Isosurface Extraction in Time-varying Fields Using a Temporal Hierarchical Index Tree

Han-Wei Shen*
MRJ Technology Solutions / NASA Ames Research Center

Abstract

Many high-performance isosurface extraction algorithms have been proposed in the past several years as a result of intensive research efforts. When applying these algorithms to large-scale time-varying fields, the storage overhead incurred from storing the search index often becomes overwhelming. This paper proposes an algorithm for locating isosurface cells in time-varying fields. We devise a new data structure, called Temporal Hierarchical Index Tree, which utilizes the temporal coherence that exists in a time-varying field and adaptively coalesces the cells' extreme values over time; the resulting extreme values are then used to create the isosurface cell search index. For a typical time-varying scalar data set, not only does this temporal hierarchical index tree require much less storage space, but also the amount of I/O required to access the indices from the disk at different time steps is substantially reduced. We illustrate the utility and speed of our algorithm with data from several large-scale time-varying CFD simulations. Our algorithm can achieve more than 80% of disk-space savings when compared with the existing techniques, while the isosurface extraction time is nearly optimal.

Keywords: scalar field visualization, volume visualization, isosurface extraction, time-varying fields, marching cubes, span space.

1 Introduction

An isosurface represents regions that have a constant value in a three-dimensional scalar field. Displaying isosurfaces is a useful technique for analyzing scalar data due to its effectiveness in revealing the spatial structures of the field's value distribution. To compute the isosurface, Lorensen and Cline [1] proposed a Marching Cubes algorithm which extracts small polygon patches from individual cells in the field. The Marching Cubes algorithm is simple and robust. However, the process of linear search for isosurface cells can be expensive. To improve the performance, researchers have proposed various schemes that can accelerate the search process. Examples include Wilhelm and Van Gelder's Octrees[2], Livnat *et al.* 's NOISE method[3], Shen *et al.* 's ISSUE algorithm[4], Itoh and Koyamada's Extrema Graph method [5, 6], Bajaj *et al.* 's Fast Isocontouring method [7, 8], and Cignoni *et al.* 's Interval Tree[9] algorithm.

Inevitably, these acceleration algorithms incur overhead for storing extra search indices. For a steady scalar field, i.e., only a single time step of data is present, this extra space is often affordable, and the highly interactive speed of extracting isosurfaces can compensate for the overhead. However, for time-varying simulations, a typical solution can contain a large number of time steps, and every simulation step can produce a great amount of data. The overall storage requirement for the search index structures can be overwhelming. Furthermore, when analyzing a time-varying scalar field, a user may want to explore the data back and forth in time, with the same or different isovalues. This will require a significant amount of disk I/O for accessing the indices for data at different time steps when there is not enough memory space for the entire time sequence. As a result, the performance gain from the efficient isosurface extraction algorithm could be offset by the I/O overhead.

This paper presents an efficient isosurface extraction algorithm for time-varying scalar fields. The main focus is to devise a new search index structure for a time-varying field so that the storage overhead is kept small, while the performance of the isosurface extraction is still high. In addition, our algorithm allows flexible control of the tradeoff between performance and storage space and, thus, can be used for data with different characteristics in different computing environments. To achieve these goals, we characterize each cell in the field based on its extreme values and the variation of the extreme values over time. Consider a cell that has a high temporal coherence and, thus, a small scalar variation over several time steps. Such a cell, in a period of several time steps, may be referenced by a single index entry based on that cell's overall extreme values in time. On the other hand, for a cell that has little coherence and, thus, a high scalar variation, the cell is indexed individually at every time step by its corresponding extreme values. Our algorithm creates an isosurface cell search index for the time-varying field, called *Temporal Hierarchical Index Tree*. Cells that have a small amount of variation over time are placed in a single node of the tree that covers the entire time span. Cells with a larger variation are placed in multiple nodes of the tree multiple times, each for a short time span. When generating an isosurface, a simple traversal will retrieve the set of nodes that contains all of the cell index entries needed for a given time step. The cells are organized at each node using a data structure that was developed for generating isosurfaces from a steady data set. For a typical time-varying scalar field, not only does this temporal hierarchical index tree require much less storage space, but also the amount of I/O required to access the indices at different time steps from the secondary storage is greatly reduced.

