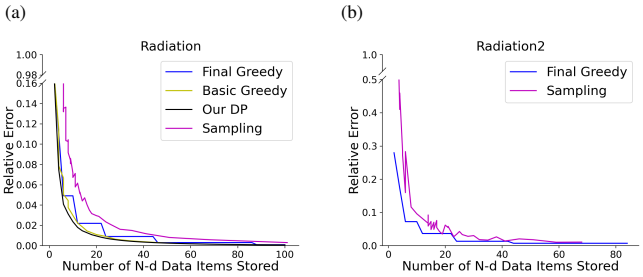


**Figure 5:** *Total error from various methods for (a) the larger dataset* Radiation*; (b) one of the largest datasets* Radiation2*.*

**In-Core Data: Analysis of the Solution Quality**
Next, we compare the solution quality of **AR-DP**, **Our DP**, **Basic Greedy**, **Final Greedy**, and **Sampling**, which just selects every $m$ time steps for a fixed $m$ (and by taking $m = 2, 3, \cdots$, it selects $T/2, T/3, \cdots$ time steps). We remark that **Sampling** is basically the default method in common practice. The results of error vs. number of segments for *Vortex* are shown in Fig. 3(a). As expected, **Our DP** gets the best quality since it is globally optimal for the *general* key time steps selection problem, while **AR-DP** is worse since it is only optimal for the *restricted* problem. We also see that **Basic Greedy** is very close to **Our DP**, and both **Basic Greedy** and **Final Greedy** are actually *better* than **AR-DP**, which is in turn better than **Sampling**. However, note that for $k$ segments, the restricted version only outputs $k+1$ data points (endpoints) while our output is $2k$ interpolation coefficients, equivalent to $2k$ data points in size. To account for this difference for fair comparison, we need to plot error vs. output size; we do so in Fig. 3(b) and all remaining plots. Now we see in Fig. 3(b) that **Our DP** is similar to **AR-DP**, and **Final Greedy** is worse than **Sampling** yet is still comparable. The results for *Isabel* is shown in Fig. 4(a). As seen, **Our DP** is similar to **AR-DP**, and **Basic Greedy** is very close to **Our DP**. More importantly, **Final Greedy** is distinctively better than **Sampling**.

We show the volume rendering of a few time steps reconstructed by **Final Greedy**, **AR-DP**, and **Sampling**; see Fig. 8 in Appendix C (Supplementary Materials). Our results are very close to the ground truth, and have the lowest NRMSE values (defined in Appendix C).

**Larger Data: Analysis of Efficiency**
For larger datasets *TeraShake* and *Radiation*, we ran the out-of-core version of our DP approach (see Sec. 3.3.2), and compare it with both of our greedy methods; the results are shown in Table 3. As can be seen, our greedy methods are significantly faster. We remark that our DP needs to read the data file from disk multiple times. However, since the RAM size was 192GB, the OS was able to cache the data file when it was first read, making the subsequent I/O operations much faster, which also made the overall running time much faster. Even under this effect, our greedy methods are still significantly faster. Therefore the run-time improvement of our greedy methods would be even greater when the data size is really out-of-core. Finally, the relative performances of Final Greedy compared to Basic Greedy are similar to what we saw in Table 2 in terms of the I/O time, total runtime (but now about 3 times), and memory footprint.

**Larger Data: Analysis of Solution Quality**
We compare the solution quality of Our DP, Basic Greedy, Final Greedy, and Sampling, on *TeraShake* and *Radiation*. The results are shown in Figs. 4(b) and 5(a). Similar to what we saw previously, Basic Greedy is very close to the optimal result of our DP, and Final Greedy is not far away, especially at the points where the threads actually ran. In addition, Sampling can perform very poorly (see *TeraShake* in Fig. 4(b)) and thus a much better method is needed to improve the common practice. The volume rendering of the reconstruction results of Final Greedy and Sampling are shown in Fig. 9 in Appendix C (Supplementary Materials).

It is informative to see how our theory predicts in practice for Basic Greedy. To this end, we took the optimal error values obtained from our DP, using $\varepsilon = 0.7$ and different values of the parameter $k$, to get the upper bounds on the total error and on the number of resulting segments as given in Corollary 3. These theoretical bounds are compared against the actual values obtained by running Basic Greedy, shown in Fig. 6. As seen, the upper bounds are conservative. In particular, in practice we perform much better on the number of resulting segments.

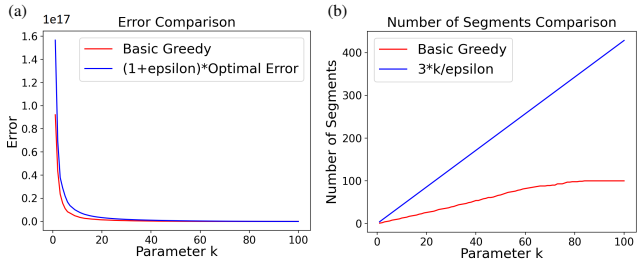


**Figure 6:** *Verification of Corollary 3 using the* Radiation *dataset, with* $\varepsilon = 0.7$. *(a) Actual error obtained vs. the error upper bound calculated using Corollary 3. (b) Actual number of segments obtained vs. the number of segments upper bound calculated using Corollary 3.*

