Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

Using Graphs for Fast Error Term Approximation of Time-varying Datasets

C. Nuber,[1], E. C. LaMar[2], V. Pascucci [2], B. Hamann[1] and K. I. Joy[1]

[1] Center for Image Processing and Integrated Computing
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562, U.S.A.
[2] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550-9234, U.S.A.

Abstract

We present a method for the efficient computation and storage of approximations of error tables used for error estimation of a region between different time steps in time-varying datasets. The error between two time steps is defined as the distance between the data of these time steps. Error tables are used to look up the error between different time steps of a time-varying dataset, especially when run time error computation is expensive. However, even the generation of error tables itself can be expensive. For n time steps, the exact error look-up table (which stores the error values for all pairs of time steps in a matrix) has a memory complexity and pre-processing time complexity of $O(n^2)$, and $O(1)$ for error retrieval.
Our approximate error look-up table approach uses trees, where the leaf nodes represent original time steps, and interior nodes contain an average (or best-representative) of the children nodes. The error computed on an edge of a tree describes the distance between the two nodes on that edge. Evaluating the error between two different time steps requires traversing a path between the two leaf nodes, and accumulating the errors on the traversed edges. For n time steps, this scheme has a memory complexity and pre-processing time complexity of $O(n\log(n))$, a significant improvement over the exact scheme; the error retrieval complexity is $O(\log(n))$. As we do not need to calculate all possible n^2 error terms, our approach is a fast way to generate the approximation.

Categories and Subject Descriptors (according to ACM CCS): E.2 [Data Storage Representations]: Object representation, G.1.0 [General]: Error analysis, I.4.2 [Compression (Coding)]: Approximate methods, I.4.8 [Scene Analysis]: Time-varying imagery

1. Introduction

The increase in computation power and storage capacity over recent years has led to a substantial, multiple-order-of-magnitude increase in size of datasets generated by simulations. Advancing technology allows us to perform simulation processes with increasing speed, leading to both an increase in the number of time steps as well as the size of the data generated for a single time step.

Due to the size of the datasets, the data must be reduced to a size where it can be interactively accessed, retrieved and visualized. For steady, time-independent datasets, techniques such as subdivision, multi-resolution, view-dependent rendering and adaptive refinement have already been successfully applied.

Considering time varying datasets, techniques based upon adaptive and incremental updates and key-framing help to reduce the amount of data to be loaded. Techniques based solely on adaptive and incremental updates are mostly based on pre-defined error bounds. Key-framing allows a user to start at an arbitrary key-frame, proceeding to the next, with only one direction to go. None of the above techniques provide the support of arbitrary error-bounds, arbitrary navigation in time and arbitrary resolution; they also provide only a limited notion of re-using data from previous time steps, especially for arbitrary navigation in time.

When scientists need to control and know the error in the data displayed, techniques not providing methods for explicit error control are inappropriate. Spatio-temporal reuse[9] with multi-resolution datasets is a solution to the problem, providing bidirectional navigation using arbitrary error bounds. It reuses parts of a dataset over longer periods of sudden changes in time, replacing or refining parts only when a given error criterion is no longer satisfied. This approach is based on pre-calculated errors using an error table for every region in a multi-resolution dataset.

Error look-up tables for each region in a multi-resolution dataset are an easy means for retrieving errors between different time steps without touching the data. These tables are independent of the underlying data, which can be, for example, an image, a volume or an isosurface. As error calculation can be very expensive, error look-up tables (or simply error tables) help to avoid these costs during run time by pre-calculating errors in advance for later retrieval. The information provided by error-tables can also be used for replacement-optimization for adaptive, view-dependent, multi-resolution replacement strategies.

A naive approach to represent these error tables uses $n \times n$-matrices for each region, where each pair of time steps is evaluated and stored within a matrix ($e(t_0,t_1) := e[t_0][t_1]$). This approach gives us an exact error, as every pair of time steps is stored within the matrix. With a memory complexity of $O(n^2)$ for n time steps, both the size of this error table and the number of error-calculations to set up the matrix can become very large. Reducing memory requirement and computation time for error tables, while maintaining accuracy at acceptable error access and retrieval times, is a necessary task to allow one to apply error tables to simulations with larger number of time steps. For an error-based multi-resolution display of time-varying data as described by Nuber et al.[9], error-tables are a solution for fast error-retrieval. With an increasing number of time steps these tables can become very large, so that for a large enough number of time steps n the size of the error-tables becomes larger than the size of the dataset that is displayed, rendering simple tables for spatio-temporal reuse with larger n unusable.

We solve the memory problem using trees: Leaf-nodes correspond to time steps of the dataset, while internal nodes contain approximations of their children – an edge between two nodes describes the error between the two nodes. To retrieve the error between two given time steps, we calculate an approximated error by traversing the tree from one leaf-node to the other, accumulating the errors of the edges traversed. By using trees, we achieve a significant reduction with respect to memory, at the same time reducing the number of necessary evaluations of error terms to set up the tree. A tree with a fixed bifurcation factor k has a memory and initialization complexity of $O(n \log_k(n))$, and a retrieval complexity of $O(log_k(n))$. We used trees with both fixed and adaptive bifurcation factors. When using incremental error-lists, where $e(t_i,t_j) = \sum_{k=i}^{j-1} e(t_k,t_{k+1})$, $i < j$, memory and initialization complexity are both $O(n)$, but retrieval complexity is also $O(n)$.

To reduce the evaluation error, introduced by adding error terms during traversal, and improve the approximation quality, we use the concept of tunneling during traversal, navigating not only between parent and child nodes but also between node neighbors on the same tree level.