We begin this paper by giving an overview of the isosurface extraction problem and some existing techniques. We then present our algorithm on building the temporal hierarchical index tree and the isosurface extraction method for time-varying fields. Finally, we present experimental results to demonstrate the effectiveness of our algorithm and provide concluding remarks and future research plans.

2 Background and Related Work

Given an isovalue, cells that have minimum value lower, and maximum value higher, than the isovalue are intersected by the isosurface. We call these cells isosurface cells. To expedite the isosurface cell search process, researchers have proposed various techniques for creating search indices by partitioning the cells based on their spatial and/or value information. An example of the space-partition methods is Wilhelm and Van Gelder's octrees algorithm [2], which partitions the data hierarchically and coalesces the extreme values,

*NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035 (hwshen@nas.nasa.gov)

i.e., the minimum and maximum values, of cells within each local cluster. The octrees algorithm is primarily for structured grid data. The efficiency of the method is reported to be $O(k+log(n/k))$ [3], where k is the number of isosurface cells, and n is the total number of cells.

There are many value-partition methods proposed in the past years [10, 11, 12, 3, 4, 9]. Among those methods, Livnat *et al.* [3] proposed *span space*, a two-dimensional space where every cell in the field is represented by a point. The point's x coordinate represents the corresponding cell's minimum value, and the y coordinate represents the cell's maximum value. Livnat *et al.* use a Kd-Tree, and subsequently Shen *et al.* [4] use a lattice subdivision, to subdivide the cells in span space based on their value ranges. Cignoni *et al.* [9] proposed the use of an *interval tree* as the search index, which has an optimal efficiency of $O(log(n))$. Recently, Chiang and Silva [13] proposed I/O optimal techniques to build the interval tree on disk, and the access of the interval tree is driven by demand. Chiang, Silva, and Schroeder also expanded the I/O-optimal techniques for out-of-core isosurface extraction [14].

In addition to the space- and value-partition methods, Itoh *et al.* [5, 6] and Bajaj *et al.* [7, 8] proposed algorithms using a surface propagation scheme. In their methods, a small set of *seed* cells is first extracted; and isosurfaces of any given isovalue can then be computed by propagating surfaces from certain seeds through adjacencies. Bajaj *et al.* 's algorithm is able to create only a small number of seeds and has an optimal efficiency of $O(log(n))$.

The acceleration algorithms described above inevitably incur overhead for storing extra search indices. For instance, the BON octrees proposed in [2] increase the original data by 16%, which is the ratio of the number of tree nodes to the original data points. This overhead does not yet include the minimum and maximum scalar values associated with each node – necessary information for isosurface extraction. In addition, the leaf node in the BON octrees is a cluster of eight cells, i.e., individual cells are not indexed. The value-partition methods index down to individual cells so that higher interactivity can be provided. However, each cell index entry needs to store the cell's minimum and maximum values and the cell identification. As a result, the total space required for the index can be larger than the size of the original data. Bajaj *et al.* 's method creates seed sets that incur the least amount of space overhead. However, for unstructured grid data, the required adjacency information is often not available and, thus, the space overhead can be comparable to, or even higher than, the value-partition methods if the adjacencies need to be computed and stored.