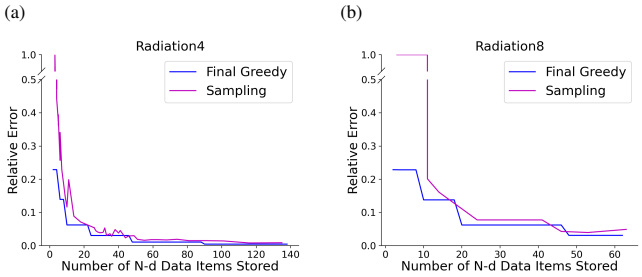


**Figure 7:** *Total error from various methods for the largest datasets: (a)* Radiation4*; (b)* Radiation8.

**Largest Data: Solution Quality and Efficiency**
Finally, we ran our Final Greedy algorithm on the largest datasets *RadiationX* with $X = 2, 4, 8$ (same dimensions as *Radiation* but the number of time steps are $X$ times). The solution quality is compared against Sampling and shown in Figs. 5(b) and 7. Again our Final Greedy is better. The efficiency results of Final Greedy are shown in Table 4. We see that the number of threads (the $\log_\sigma(E_{max}/E_{min})$ factor) is more or less the same, around 10, and the I/O time and total time are both roughly linear in the dataset size. More importantly, it runs very fast: about 2.12 hours for *Radiation8*. As a comparison, the approximate out-of-core method of [ZC18] needed 19.5 hours (though on a different platform, as reported from their paper, not from our experiments). Moreover, our Final Greedy is *online streaming* and is suitable for the *in situ* setting.

**Further Discussions**
Consider running Final Greedy in the *in situ* setting. The I/O time in all tables (for reading the input) would not be needed, but the I/O time for writing the output should be added, which would be $p/T$ times the I/O time in the tables, with $p$ the number of $N$-d data items written and $T$ the number of time steps in the datasets. For the solution quality, Final Greedy always outperforms Sampling in our experiments except for *Vortex* (where we are still comparable), sometimes quite significantly (*TeraShake*), showing that there exists a real-world dataset that needs our new approach with strong theoretical worst-case guarantees to greatly improve over prior methods.

**Table 3:** *Results for larger datasets. R-Time: total runtime; I/O: I/O time; DP: dynamic programming time in* ***milliseconds****; e-time: time to compute* $e(,)$ *in* ***hours****; Mem: memory footprint; Th: # threads in Final Greedy. (*): File was cached in RAM after being read the first time, so subsequent I/O was much faster. This also made R-Time much faster.*

| Dataset | Method | R-Time | I/O | DP | *e*-time | Mem |
|---|---|---|---|---|---|---|
| *TeraShake* | | | | | | |
| (23.7GB, $T$: 227) | Our DP | 6.00h* | 39m* | 12 | 6.00 | 676MB |
| $N = 28$M | Basic Greedy | 3.4m | 0.50m | N/A | N/A | 675MB |
| (Th: 14) | Final Greedy | 11.56m | 0.49m | N/A | N/A | 6223MB |
| *Radiation* | | | | | | |
| (27.4GB, $T$: 200) | Our DP | 6.09h* | 58m* | 11 | 6.09 | 886.5MB |
| $N = 32$M | Basic Greedy | 4.0m | 0.53m | N/A | N/A | 885.7MB |
| (Th: 10) | Final Greedy | 12.46m | 0.54m | N/A | N/A | 5912MB |

## 5. Conclusions

In this paper, we formulated the key time steps selection problem into that of optimal piece-wise linear interpolation. By applying a method in numerical linear algebra, we obtain a key building block of our techniques, an *online streaming* approach for computing linear interpolation solutions and their errors. We then use this building block to obtain a globally optimal solution via dynamic programming, and moreover devise a novel approximate, greedy algorithm for the general key time steps selection problem which is online streaming and suitable for the *in situ* setting. It is very efficient in computing time and main memory space, both in theory and in practice. More importantly, it is the first algorithm suitable for *in situ* key time steps selection with strong theoretical guarantees on the approximation quality and the number of resulting segments. Our experimental analysis shows its superior performance in practice in terms of both efficiency and the solution quality.

**Limitations** Our setting (general version) and error metric (sum of squared errors) follow exactly what were used in the previous in situ work [MLF*16]. That said, unlike the restricted version, our method cannot preserve the original data for any time step. Also, currently our method is limited to the $L_2$ error metric.

**Future Work** While $L_2$ is common, it is also interesting to consider other error metrics like $L_\infty$ or $L_1$. This will be our future work: the challenge is that it is less clear how to solve the resulting non-$L_2$ regression problem in a one-pass streaming fashion. We have found that numerical linear algebra can provide very useful and powerful tools in solving our problems, and is an interesting new direction to explore. Our approach just performs linear approximation in encoding space, then decodes to recover the volumes. We believe that our method should be easy to combine with any non-linear encoding of the volumes given by a neural network in a way similar to [PXvO*19], which would be an interesting future research.

**Table 4:** *Efficiency results of our Final Greedy for the largest datasets. T: # time steps in the datasets; Total: total runtime; I/O: I/O time; Th: # threads; Memory: memory footprint.*

| Dataset | Size | $T$ | Total | I/O | Th | Memory |
|---|---|---|---|---|---|---|
| *Radiation* | 27.4GB | 200 | 12.46m | 0.54m | 10 | 5912MB |
| *Radiation2* | 54.8GB | 400 | 27.68m | 1.04m | 11 | 6475MB |
| *Radiation4* | 109.6GB | 800 | 59.69m | 2.14m | 11 | 6475MB |
| *Radiation8* | 219.2GB | 1600 | 127.37m | 3.81m | 12 | 7039MB |

## References


[ADLS16] ACHARYA J., DIAKONIKOLAS I., LI J., SCHMIDT L.: Fast algorithms for segmented regression. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (2016), p. 2878–2886. 3

[AFM06] AKIBA H., FOUT N., MA K.-L.: Simultaneous classification of time-varying volume data based on the time histogram. In *Proc. IEEE/Eurographics Symp. EuroVis '06* (2006), pp. 171–178. 2

[AHPV04] AGARWAL P. K., HAR-PELED S., VARADARAJAN K. R.: Approximating extent measures of points. *J. ACM 51*, 4 (2004), 606–635. 3