2. Related Work

One area of related work is the visualization of time-varying data. Finkelstein et al.[3] used a multi-resolution image technique and a quad-tree to decompose an image. In their approach, each node contains the averaged color of the children, and time was encoded using a time-spanning binary tree for each subregion. Other techniques include *Time-Space Partitioning* by Shen et al.[12], decomposing space using an octree, and decomposing time in each node using a binary tree, or *THIT* by Shen[11], which is a temporal hierarchical index tree for accelerated isosurface extraction, based upon the use of temporal subdividing binary trees. Ma et al.[7] used voxel-level quantization, octree encoding and differential encoding for compression on voxel, spatial and temporal dimensions, achieving compression ratios up to 90%. Nuber et al.[9] investigated the spatio-temporal reuse of data in a multi-resolution time-varying datasets using error tables to determine whether re-use, re-load or refinement is necessary. By relating the error directly with time and evaluating error at run time their method supports arbitrary navigation and interactive re-definition of the error during run time.

Error tables can also be regarded as height-field data, defined on a regular grid. These height-fields, which can easily be represented as regular triangulated surfaces, can be approximated using mesh simplification methods[10, 14]. (An overview was given by Cignoni et al.[2]) Mesh simplification methods generate a (sometimes multi-resolution) representation of the underlying triangulation. An area related to height-field simplification and reduction in this context is work done on surface approximation[4, 5, 13] and subdivision surfaces with wavelets[1, 6]. Although mesh simplification and surface methods can produce very good results, these techniques have the common drawback that additionally to the time needed for data approximation, all error terms must be evaluated a priori for an appropriate approximation.

3. Approach

Our approach is based on the idea of approximating error tables for time-varying datasets describing the error (or

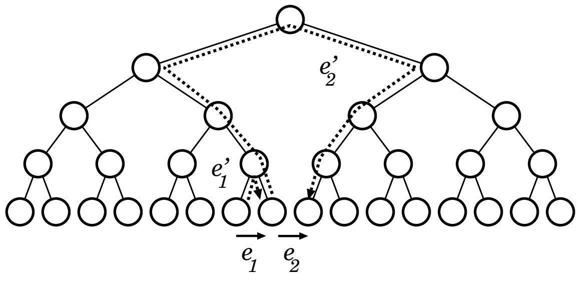


Figure 1: *Influence of location in branches for tree-based error evaluation between successive time steps: accurate for same parent ($e'_1 \approx e_1$), not accurate for different branches ($e'_2 \gg e_2$)*

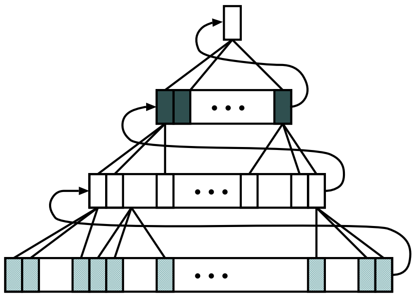


Figure 2: *Array-based tree organization*

difference) between arbitrary time steps using tree structures. We use each of the nodes in the tree to represent or approximate a time step, where leaf nodes represent the time steps t_i and internal nodes represent either averages or best-representatives of their children. The error between two nodes connected by an edge is associated with the edge itself. The error between a parent and a child node is stored with the child node.

When calculating the error for two arbitrary time steps, we traverse the tree from one node to the other, starting at the leaf-node representing the smaller time step, and accumulating the error terms of all the edges we traverse until we reach the larger time step. This approach requires the underlying error norm $e(t_i, t_j) = |dist(t_j, t_i)|$ to satisfy $|dist(t_i, t_k)| + |dist(t_k, t_j)| \geq |dist(t_i, t_j)|$.

As long as changing the time restricts motion to one branch within the tree (switching to a node that has the same parent as the current node) the results should be rather close to the real values. If the time steps are located in different branches, moving from one branch to another introduces most likely a large error term into the calculation. Even when the leaves representing the time steps are located next to each other in time, the retrieved error can differ remarkably from the real error, depending on the number of parental nodes that need to be traversed to get from one node to the other. This means that the approximated error for two succeeding time steps within the same branch would be fairly accurate, while the error to the next time step in a different branch can become very large, as edges to nodes representing a larger time-span need to be traversed (Figure 1), even if the error between both pairs is the same.

Assuming that only errors between close datasets are retrieved, a simple list with incremental errors between successive time steps would be a solution, simply accumulating all error terms between two time steps. For small steps the result would be quite accurate, whereas for larger steps, even with small changes in the dataset over time, the estimated error term and the real error would diverge with an increasing number of time steps.

While lists give us better approximations for errors between close time steps, trees give us better approximations for distant time steps. We use trees with internal shortcuts, combining both approaches. Shortcuts are additional edges, connecting nodes within the graph on the same level. We can use "tunneling" when moving from one branch to the next, reducing the number of edges to traverse and thus minimizing the resulting error term. Tunneling is the process of moving from one branch in the tree to another branch between nodes using shortcuts, without traversing a common ancestor.

3.1. Tree Representation

As the tree representation should be as compact as possible, the overhead introduced by the tree data structure should be as small as possible. We have restricted ourselves to trees with all leaves on the same level. An index-based tree representation was used, storing each node within a large static array (Figure 2). Generally, there are two possible approaches to store trees: *(i)* a fully implicit tree structure, where the tree is fully expanded and the parent/child indices can be derived from the index of the node itself (at least for an n-ary tree), and *(ii)* a partially implicit structure, where only the number of nodes required is used and additional information is stored to maintain parent-child relations. We use a partially implicit structure, as the fully implicit tree structure for an n-ary tree induces large memory overhead (especially for $n > 2$) when the number of time steps (and thus number of leaves) is slightly larger than a potential (n^k), with k being the height of the tree. The overhead can be n times as large as the memory needed when the number of time steps equals $(n^k + 1)$. An example is shown in Figure 3.

In a partially implicit structure, we only need the following information to reconstruct the internal structure of the tree:

- number of leaf nodes and
- number of children per non-leaf node.