To our knowledge, to date there is no isosurface extraction algorithm that is optimized for time-varying data. Although it is possible to extend the octrees to the fourth dimension, i.e., time, it can only be used for structured grid data. In addition, the four-dimensional 'octrees' couple together the temporal and the spatial dimensions, which makes cell partitioning awkward because the underlying data may have very different resolutions in time and space. Furthermore, treating temporal and spatial domains as equals impedes the utilization of the temporal coherence existing in the data. In the following, we propose an optimization algorithm for isosurface extraction in time-varying fields. The value-partition paradigm is used because of its interactivity and its equal effectiveness for both structured and unstructured grid data. We assume that the time-varying field has a steady grid, or has a grid that is transformed, but not redefined, over time. Our goal is to reduce the overall size of the search index for data in a time-varying field, while still providing high-performance isosurface extraction.

3 Isosurface Extraction from Time-varying Fields

Given a time interval $[i, j]$ and a time-varying field, we define a cell's *temporal extreme values*, that is, the extreme values over time, in this interval as:

$$min_i^j = \text{MIN}(min_t), t = i..j$$
$$max_i^j = \text{MAX}(max_t), t = i..j$$

where MIN and MAX are the functions that compute the minimum and the maximum values, and min_t and max_t are the cell's extreme values at the t^{th} time step; we call them the cell's *time-specific extreme values*. To locate the isosurface cells in the time-varying field, one can approximate a cell's extreme values at any time step within the time span $[i, j]$ by the cell's temporal extreme values, min_i^j and max_i^j, and use them to create a single search index. Using this approximated search index, an isosurface at a time step $t, t \in [i, j]$, can be computed by first finding the cells that have min_i^j smaller and max_i^j larger than the isovalue. The actual scalar data of these cells at the specific time t are then used to compute the geometry of the isosurface. Using the approximated search index can greatly reduce the storage space required since only one index is used for all the $j - i + 1$ time steps. It also guarantees to find all the isosurface cells because:

$$\text{if } t \in [i, j] \text{ and } min_t < V_{iso} \text{ and } max_t > V_{iso}$$
$$\Longrightarrow min_i^j < V_{iso} \text{ and } max_i^j > V_{iso}$$

where V_{iso} is the isovalue and t is the time step at which the query is issued.

The algorithm just described can be inefficient because the temporal extreme values only provide a necessary but not a sufficient condition to qualify a cell as an isosurface cell. As a result, many non-isosurface cells are visited as well. In the following, we propose an adaptive scheme that enables high performance isosurface extraction, while it also reduces the storage overhead incurred by the search index for isosurface extraction in time-varying fields. We devise a new search index structure, called *Temporal Hierarchical Index Tree*. This tree is built by classifying the cells according to the amount of variation in the cell's values over time. Cells that have a small amount of variation are placed in a single node of the tree that covers the entire time span. Cells with a larger variation are placed in multiple nodes of the tree multiple times, each for a short time span. When generating an isosurface, a simple traversal will retrieve the set of nodes that contains all cell index entries needed for a given time step. The cells in each node can be organized using existing algorithms developed for generating isosurfaces from a steady data set. It is noteworthy that a similar concept independently developed by Finkelstein *et al.* [15] on building a hierarchical representation of multiresolution video has been recently brought to our attention. The paper proposes a 'Time Tree' which is a binary tree of sparse quadtrees. Each node in the time tree corresponds to a single frame at some temporal resolution. The tree can grow to different depths for different regions of the frame to support a video sequence with different temporal resolutions.

3.1 Temporal Hierarchical Index Tree

In this section, the temporal hierarchical index tree data structure is described. We first discuss how to characterize a cell by the temporal variation of its extreme values. We then present the tree construction algorithm using the results of cell characterization.