[AM07] AKIBA H., MA K.-L.: A tri-space visualization interface for analyzing time-varying multivariate volume data. In *Proc. IEEE/Eurographics Symp. EuroVis '07* (2007), pp. 115–122. 2

[Arr] ARRAYFIRE:. https://arrayfire.com. 8

[BAA*16] BAUER A., ABBASI H., AHRENS J., CHILDS H., GEVECI B., KLASKY S., MORELAND K., O'LEARY P., VISHWANATH V., WHITLOCK B., BETHEL E.: In situ methods, infrastructures, and applications on high performance computing platforms. *Computer Graphics Forum 3*, 35 (2016), 577–597. 2

[BDM*20] BRAVERMAN V., DRINEAS P., MUSCO C., MUSCO C., UPADHYAY J., WOODRUFF D. P., ZHOU S.: Near optimal linear algebra in the online and sliding window models. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS)* (2020), pp. 517–528. 2, 4

[Bel61] BELLMAN R.: On the approximation of curves by line segments using dynamic programming. *Communications of the ACM 6*, 4 (1961). 3

[BS05] BORDOLOI U., SHEN H.-W.: View selection for volume rendering. In *Proc. IEEE Visualization* (2005), pp. 487–494. 2

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 8

[CFV*16] CHEN M., FEIXAS M., VIOLA I., BARDERA A., SHEN H.-W., SBERT M.: *Information Theory Tools for Visualization*. AK Peters/CRC Press, 2016. 2

[CM10] CORREA C. D., MA K.-L.: Dynamic video narratives. *ACM Transactions on Graphics (Proc. SIGGRAPH '10) 29*, 4 (2010), 88:1–88:9. 2

[FE17] FREY S., ERTL T.: Flow-based temporal selection for interactive volume visualization. *Computer Graphics Forum 36*, 8 (2017), 153–165. 1, 2

[FMHC07] FANG Z., MÖLLER T., HAMARNEH G., CELLER A.: Visualization and exploration of time-varying medical image data sets. In *Proc. Graphics Interface '07* (2007), pp. 281—288. 2

[FRVR14] FELDMAN D., ROSSMAN G., VOLKOV M., RUS D.: Coresets for k-segmentation of streaming data. In *Proc. NIPS* (2014), pp. 559–567. 3

[GIMS20] GEPPERT L., ICKSTADT K., MUNTEANU A., SOHLER C.: Streaming statistical models via merge and reduce. *International Journal of Data Science and Analytics 10* (10 2020). 3

[GLPW16] GHASHAMI M., LIBERTY E., PHILLIPS J. M., WOODRUFF D. P.: Frequent directions: Simple and deterministic matrix sketching. *SIAM J. Comput. 5*, 45 (2016), 1762–1792. 2, 4

[Gum02] GUMHOLD S.: Maximum entropy light source placement. In *Proc. IEEE Visualization* (2002), pp. 275–282. 2

[GW11] GU Y., WANG C.: Transgraph: Hierarchical exploration of transition relationships in time-varying volumetric data. *IEEE Transactions on Visualization and Computer Graphics (Vis'11) 17*, 12 (2011), 2015–2024. 2

[HWG*20] HE W., WANG J., GUO H., WANG K.-C., SHEN H.-W., RAJ M., NASHED Y. S., PETERKA T.: InSituNet: Deep image synthesis for parameter space exploration of ensemble simulations. *IEEE Trans. Vis. Comput. Graph. (TVCG (VIS'19)) 1*, 26 (2020), 23–33. 2

[HXL*11] HU W., XIE N., LI L., ZENG X., MAYBANK S.: A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics-Part C: Applications and Reviews 41*, 6 (2011), 797–819. 2

[KMLC20] KAWAKAMI Y., MARSAGLIA N., LARSEN M., CHILDS H.: Benchmarking in situ triggers via reconstruction error. In *Proc. ACM Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'20)* (2020), p. 38–43. 1, 2, 3

[LHV06] LEE C.-H., HAO X., VARSHNEY A.: Geometry-dependent lighting. *IEEE Transactions on Visualization and Computer Graphics 12*, 2 (2006), 197–207. 2

[LK02] LIU T., KENDER J.: Optimization algorithms for the selection of key frame sequences of variable length. In *Proc. European Conference on Computer Vision-Part IV (ECCV'02)* (2002), pp. 403–417. 2

[LS08] LU A., SHEN H.-W.: Interactive storyboard for overall time-varying data visualization. In *Proc. IEEE Symp. Pacific Visualization* (2008), pp. 143–150. 2

[LS09a] LEE T.-Y., SHEN H.-W.: Visualization and exploration of temporal trend relationships in multivariate time-varying data. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1359–1366. 2

[LS09b] LEE T.-Y., SHEN H.-W.: Visualizing time-varying features with TAC-based distance fields. In *Proc. IEEE Pacific Visualization Symposium* (2009), pp. 1–8. 2

[LSX21] LOKSHTANOV D., SURI S., XUE J.: Efficient algorithms for least square piecewise polynomial regression. In *29th Annual European Symposium on Algorithms (ESA 2021)* (2021), vol. 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 63:1–63:15. 3

[LWM*18] LARSEN M., WOODS A., MARSAGLIA N., BISWAS A., DUTTA S., HARRISON C., CHILDS H.: A flexible system for in situ triggers. In *Proc. ACM Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'18)* (2018), p. 1–6. 3

[MAB*97] MARKS J., ANDALMAN B., BEARDSLEY P. A., FREEMAN W. T., GIBSON S. F., HODGINS J. K., KANG T., MIRTICH B., PFISTER H., RUML W., RYALL K., SEIMS J., SHIEBER S. M.: Design galleries: a general approach to setting parameters for computer graphics and animation. In *Proc. ACM SIGGRAPH* (1997), pp. 389–400. 2