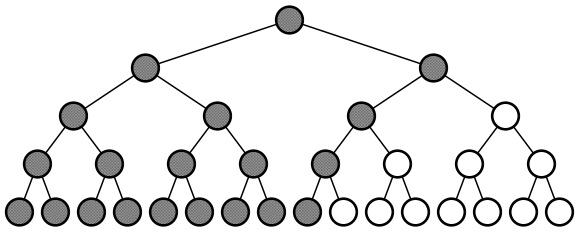


Figure 3: *Memory requirement for fully implicit (all nodes) and partially implicit structure (gray nodes) of a binary tree*

The number of levels and the number of nodes per level can be derived from the given values when reconstructing the tree structure in a bottom-up manner. For navigation within the tree, to traverse the tree from parent to children and vice versa, we additionally need

- the index of the parent node within the parent level and
- the index of the first child node within the next level.

This tree-organization fits our method of construction of and navigation in the tree between neighboring nodes – both operations are solely based on index arithmetics.

3.2. Constructing the Tree

We generate the tree in a bottom-up approach, starting at the leaves and combining several nodes under a new parental node. This process is done for each level, until the number of parental nodes generated equals one, which is the root. Each edge connecting a child to its parent contains the error term between the child and the parent. By using a bottom-up approach the array containing the tree can be extended without the need to move branches within the array. The difference between *n*-ary and adaptive trees is the fact that the number of nodes for a parent node must be determined when initializing an adaptive tree. In the following, we first describe how we construct *n*-ary trees and the adaptive tree, before introducing additional error terms stored with the nodes. These additional terms are used later during the error term approximation between two arbitrary time steps.

3.2.1. Selecting a Dataset Represented by the Parental Node

Generally, there are two ways to select a dataset for an inner node: either one of the child nodes is used to represent the parent, or an average of all the child nodes is used for the parental node. The advantage of using the average is that the average error between parent and children will be smaller when using averaged data. The advantage of selecting a child node is that the error approximation can be improved during tree traversal; the error term can be reset to zero when encountering the initial time step, or the traversal can be terminated earlier when the target time step is encountered during traversal. In both cases, the number of edges contributing to the error term is reduced, improving the final error approximation (see section 4.4 for details).

3.2.2. *n*-ary Trees

n-ary trees have the advantage that the number of nodes required can be derived immediately from *n* and the number of leaf nodes (or time steps), as well as the number of nodes in the upper and lower levels. As the number of edges is also known prior to error term calculation, the number of operations to calculate the error terms is known.

After selecting an approximation for the parent node, the error to all the child nodes needs to be calculated and is stored with the child nodes, representing the error term for the connecting edge.

3.2.3. Adaptive Trees

Adaptive trees are more complicated to generate than *n*-ary trees. Given an initial error tolerance, nodes are assigned to a common parent as long as the error criterion is fulfilled (e.g., maximum error between two arbitrary nodes or sum of error increments between consecutive time steps). All nodes are then grouped under a new parental node, an approximation is chosen, and the error terms evaluated. The advantage of adaptive trees is the fact that the bifurcation factor depends on the error: larger numbers of leaves can be grouped under a parent node if the change over time is small, and smaller groups if the change over time is large.

3.2.4. Additional Error Terms

Together with the error term between parent and child node, we also store additional error terms:

- **intra-level error terms:** error between direct neighbors on the same level, and
- **inter-level error terms:** error between a node and the right neighbor of the parent

The intra-level error terms are the basis for tunneling between branches (see section 3.3.2). By exploiting the array structure, the right neighbor of a node with index i can be accessed via the index $i+1$, as long as the node is not the last on the level; the left neighbor can be found similarly using index $i-1$, if the node is not the first one on the level. As the error terms are symmetric, $e(t_i,t_j) = e(t_j,t_i)$, we need to store only the error to the right neighbor; the error to the left neighbor is the error of the left neighbor to the current node. Inter-level error terms, when applicable, further reduce the number of edges to traverse during error calculation for larger time steps (see section 3.3.3).

3.3. Evaluation of Error Terms

During tree traversal we add up the error terms represented by the traversed edges within the graph. We have considered

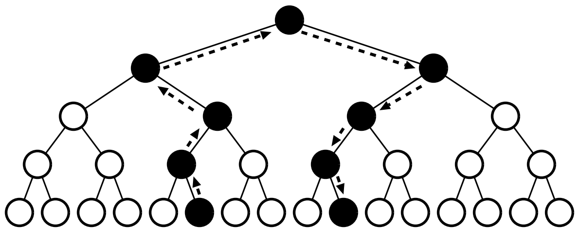


Figure 4: *Edge-based error calculation by following tree edges only*

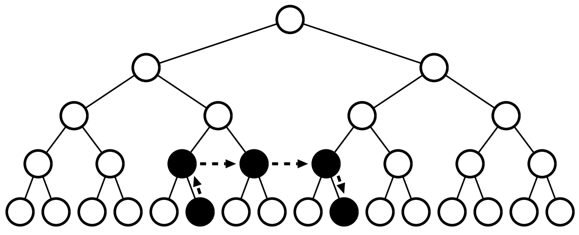


Figure 5: *Intra-level error calculation by following tree edges and intra-level edges*

three different approaches to evaluate an error term within the tree:

- **edge-based approximation:** traversing only the edges found in the underlying tree structure
- **intra-level approximation:** traversing the edges found in the underlying tree structure plus edges between nodes on the same level
- **inter-level approximation:** traversing the edges found in the underlying tree structure plus edges between nodes on the same level and edges between nodes on consecutive levels

We were especially interested in the relationship between evaluation mode and approximation error. (See Table 2 and Figure 8).