The span space [3] is useful for analyzing the temporal variation of a cell's extreme values. In the span space, each cell is represented by a point whose x coordinate represents its minimum value

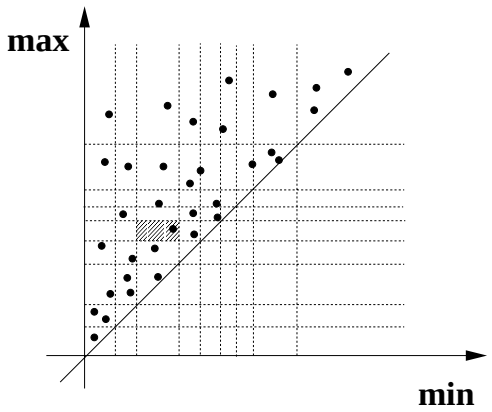


Figure 1: In this example, the span space is subdivided into 9×9 lattice elements. Each lattice element is assigned an integer coordinate based on its row and column number. The shaded lattice element in this figure has a coordinate $(2, 4)$.

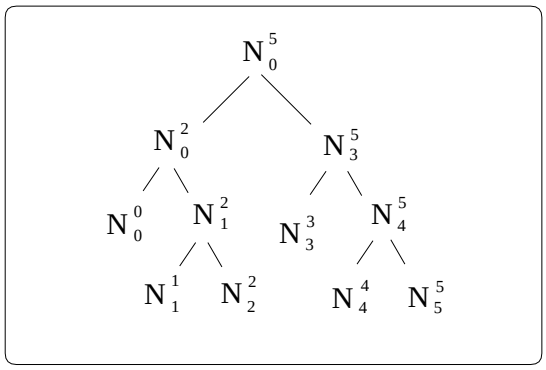


Figure 2: Cells in a time-varying field are classified into a temporal hierarchical index tree based on the temporal variations of their extreme values. In this figure, the tree is built from a time-varying field with a time interval [0,5].

and whose y coordinate represents it maximum value. For a time-varying field, a cell has multiple corresponding points in the span space, and each point represents the cell's extreme values at one time step. To characterize a cell's scalar variation over time, the area over which the corresponding points spread in the span space provides a good measure – the wider these points spread, the higher is the cell's temporal variation. This variation can be quantified by using the *lattice subdivision* scheme of the span space [4], which subdivides the span space into $L \times L$ non-uniformly spaced rectangles, called *lattice elements*. To perform the subdivision, we first sort, in ascending order, all the distinct extreme values of the cells in the time-varying field within the given time interval and establish a list. We then find $L + 1$ scalar values, $\{d_0, d_1, ..., d_L\}$, in the list that can evenly separate the list into L sublists with an equal length. These $L + 1$ scalar values are used to draw $L + 1$ vertical lines and $L + 1$ horizontal lines to subdivide the span space. The list d_i is chosen in this way to ensure that cells can be more evenly distributed among the lattice elements. Fig. 1 is an example of the lattice subdivision.

Using the lattice subdivision, we propose a binary tree data structure, called *Temporal Hierarchical Index Tree*, to classify the cells in a time-varying field based on the temporal variations of their extreme values. Given a time interval $[i, j]$ in the time-varying field, the root node in the temporal hierarchical index tree, denoted as N_i^j, contains cells that have low scalar variations in the time interval $[i, j]$. We determine that a cell has a low temporal variation by inspecting the locations of the cell's $j - i + 1$ corresponding points in the span space. If all of the cell's corresponding points are located within an area of 2×2 lattice elements, we characterize the cell as a cell of low temporal variation. This cell is then placed into the node N_i^j, and is represented by its temporal extreme values min_i^j and max_i^j. On the other hand, for cells that do not satisfy the criterion, we split the time interval $[i, j]$ in half, that is, into $[i, i + (j - i + 1)/2 - 1]$ and $[i + (j - i + 1)/2, j]$, and continue to classify the cells recursively into each of N_i^j's two subtrees that have roots $N_i^{i+(j-i+1)/2-1}$ and $N_{i+(j-i+1)/2}^j$. The temporal hierarchical tree has leaf nodes $N_t^t, t = i..j$. The leaf nodes contain cells that have the highest scalar variations in time so that the cells' time-specific extreme values are used. Cells that are classified into non-leaf nodes are represented by their temporal extreme values. The use of the temporal extreme values directly contributes to the reduction of the overall index size because the temporal extreme values are used to refer to a cell for more than one time step. Fig. 2 shows an example of the temporal hierarchical index tree with a time interval $[0, 5]$.