[MLF*16] MYERS K., LAWRENCE E., FUGATE M., BOWEN C. M., TICKNOR L., WOODRING J., WENDELBERGER J., AHRENS J.: Partitioning a large simulation as it runs. *Technometrics 58*, 3 (2016). 2, 3, 10

[PXvO*19] PORTER W. P., XING Y., VON OHLEN B. R., HAN J., WANG C.: A deep learning approach to selecting representative time steps for time-varying multivariate data. In *Proc. IEEE VIS (Short Papers)* (2019), pp. 131–135. 1, 3, 10

[SB06] SOHN B.-S., BAJAJ C.: Time-varying contour topology. *IEEE Transactions on Visualization and Computer Graphics 12*, 1 (2006), 14–25. 2

[SBP*15] SALLOUM M., BENNETT J., PINAR A., BHAGATWALA A., CHEN J.: Enabling adaptive scientific workflows via trigger detection. In *Proc. ACM Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'15)* (2015), p. 41–45. 3

[Str16] STRANG G.: *Introduction to Linear Algebra*, 5th ed. Wellesley-Cambridge Press, Wellesley, MA, 2016. 4

[TFO09] TAKAHASHI S., FUJISHIRO I., OKADA M.: Applying manifold learning to plotting approximate contour trees. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1185–1192. 2

[TLS12] TONG X., LEE T.-Y., SHEN H.-W.: Salient time steps selection from large scale time-varying data sets with dynamic time warping. In *Proc. IEEE Symp. Large Data Analysis and Visualization (LDAV '12)* (2012), pp. 49–56. 1, 3

[WFMF02] WONG K.-P., FENG D., MEIKLE S., FULHAM M.: Segmentation of dynamic pet images using cluster analysis. *IEEE Transactions on Nuclear Science 49*, 1 (2002), 200–207. 2

[WLS13] WEI T.-H., LEE T.-Y., SHEN H.-W.: Evaluating isosurfaces with level-set-based information maps. *Computer Graphics Forum (Special Issue for EuroVis '13) 32*, 3 (2013), 1–10. 2

[WS09a] WOODRING J., SHEN H.-W.: Multiscale time activity data exploration via temporal clustering visualization spreadsheet. *IEEE Transactions on Visualization and Computer Graphics 15*, 1 (2009), 123–137. 2

[WS09b] WOODRING J., SHEN H.-W.: Semi-automatic time-series transfer functions via temporal clustering and sequencing. *Computer Graphics Forum 28*, 3 (2009), 791–798. 2

[WYM08] WANG C., YU H., MA K.-L.: Importance-driven time-varying data visualization. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1547–1554. 2

[XLS10] XU L., LEE T.-Y., SHEN H.-W.: An information-theoretic framework for flow visualization. *IEEE Trans. Vis. Comput. Graph. (Vis '10) 16*, 6 (2010), 1216–1224. 2

[YHSN19] YAMAOKA Y., HAYASHI K., SAKAMOTO N., NONAKA J.: In situ adaptive timestep control and visualization based on the spatio-temporal variations of the simulation results. In *Proc. ACM Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'19)* (2019), pp. 12–16. 3

[ZC18] ZHOU B., CHIANG Y.-J.: Key time steps selection for large-scale time-varying volume datasets using an information-theoretic storyboard. *Comput. Graph. Forum (EuroVis '18) 3*, 37 (2018), 37–49. 1, 2, 3, 5, 6, 8, 10

In this Supplementary Material of appendices, all figure and equation numbers are continued from the paper.

## Appendix A: Formal Proof of Theorem 2

**Theorem 2.** *Let $\tilde{F}$ be the piece-wise linear function returned by Algorithm 1 (represented by $S_{\tilde{F}}$). For any $k$, let $F^*$ be the optimal interpolant with $k$ pieces, defined in Problem 2. Then $\tilde{F}$ has* $\text{Cost}(\tilde{F}) \leq \text{Cost}(F^*) + Ek$ *and at most* $q = k + \frac{2\cdot\text{Cost}(F^*)}{E}$ *segments.*

*Formal Proof of Theorem 2.* In any piece-wise linear interpolation, each segment is determined by its start (time) point, so we will sometimes reference segments by their start points.

Let $1 = s_1^* \leq \ldots \leq s_k^*$ be the start points of the segments in $F^*$ and let $(m_1^*, b_1^*), \ldots, (m_k^*, b_k^*)$ be the corresponding interpolation coefficients. Similarly, let $1 = \tilde{s}_1 \leq \ldots \leq \tilde{s}_q$ be the start points of the segments in $\tilde{F}$ and let $(\tilde{m}_1, \tilde{b}_1), \ldots, (\tilde{m}_q, \tilde{b}_q)$ be the corresponding interpolation coefficients. Note that these coefficient are chosen optimally for each segment (in Line 4 of Algorithm 1).

We split the segments of $\tilde{F}$ into two groups, $G$ and $H$. As discussed before, let $G$ contain any segment that is entirely "contained" within one of the segments of $F^*$, possibly sharing a start time point. Namely, the start point $\tilde{s}_i$ of a $G$ segment must satisfy $s_j^* \leq \tilde{s}_i$ and $\tilde{s}_{i+1} - 1 < s_{j+1}^* - 1$ for some $j \in \{1, \ldots, k\}$. Recall from Eq. (4) that the segment of $F^*$ starting at $s_j^*$ ends at $s_{j+1}^* - 1$, and similarly the segment of $\tilde{F}$ starting at $\tilde{s}_i$ ends at $\tilde{s}_{i+1} - 1$. Let $H$ contains all other segments – specifically, those that "cross" between segments in $F^*$, or share an end time point with a segment in $F^*$. The sets $G$ and $H$ are visualized in Figure 2.