3.3.1. Edge-based Error Term Approximation

When performing a "edge-based" approximation, we only use the edges found in the original tree structure (Figure 4). The results achieved are fairly accurate, as long as the nodes are not too far away from each other or located in a hierarchy of different branches.

3.3.2. Intra-level Error Term Approximation

During intra-level approximation, we also use the intra-level error terms between neighboring nodes (Figure 5). This approach allows us to perform, for example, tunneling for configurations where the edge-based approximation might generate poor results. Using intra-level approximation, the number of error terms to be stored is doubled when compared to a tree only suited for edge-based approximation.

We decide which edge to follow primarily based on the number of edges that need to be traversed. For two nodes on level k with a common ancestor on level $k+l$, the upper bound of edges to be traversed is $2*l$, which is the number of edges to be traversed during edge-based approximation. We generally use the intra-level error terms when $2*l > index(a) - index(b)$. In some cases, better results can be achieved by traversing differently, but detecting these cases is expensive, as all possible paths need to be traversed in order to find the optimal one.

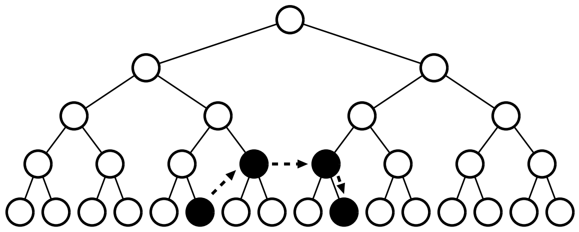


Figure 6: *Inter-level error calculation by following tree edges, inter- and intra-level edges*

Due to array organization, the distance between two nodes on the same level can be determined by the difference in their indices. The decision whether to move within the same level or go one level further up can be reduced to a comparison of index difference and distance to the common ancestor.

We also use enforced tunneling when the left node is the right most under its parent (going to the right) or the right node is the left most under its parent (going to the left), which implies tunneling the parent in both cases. We can therefore move between different subtrees, possibly reducing the maximum height to traverse by eliminating a high level ancestor.

3.3.3. Inter-level Error Term Approximation

Inter-level approximation is an improvement over intra-level approximation. It increases the number of error terms per node by a factor of 1.5, when compared to intra-level approximation, or a factor of three, when compared to edge-based approximation. The difference between intra-level and inter-level approximation is that we also store inter-level error terms between a node and the right neighbor of its parent, further reducing the number of edges to be traversed. The maximum reduction of number of edges corresponds to the height of the highest node visited during traversal, replacing a "up-and-to-the-right step" by a single step (Figure 6).

3.4. Storage Scheme

In case of out-of-core applications, where tables must be retrieved from disk, we need an efficient way to load and initialize the tables. Storing and retrieving error trees is fairly easy due to the underlying array structure. As all nodes are stored consecutively in the array, we can simply write the arrays containing the error term information to disk. Besides the error terms themselves, we also need to store the number of time steps. If we have an n-ary tree, we only need to store the number of time steps, k. The numbers of nodes per level are given by

$$\begin{aligned} nodes(level_0) &= k \quad \text{and} \\ nodes(level_{l+1}) &= \left\lceil \frac{nodes(level_l)}{n} \right\rceil . \end{aligned}$$

Similarly, the index of the ith child for the jth node at level l can be calculated as

$$\begin{aligned} child_i(j) &= index(parent) - j - nodes(level_{l-1}) \\ &\quad + j * n + i \\ &= index(parent) - nodes(level_{l-1}) \\ &\quad + j * (n-1) + i. \end{aligned}$$

Concerning adaptive trees, we need to store the number of children for non-leaf nodes. If we store the number of children for all non-leaf nodes in the order they are found during traversal of the array, we can restore the tree structure following this algorithm:

```
nodesTraversed = 0;
nextParentIndex = numberOfTimesteps;
while ( nextParentIndex < numTreeEntries ) {
    numberOfNodes = next number of nodes under parent;
    for ( i=0; i < numberOfNodes; i++ )
        parent(nodesTraversed+i) = nextParentIndex;
    nodesTraversed += numberOfNodes;
    nextParentIndex++;
}
```

Starting at the first leaf node, we set the parent index of the following number of nodes to the same value, which is initialized as the number of time steps, and increase the parent index after wards, until the index of the next parent equals the number of nodes in the tree. To restore a tree from disk, we need to load the error terms and reconstruct the internal structure in a single pass.

4. Results

We have applied our method to a simulated Richtmyer-Meshkov dataset[8] (Figure 7). The simulation describes the process of two fluids mixing when a shock wave has passed through them. We have extracted one data slice from the volumetric dataset for each of the 274 time steps. The images in Figure 7 show a cross section through the volumetric dataset parallel to the direction of the shock wave. The error between two time steps is given by adding the absolute differences for all pixels, and subsequently normalizing the difference to a value between zero and one. (A value of one corresponds to the maximum possible difference for a single pixel multiplied by the number of pixels per dataset.)

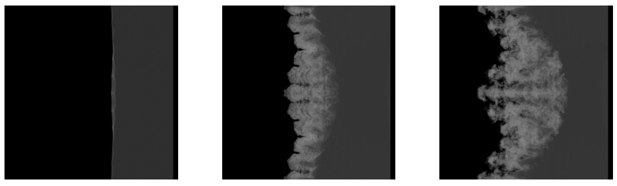


Figure 7: *Cross-section of three time steps of the Richtmyer-Meshkov simulation*

Method	Number of Entries/Percentage		
#Children	Edge-based	Intra-level	Inter-level
squared		37675/100%	
adapt. (R)	707 / 1.9%	1414 / 3.8%	2121 / 5.6%
adapt. (A)	655 / 1.7%	1310 / 3.5%	1965 / 5.2%
2	553 / 1.5%	1106 / 2.9%	1659 / 4.4%
3	415 / 1.1%	830 / 2.2%	1245 / 3.3%
4	369 / 1.0%	738 / 2.0%	1107 / 2.9%
5	355 / 0.9%	710 / 1.9%	1065 / 2.8%
10	306 / 0.8%	612 / 1.6%	918 / 2.4%