With these definitions in place, we first bound $\text{Cost}(\tilde{F})$. We have:

$$\text{Cost}(\tilde{F}) = \sum_{\tilde{s}_i \in G} \text{Cost}_i(\tilde{F}) + \sum_{\tilde{s}_i \in H} \text{Cost}_i(\tilde{F}), \tag{9}$$

where $\text{Cost}_i(\tilde{F})$ is defined as $\text{Cost}_i(\tilde{F}) = \sum_{j=\tilde{s}_i}^{\tilde{s}_{i+1}-1} \left\|\tilde{m}_i \cdot j + \tilde{b}_i - Y_{j:}\right\|_2^2$. For the first term, we clearly have

$$\left[\sum_{\tilde{s}_i \in G} \text{Cost}_i(\tilde{F})\right] \leq \text{Cost}(F^*). \tag{10}$$

Specifically, consider any segment of $F^*$, $s_j^*$, with interpolation coefficients $(m_j^*, b_j^*)$ and consider the $G$ segments $\tilde{s}_w, \ldots, \tilde{s}_{w+z}$ contained in $s_j^*$. We have that:

$$\begin{aligned}
\sum_{i=w}^{w+z} \text{Cost}_i(\tilde{F}) &= \sum_{i=w}^{w+z} \left[\min_{m,b \in \mathbb{R}^N} \sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}-1} \left\|m \cdot \ell + b - Y_{\ell:}\right\|_2^2\right] \\
&\leq \sum_{i=w}^{w+z} \left[\sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}-1} \left\|m_j^* \cdot \ell + b_j^* - Y_{\ell:}\right\|_2^2\right] \\
&= \sum_{\ell=\tilde{s}_w}^{\tilde{s}_{w+z+1}-1} \left\|m_j^* \cdot \ell + b_j^* - Y_{\ell:}\right\|_2^2 \\
&\leq \sum_{\ell=s_j^*}^{s_{j+1}^*-1} \left\|m_j^* \cdot \ell + b_j^* - Y_{\ell:}\right\|_2^2 = \text{Cost}_j(F^*).
\end{aligned}$$

Summing over all $j$ proves (10). Next, consider the second term of (9). There are $k$ segments in $H$, and each must have cost $< E$, since otherwise it would have been terminated earlier by Algorithm 1. So we conclude that $\sum_{\tilde{s}_i \in H} \text{Cost}_i(\tilde{F}) < Ek$. Combined with (10), this yields the approximation guarantee of Theorem 2.

It remains to prove the upper bound on the number of segments $q$ in $\tilde{F}$. This requires bounding the number of $H$ segments (equal to $k$) and the number of $G$ segments, which is more challenging. To do so, we continue with the notation above: let $s_j^*$ be a segment of $F^*$ and consider the $G$ segments $\tilde{s}_w, \ldots, \tilde{s}_{w+z}$ contained in $s_j^*$. Let $m_j^*, b_j^*$ be the interpolation coefficients for segment $s_j^*$. We claim that for all $i \in \{w, \ldots, w+z\}$:

$$\sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}} \left\|m_j^* \cdot \ell + b_j^* - Y_{\ell:}\right\|_2^2 \geq E. \tag{11}$$

This must be true because, otherwise, $m_j^*$ and $b_j^*$ provide a piecewise interpolation for all $\ell \in [\tilde{s}_i, \tilde{s}_{i+1}]$ with error $< E$, and thus we would have extended segment $\tilde{s}_i$ by at least one more data point when running Algorithm 1. I.e., we would have included $\tilde{s}_{i+1}$ in that segment..

Let $|G|$ denote the number of elements in the set of $G$ segments. For any $\tilde{s}_i \in G$, let $M(i) = j$ if $\tilde{s}_i$ is contained in segment $s_j^*$. Summing (11) over all $j = 1, \ldots, k$ we have:

$$\sum_{\tilde{s}_i \in G} \left[\sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}} \left\|m_{M(i)}^* \cdot \ell + b_{M(i)}^* - Y_{\ell:}\right\|_2^2\right] \geq \sum_{\tilde{s}_i \in G} E = |G| \cdot E \tag{12}$$

At the same time, we have that

$$\begin{aligned}
&\sum_{\tilde{s}_i \in G} \left[\sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}} \left\|m_{M(i)}^* \cdot \ell + b_{M(i)}^* - Y_{\ell:}\right\|_2^2\right] \\
&\leq 2 \sum_{\tilde{s}_i \in G} \left[\sum_{\ell=\tilde{s}_i}^{\tilde{s}_{i+1}-1} \left\|m_{M(i)}^* \cdot \ell + b_{M(i)}^* - Y_{\ell:}\right\|_2^2\right] \leq 2 \cdot \text{Cost}(F^*)
\end{aligned} \tag{13}$$

The first inequality comes from the fact that, for each $\ell$, $\left\|m_{M(i)}^* \cdot \ell + b_{M(i)}^* - Y_{\ell:}\right\|_2^2$ appears in the first equation *at most* twice: specifically, only if $\ell = \tilde{s}_i$ for some $i$. Otherwise it appears once. The second inequality holds because the sum is exactly the cost of $F^*$ restricted to some subset of time points in our dataset – the total cost of $F^*$ can only be higher. Combining (13) and (12), we have:

$$2 \cdot \text{Cost}(F^*) \geq |G| \cdot E$$

and thus $|G| \leq \frac{2\cdot\text{Cost}(F^*)}{E}$. Adding in the $k$ segments in $H$, we conclude that the total number of segments produced by Algorithm 1 is $\leq k + \frac{2\cdot\text{Cost}(F^*)}{E}$. □

## Appendix B: Final Greedy Algorithm

Below we provide formal pseudocode for the final greedy method with automatic parameter tuning, which is outlined in Section 3.4.2. Note that we take as convention that $\max(\{\}) = -\infty$ – i.e., the maximum value of the empty set should evaluate to $-\infty$. We also use an extending function signature for subroutine calls to Algorithm 1, BasicGreedy. In addition to passing in the time steps $t = [1, \ldots, T]$, the data matrix $Y$, and a threshold parameter $E$, we also assume the algorithm is modified to take a 4th parameter $J$ (an integer in $\{1, \ldots, T\}$) and a fifth, $Z$, which itself is a piece-wise linear function (represented as a set of tuples). These parameters indicate

that BasicGreedy should be initialized with starting solution $Z$ and should only start its main loop (line 3 in Algorithm 1) at index $j = J$ — i.e., it should operate exactly as if for the first $J-1$ time steps the algorithm already produced solution $Z$, but has not yet "closed" the last segment in $Z$. Finally, to keep notation concise, in the pseudocode below, we assume that $k$ piece-wise linear functions are specified simply by a list of the starting points of each segment. That representation is what is passed to the BasicGreedy algorithm with extended input signature. In reality, we would also need to pass in the optimal coefficients for each segment, which have already been computed by each call to the BestLinearFit() function.

**Algorithm 2** Final Greedy Algorithm

**Input:** Time steps $[1,\dots,T] = t$, data matrix $Y \in \mathbb{R}^{T\times N}$, resolution $\sigma > 1$

**Initialization:** $\Lambda = \{\}$, $s \leftarrow 1$, ZeroErrorSolution $\leftarrow \{1\}$, $\text{Err}_{min} = \infty$

1: **for** $j = 2,\dots,T$ **do**
2: $\quad \text{Err}_{max} \leftarrow \text{BestLinearFit}\left([1,\dots,j],[Y_{1:};\dots;Y_{j:}]\right)$
3: $\quad$ **if** $\text{Err}_{max} = 0$ **then**
4: $\quad\quad$ **continue**
5: $\quad$ **end if**
6: $\quad \text{Err}_{local} \leftarrow \text{BestLinearFit}\left([s,\dots,j],[Y_{s:};\dots;Y_{j:}]\right)$
7: $\quad$ **if** $\text{Err}_{local} > 0$ **then**
8: $\quad\quad \text{Err}_{min} \leftarrow \min(\text{Err}_{local}, \text{Err}_{min})$
9: $\quad\quad$ ZeroErrorSolution $\leftarrow$ ZeroErrorSolution $\cup \{j\}$
10: $\quad\quad s \leftarrow j$
11: $\quad$ **end if**
12: $\quad \Lambda_{new} = \{\lfloor \log_\sigma(\text{Err}_{min})\rfloor, \dots, \lceil \log_\sigma(\text{Err}_{max})\rceil\}$
13: $\quad$ **for** $L \in \Lambda_{new} \setminus \Lambda$ **do**
14: $\quad\quad$ **if** $L > \max(\Lambda)$ **then**
15: $\quad\quad\quad$ BasicGreedy$(t, Y, \sigma^L, j+1, \{1\})$
16: $\quad\quad$ **else if** $L > \log_\sigma(\text{Err}_{min})$ **then**
17: $\quad\quad\quad Z =$ ZeroErrorSolution $\setminus \{j\}$
18: $\quad\quad\quad$ BasicGreedy$(t, Y, \sigma^L, j+1, Z)$
19: $\quad\quad$ **else**
20: $\quad\quad\quad$ BasicGreedy$(t, Y, \sigma^L, j+1,$ ZeroErrorSolution$)$
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: $\quad \Lambda \leftarrow \Lambda_{new}$
24: **end for**

**Efficient Implementation**

Naively we may let each thread perform the tasks of **Basic Greedy**, including reading input from disk. However, this would cause I/O contention and thus a major slowdown if the threads share a single disk. Even on parallel disks, to achieve parallel disk reads, we would typically need to partition the input into subgroups and copy/move these subgroups among the disks, which is nontrivial and has an additional I/O overhead.

To address this issue, we observe that all threads actually read the *same* time-step data from disk, which is not really necessary — the key idea is that we can just let *one* thread perform the data reading, and *share* the data read among the threads.

In our implementation, we have three types of threads: (1) one *reading* thread that reads the input from disk, (2) one *main* thread that computes $E_{max}$ and $E_{min}$ (and their corresponding solutions) and also creates new threads if needed, and (3) additional threads that run **Basic Greedy** with thresholds in the range $(E_{min}, E_{max})$ (exclusive). In each iteration, the reading thread reads one time step from disk, which is shared among all threads (a shared variable) when the reading is done. Then all threads of types (2) and (3) proceed with their own computation on the current time step. When all such threads are done, we move on to the next iteration for the next time step. In this way, the total I/O time for reading the input is essentially the same as that of **Basic Greedy**, which is optimal.

**Analysis of Runtime and Memory Footprint**

Now we analyze the running time of our implementation. Let $\tau$ be the time of **Basic Greedy** to compute on a single time step (i.e., *not* including the disk reading time). In each iteration, the main thread would take about $2\tau$ time to finish its computation; within the same $2\tau$ time, all existing threads of type (3) would also finish the computation. If the main thread does not create any new threads (of type (3)), then this iteration is done. Otherwise, the newly created threads would take an additional $\tau$ time to finish. Therefore, the computing time for each iteration is about $2\tau$ to $3\tau$, and thus the total computing time of our implementation of **Final Greedy** is about 2-3 times of that of **Basic Greedy**.