Table 1: *Number of entries for 274 time steps of a Richtmyer-Meshkov simulation. Shown are the numbers for all three modes and the percentage values of error terms when compared to the exact error table.*

4.1. Memory Requirements

Table 1 shows the numbers achieved when applying our method to the 274-time-step Richtmyer-Meshkov dataset. This table lists the number of error terms stored for a look-up matrix (first row), and the number of nodes required for the different trees and the different evaluation modes. For intra-level evaluation, *i.e.*, when using the adaptive tree with representatives, our method requires 1414 error terms, or 3.8% when compared to the exact look-up matrix (37673 entries). The maximum error between the look-up matrix and the adaptive tree is $1.01 * 10^{-4}$, with a maximum value of $4.74 * 10^{-4}$ in the look-up matrix itself (Table 2). Thus, the number of entries for the trees is much smaller than the number of entries for a matrix.

4.2. Quality of Approximation

The quality of the approximation depends on the dataset, the evaluation scheme, and the error tree used. Inter-level and intra-level evaluation generate nearly identical results, with inter-level results being slightly better. Table 2 shows a

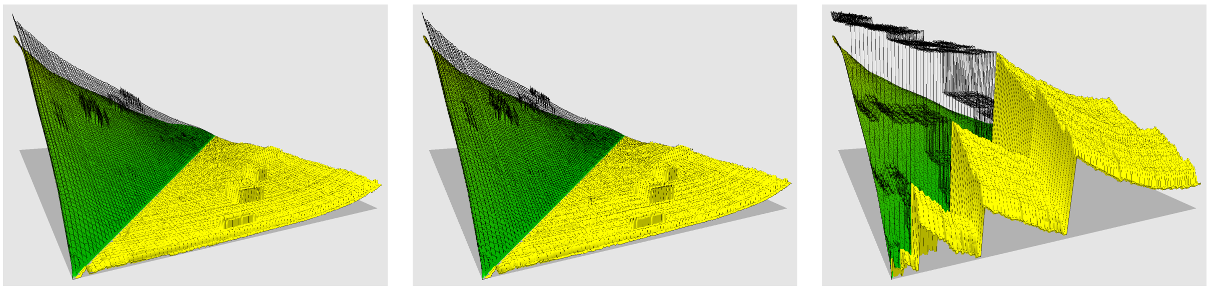


Figure 8: *Surfaces showing exact two-dimensional error table (green), adaptive tree (wire frame, using original datasets for internal nodes) as a height field $e(t_i,t_j)$, and difference between both tables (yellow): inter-level evaluation (left), intra-level evaluation (middle), edge-based evaluation (right).*

Method	$\max(e_{appr.} - e)$	$\overline{e_{appr.} - e}$	$\sigma(e_{appr.} - e)$
	adaptive, representatives		
edge-b.	$4.57 * 10^{-4}$	$1.67 * 10^{-4}$	$2.81 * 10^{-2}$
intra-l.	$1.01 * 10^{-4}$	$1.81 * 10^{-5}$	$5.28 * 10^{-3}$
inter-l.	$9.56 * 10^{-5}$	$1.67 * 10^{-5}$	$5.07 * 10^{-3}$
	binary, representatives		
edge-b.	$4.75 * 10^{-4}$	$1.82 * 10^{-4}$	$3.06 * 10^{-2}$
intra-l.	$9.92 * 10^{-5}$	$1.86 * 10^{-5}$	$5.63 * 10^{-3}$
inter-l.	$9.36 * 10^{-5}$	$1.68 * 10^{-5}$	$5.29 * 10^{-3}$

Table 2: *Maximum difference (max), average difference ($\overline{e}$) and variance (σ) of an adaptive and a binary tree using representatives with respect to the exact error table (max. value found in error table being $e = 4.74 * 10^{-4}$). Images for the adaptive tree are shown in Figure 8.*

comparison of all evaluation methods for an adaptive and a binary tree. Images of the corresponding error-surfaces for the adaptive tree are shown in Figure 8. For non-edge-based evaluation, the approximation is very good from very small to relatively large differences in time, when comparing the difference surface to the exact height field. Noticeable differences are visible only for very large time differences, which is acceptable. For edge-based evaluation, especially for smaller differences in time, the resulting error can be very large.

Figure 9 shows the behavior of the approximation scheme for an adaptive and a binary tree for the Richtmyer-Meshkov simulation slices, both trees using inter-level evaluation. It can be seen that the adaptive approach generates the best error values for small to reasonable large time steps. With the binary tree, we obtain more peaks at a regular distance, especially for smaller time steps. Usually the n-ary trees are smaller than the adaptive trees, but the adaptive trees pro-

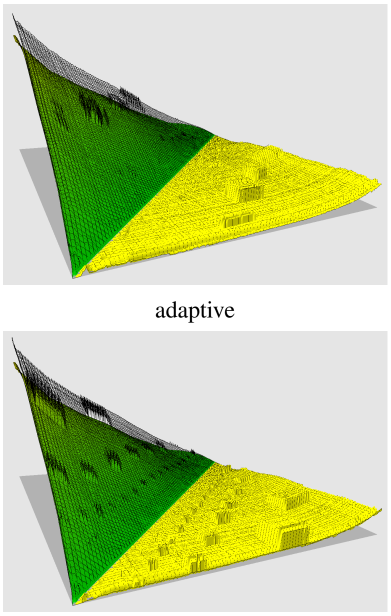


Figure 9: *Surfaces showing exact two-dimensional error table (green), tree (wire frame, using original datasets for internal nodes) as a height field $e(t_i,t_j)$, and difference between both tables (yellow): adaptive tree (top), binary tree (bottom).*

vide better approximation quality. A balance must be found between memory available, evaluation time, and error tolerance.