Next we analyze the memory footprint of our implementation. We first consider **Basic Greedy**. From Sec. 3.2, computing the optimal error $\text{Err}^*$ (see Eq. (5)) only depends on the memory size of $A'Y$, which is a $2\times N$ matrix. To compute the optimal solution $Z^* = (A'A)^{-1}A'Y$ (see Eq. (3)) in a streaming way, we need to update $A'Y$ by multiplying the new column of $A'$ with the new row of $Y$ (which is the new time step read, an $N$-tuple $R$), and adding the result to the original $A'Y$ (a $2\times N$ matrix). To do so, let $[a\ \ b]'$ be the new column of $A'$, and $P$ and $Q$ be the first and second rows of the original $A'Y$. Then we can update $P$ to $P+aR$ as follows: each time we multiply $a$ with an element of $R$ we immediately add the result to the corresponding element of $P$; we update $Q$ to $Q+bR$ in the same way. Therefore, we can update $A'Y$ without extra working space, using $2N$ scalar values for $A'Y$ and $N$ scalar values for the new time step, in a total of $3N$ scalar values. In order to achieve more accurate results in matrix operations, we use data type *double* for all the scalar values involved, so the memory footprint $M$ of **Basic Greedy** is $M = 3N$ doubles. Let $s$ be the size of one time step in the input ($N$ scalar values). If the input data type is double, then $s$ corresponds to $N$ doubles and $M$ is $3s$. If the input data type is float (as in our experiments), then $M = 6s$.

Now we analyze the memory footprint $M'$ of **Final Greedy**. Let $Th$ be the total number of threads. The reading thread (type (1)) uses memory size of $N$ doubles for the current time step. The main thread (type (2)) computes $E_{max}$ and $E_{min}$ (and their corresponding solutions), each using the memory size of $A'Y$ ($2N$ doubles), with a total of $4N$ doubles. For each of the remaining $(Th-2)$ threads (type (3)), the memory size of $A'Y$ ($2N$ doubles) is used. The total memory footprint is thus $M' = N+4N+(Th-2)\cdot 2N = 2N\cdot Th+N$ doubles. Recall that the memory footprint of **Basic Greedy** is $M = 3N$ doubles, and thus we have $M'/M = (2\cdot Th+1)/3$. To express $M'$ in terms of $s$, we have $M' = 2s\cdot Th+s$ if the input data type is double, and $M' = 4s\cdot Th+2s$ if the input data type is float.