When using n-ary trees, it can be observed that the approximated error table shows a noticeable pattern caused by the underlying tree structure (Figure 10). While the adaptive tree shows no regular pattern, the binary tree shows regular occurrences of small peaks at an equal distance from the di-

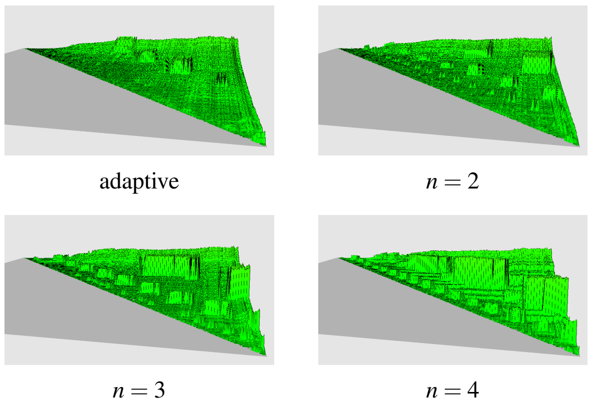


Figure 10: *Pattern on error height field caused by n-ary trees. Images show adaptive tree and trees with bifurcation-factor $n = 2,3,4$.*

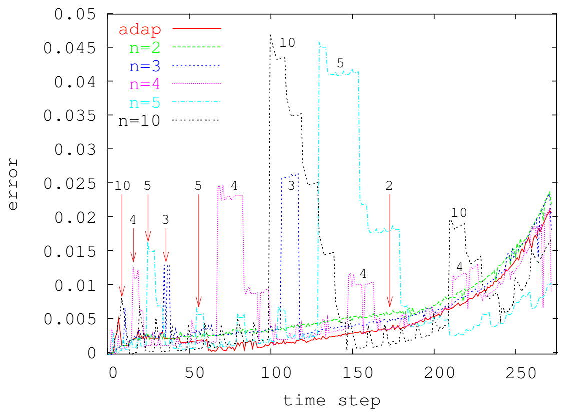


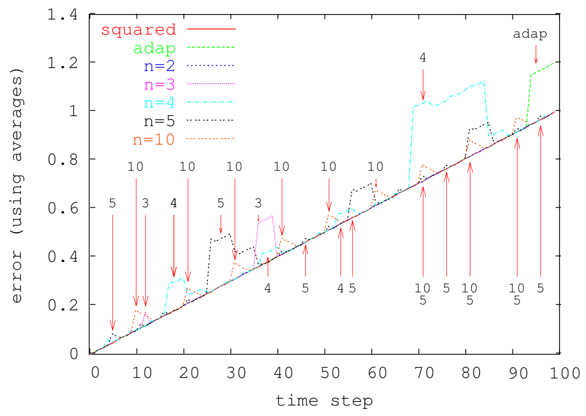
Figure 11: *Approximation errors between t_0 and t_i for different trees (adaptive: red, 2-ary: green, 3-ary: dark blue, 4-ary: pink, 5-ary: light blue, 10-ary: black)*

agonal. It can be seen that for n-ary trees, for a growing number of n, the peaks move closer to each other and the diagonal, which represents the error terms for immediate neighbors, decreasing in number but increasing in height. This observation shows that the accuracy decreases for a subset of time step combinations. With increasing n an internal node spans a larger time span, resulting in an increase of the average error between a set of child nodes and their parents, as more time steps are represented. Trees with larger n should thus not be used for datasets with continuous changes over time.

The pattern becomes even more visible when looking at the evolution of the introduced error with advancing time t, starting with the first time step t_0 (Figure 11). The frequency of peaks in the introduced error relates to n for n-ary trees. For larger n the difference becomes smaller, especially for larger t_i, whereas the average error and the variance become larger due to the peaks.

To verify these observations, we generated a synthetic dataset consisting of 100 unicolored equidistant time steps, with shades of gray from black to white. Figure 12 shows the error terms obtained when calculating $e(t_0,t_i)$, $i = 0,1,..,99$, both for average and representative trees. The difference between two time steps t_i and t_j (or the error $e(t_i,t_j)$ when using t_j to represent t_i) is $|i-j|/100$.

Both figures show that the number n of children in regular trees causes an pattern on the error approximation, independent of the way we choose the datasets for the internal nodes. The uniform characteristic of the dataset is the reason for this fact, as each representative has a maximum distance from the average equal to half a time step. The deviation increases dramatically at points where the error evaluation moves from $e(t_0,t_{k-1})$ to $e(t_0,t_k)$, $k \bmod n = 0$, showing that our tunneling approach helps suppressing the first peak one would find when advancing from $e(t_0,t_{n-1})$ to $e(t_0,t_n)$.



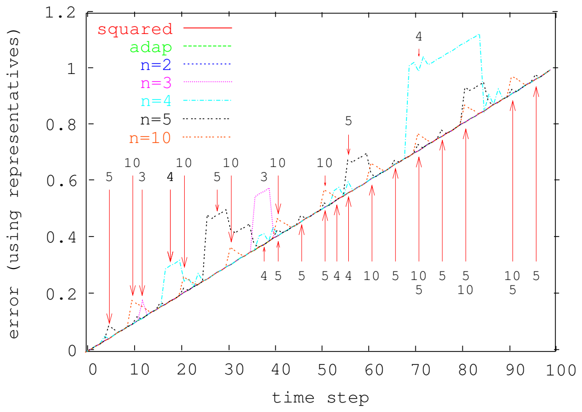


Figure 12: *Error terms shown for a 100-step unicolor dataset with shades from black to white, $e(t_i,t_{i+1}) = const$ (upper: trees using averages, lower: trees using representants); correct error $e(t_0,t_i) = i/100$ only provided by squared table.*

Further suppressing depends on the nature of the underlying dataset. One can also see that the adaptive approach follows the exact error table, with a small deviation for the average tree. As the pattern seems to affect more error pairs for larger n, very small values for n (two or three) should be chosen when no adaptive tree is used.

4.3. Timing Results

Using trees instead of a simple table has an impact on retrieval time during run time, as a simple look-up is replaced by index- and error-calculations. We have measured the evaluation-time for the simulated Richtmyer-Meshkov dataset with 274 time steps, resulting in 37675 possible pairs. As can be seen in Table 3, a simple table is most efficient when evaluating all pairs. For a tree the evaluation time increases with decreasing bifurcation factor. The evaluation time is the time needed to evaluate all 37675 time-step pairs. For smaller time differences, evaluation time for a single pair is less than for larger time differences, when more nodes need to be traversed.

Tree	Evaluation-Mode		
	Edge-based	Intra-level	Inter-level
squared		0.01s	
adapt	0.062s	0.1s	0.1s
2	0.048s	0.084s	0.081s
3	0.036s	0.062s	0.06s
4	0.031s	0.055s	0.05s
5	0.03s	0.05s	0.047s
10	0.025s	0.042s	0.038s

Table 3: *Evaluation-time for all 37675 time-step pairs. Timings show error retrieval when table/tree is completely in memory.*

4.4. Comparing the Use of Averaged and Original Time Steps for Internal Nodes

When an internal node represents an original dataset, the number of error terms to be accumulated during traversal can be reduced when the internal node represents either the start or the end time step of the error term to be evaluated, improving the accuracy of our approximation.

When nodes do not represent single time steps, the dataset needs either to be generated on the fly (being expensive and counterproductive) or loaded (requiring additional datasets to be stored, increasing the overall size of the dataset used). Especially for datasets with large single time steps, like multi-resolution volumetric datasets, the additional storage space or bandwidth might not be available, making the use of time step representatives the better or necessary choice. By inspecting the difference surfaces shown in Figure 13, it can be seen that the use of representatives can lead to ridges in the surface, marking more accurate error terms within higher-error areas due to shorter traversal paths as the representative of the starting dataset has been encountered. Depending on the properties of the underlying dataset and the storage-space available, using representatives instead of averages can be the better choice.

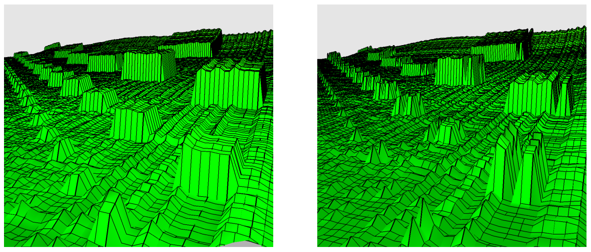

$n = 2$, average $n = 2$, representative

Figure 13: *Comparison between average and representative based trees ($n = 2$, close-up of error surface shown).*

5. Conclusions and Future Work

We have described a method extending error trees for the approximation of error tables, resulting in comparable error terms for smaller time steps and acceptable error terms for larger time steps with a reduced number of error computations for approximation generation. Deviations from the exact error with this technique for larger time steps are acceptable, as the probability for a region being replaced is higher for larger than for smaller time steps. The approach can be applied to datasets of arbitrary dimensions. Especially for multi-resolution approaches, approximated error tables can be used to estimate the error between arbitrary time steps without the necessity of touching the data, which makes the method also independent of the size of the underlying dataset. Especially for larger number of time steps the amount of memory and computation time required for a single error tree is significantly smaller than for an error matrix, so that additional computation time can be neglected, as page faults will occur less often.

We plan to investigate further the use of error trees for table approximation, especially in the context of optimization for adaptive error tree calculation, which is the most expensive type of tree to construct. Another topic of interest is to integrate error trees into existing frameworks for time-varying multi-resolution datasets, which use error tables. We expect that comparisons between a multi-resolution approximation based upon error matrices to be of the same quality as well approximating error trees.

Acknowledgments

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the National Institute of Mental Health and the National Science Foundation under contract NIMH 2 P20 MH60975-06A2; and the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159. We thank the members of the Data Science Group at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory and the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

References

1. Martin Bertram, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. In Thomas Ertl, Bernd Hamann, and Armitabh Varshney, editors, *Proceedings IEEE Visualization 2000*, pages 389–396, 2000.

2. Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A comparison of mesh simplification algorithms. *Computers and Graphics*, 22(1):37–54, 1998.

3. Adam Finkelstein, Charles E. Jacobs, and David H. Salesin. Multiresolution video. In *Proceedings of SIGGRAPH 1996, Computer Graphics*, volume 30, pages 281–290, New York, 1996. ACM Press.

4. Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. *Computer Graphics*, 28(Annual Conference Series):295–302, 1994.

5. Seungyong Lee, George Wolberg, and Sung Yong Shin. Scattered data interpolation with multilevel B-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):228–244, 1997.

6. Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Transactions on Graphics*, 16(1):34–73, 1997.

7. Kwan-Liu Ma, Diann Smith, Yun Ming-Shih, and Han-Wei Shen. Efficient encoding and rendering of time-varying volume data. Technical Report TR-98-22, Institute for Computer Applications in Science and Engineering, June 1998.

8. Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, Willian P. Dannevik, Andris M. Dimits, Mark A. Duchaeineau, Don E. Eliason, Daniel R. Schikore, Sarah E. Anderson, David H. Porter, Paul R. Woodward, L. J. Shieh, and Steve W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Proceedings of the ACM/IEEE SC99 Conference*. IEEE Computer Society, November 13–19 1999.