## Appendix C: Images Rendered from Reconstruction

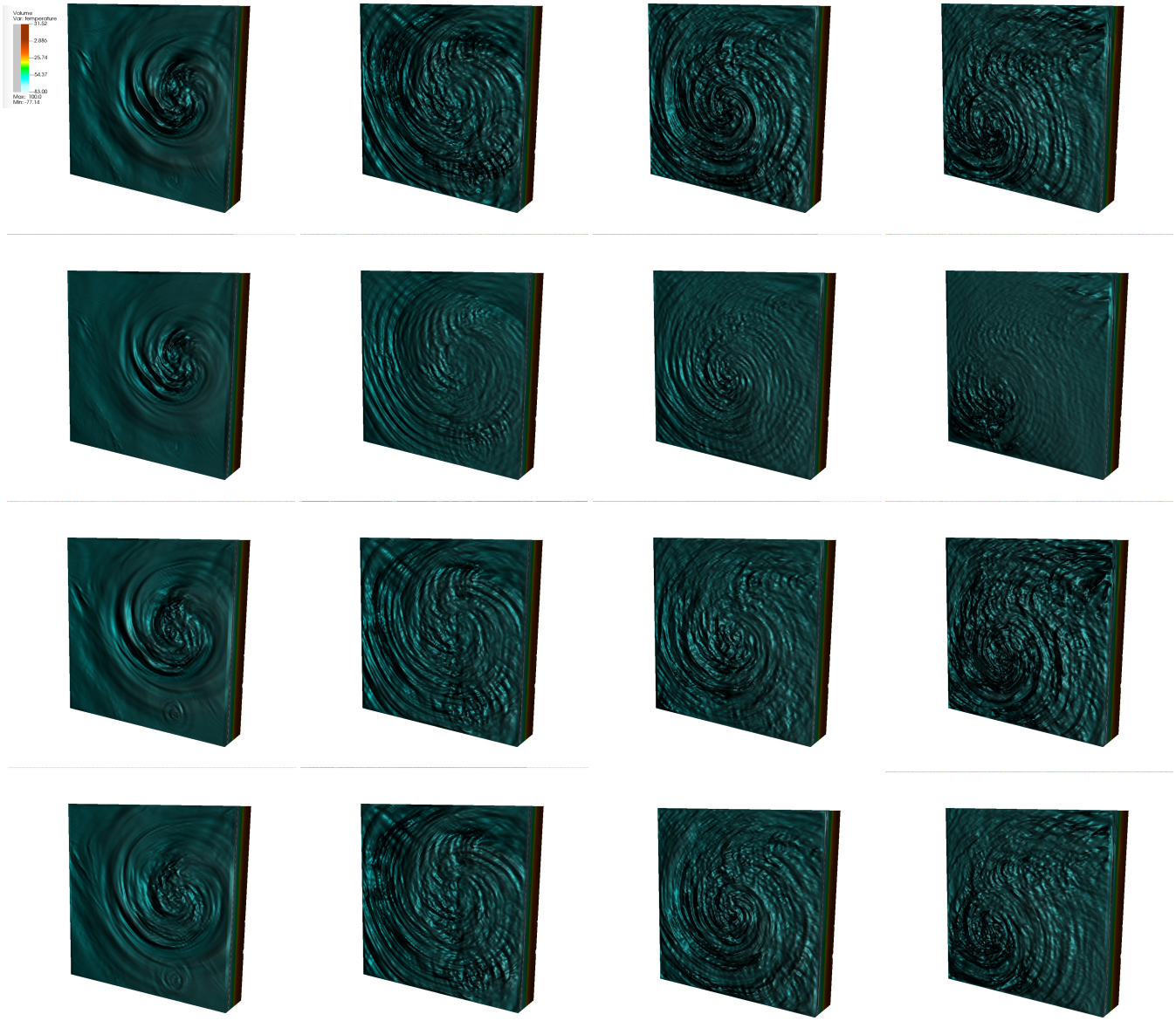


**Figure 8:** *Volume rendering of the reconstruction results of the* Isabel *dataset by* ***Final Greedy***, ***AR-DP***, *and* ***Sampling***, *with the number of N-d data items stored* $= 14$ *(and the total relative errors (see Fig. 4(a)) 0.1456, 0.0888 and 0.2411 respectively), at time steps 6, 14, 28, 44 from left to right. Top row: original data; second row:* ***Final Greedy****; third row:* ***AR-DP****; bottom row:* ***Sampling****. The normalized root-mean-square error (NRMSE) values of the images from top down: left column (0, 4.86%, 8.72%, 8.73%), middle-left column (0, 9.74%, 12.47%, 12.61%), middle-right column (0, 9.75%, 12.09%, 12.43%), right column (0, 8.57%, 12.32%, 10.80%). NRMSE is defined as* $\frac{\sqrt{\sum_{i,j}\sum_{k=1}^{3}(A[i,j,k]-B[i,j,k])^2}}{\sqrt{\sum_{i,j}\sum_{k=1}^{3}A[i,j,k]^2}}$, *where A is the ground truth and B the image of reconstruction, and the R, G, B values of the* $(i, j)$ *pixel of the image A are accessed by* $A[i, j, k]$ *for* $k = 1, 2, 3$.

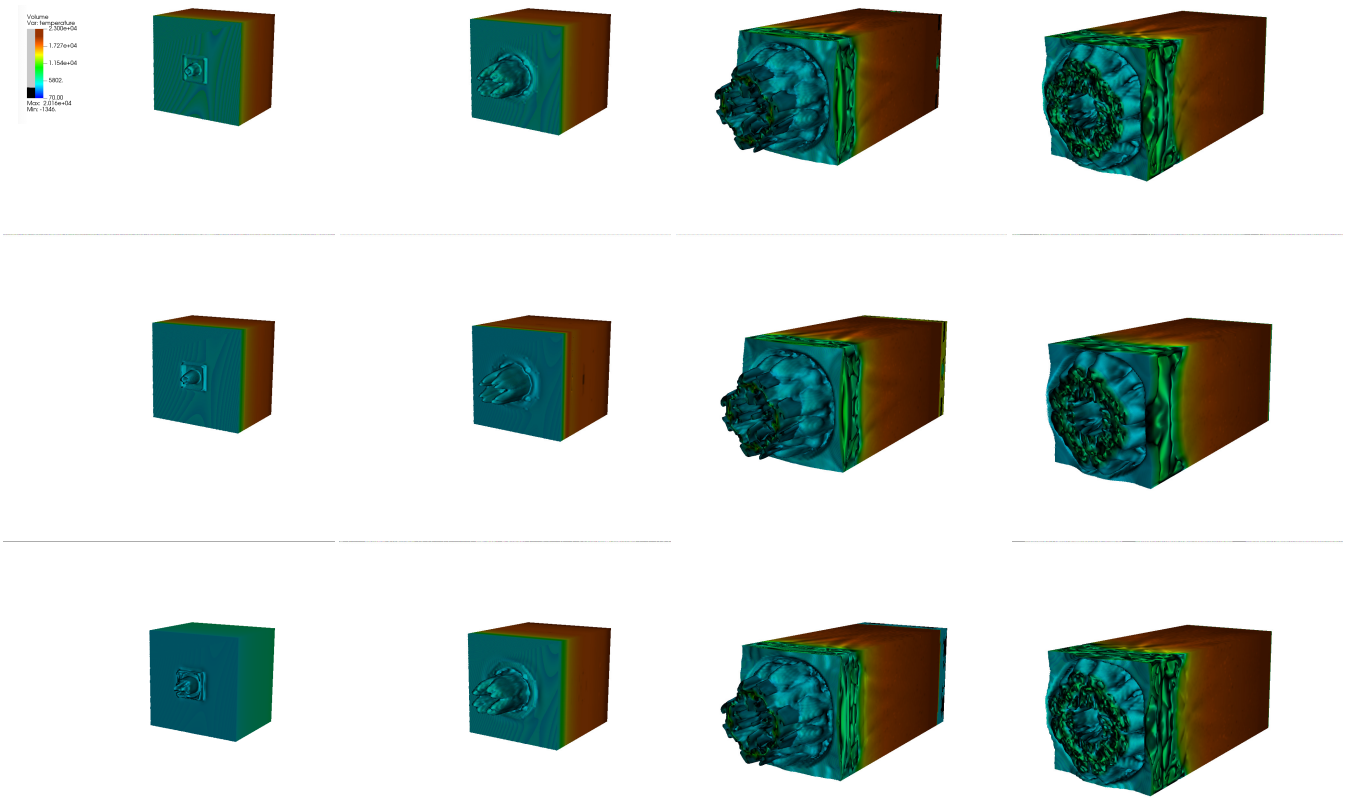

**Figure 9:** *Volume rendering of the reconstruction results of the* Radiation *dataset by* ***Final Greedy*** *and* ***Sampling****, with the number of N-d data items stored* = 24 *(and the total relative errors (see Fig. 5(a)) 0.0091 and 0.0235 respectively), at time steps 7, 48, 151, 186 from left to right. Top row: original data; middle row:* ***Final Greedy****; bottom row:* ***Sampling****. The NRMSE values of the images from top down: left column (0, 3.85%, 9.01%), middle-left column (0, 4.51%, 5.72%), middle-right column (0, 2.61%, 7.44%), right column (0, 7.57%, 7.96%).*