9. Christof Nuber, Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. Approximating time-varying multiresolution data using error-based temporal-spatial reuse. In K. Boerner, P.C. Chen P.C., R.F. Erbacher, M. Groehn, and J.C. Roberts, editors, *Proceedings of Electronic Imaging 2003*, volume 5009, Bellingham, Washington, January 2003. SPIE - The International Society for Optical Engineering.

10. William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

11. Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings IEEE Visualization '98*, pages 159–166. IEEE, Computer Society Press, October 18–23 1998.

12. Han-Wei Shen, Ling-Jan Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings IEEE Visualization '99*, pages 371–378. IEEE, Computer Society Press, October 25–29 1999.

13. Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 23rd ACM national conference*, pages 517–524, 1968.

14. Jason D. Walter and Christopher G. Healey. Attribute preserving dataset simplification. In Kenneth I. Joy, Robert J. Moorhead, and Amitabh Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 113–120. IEEE, 2001.

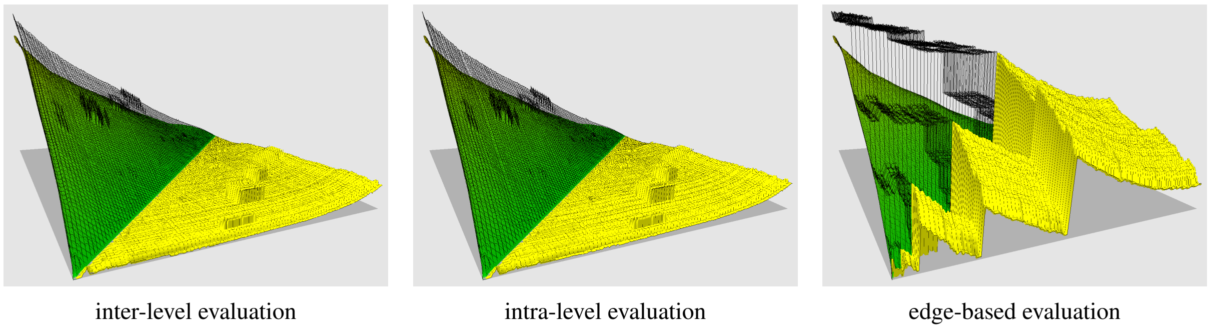


Figure 9: *Surfaces showing exact two-dimensional error table (green), adaptive tree (wire frame, using original datasets for internal nodes) as a height field $e(t_i, t_j)$, and difference between both tables (yellow): inter-level evaluation (left), intra-level evaluation (middle), edge-based evaluation (right).*

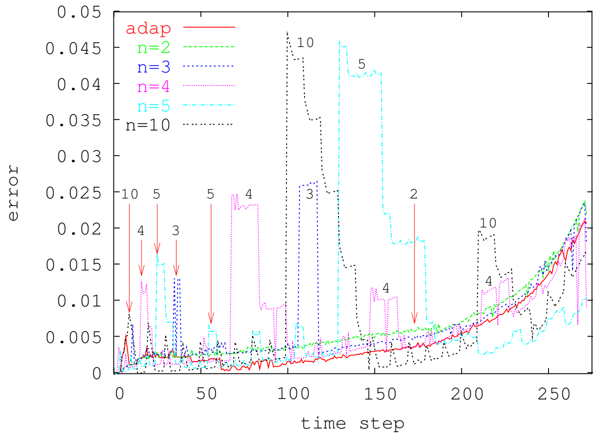


Figure 12: *Approximation errors between t_0 and t_i for different trees (adaptive: red, 2-ary: green, 3-ary: dark blue, 4-ary: pink, 5-ary: light blue, 10-ary: black)*

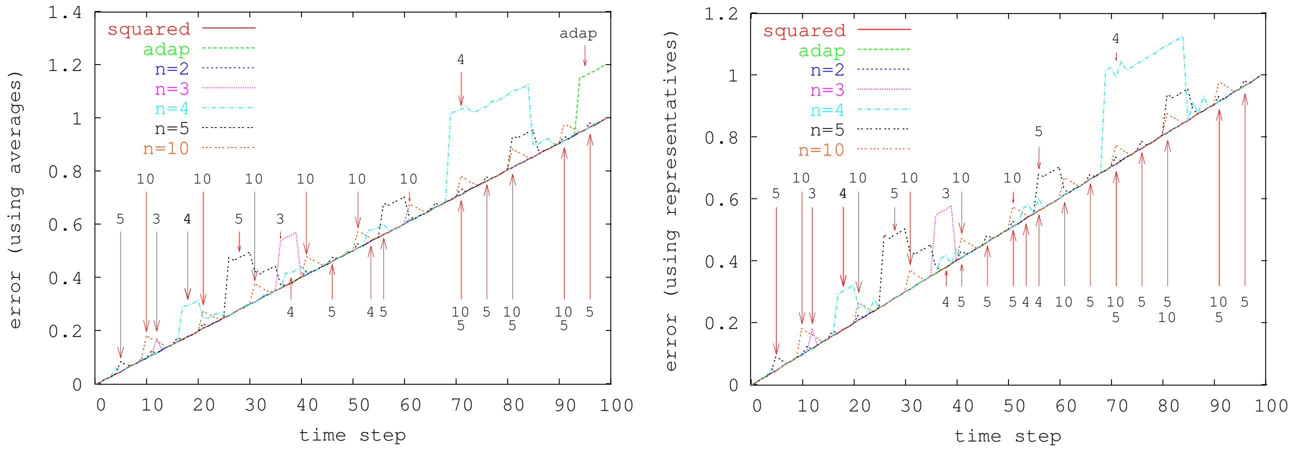


Figure 13: *Error terms shown for a 100-step unicolor dataset with shades from black to white, $e(t_i, t_{i+1}) = const$ (upper: trees using averages, lower: trees using representants); correct error $e(t_0, t_i) = i/100$ only provided by squared table.*