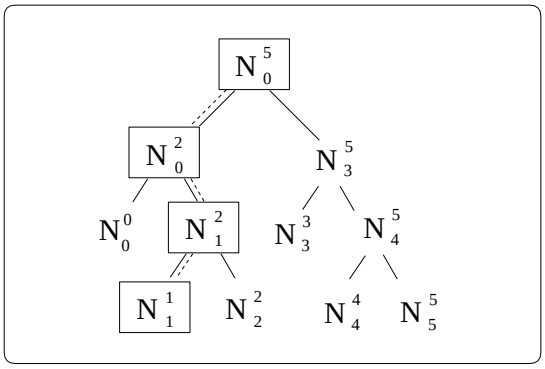


Figure 3: In this example, tree nodes that are inside the rectangular boxes are on the traversal path for an isosurface query at time step 1.

To facilitate an efficient search for isosurface cells, a search index for each node of the temporal hierarchical tree is created. This can be done by using any existing isosurface extraction algorithm based on the value-partition paradigm. Here we propose to use a modified ISSUE algorithm [4] which can provide optimal performance. For every node N_i^j in the temporal hierarchical index tree, cells contained in the node are represented by their extreme values (min_i^j, max_i^j). To create the search index, we use the lattice subdivision described previously and sort cells that belong to the lattice elements of each row, excluding the lattice element at the diagonal line, into a list based on the cells' representative minimum values in ascending order. Another list in each row is created by sorting the cells' representative maximum values in descending order. For those lattice elements at the diagonal line, the interval tree method [9] is used to create one interval tree for each element.

3.2 Isosurface Extraction

Given the temporal hierarchical index tree, this section describes the algorithm that is used to locate the isosurface cells at run time. We first describe a simple traversal method to retrieve the sets of

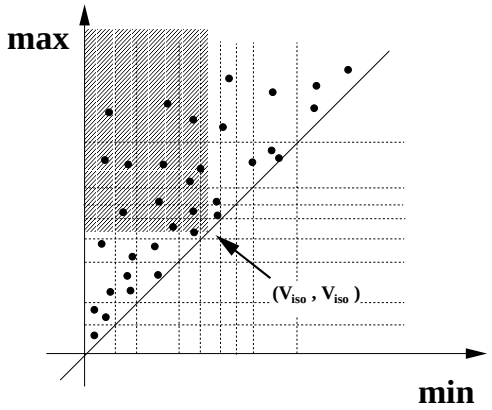


Figure 4: In this case, lattice element $(4, 4)$ contains the point (V_{iso}, V_{iso}). Isosurface cells are located in the shaded area.

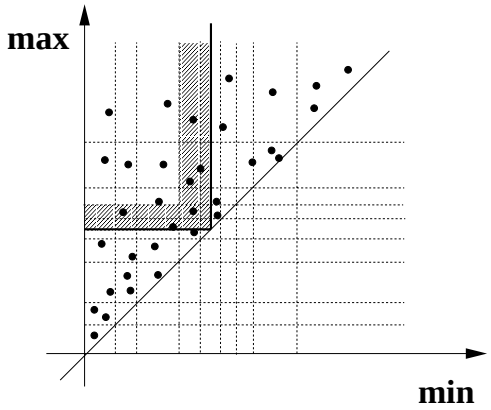


Figure 5: At every tree node, the non-isosurface cells being unnecessarily visited are confined within the two rows and two columns of the lattice elements as shown in the shaded area. Increasing the resolution of the lattice subdivision can reduce the number of cells in this area, for the price of a larger temporal hierarchical index tree.

nodes that contain all cell index entries needed for a given time step. We than describe the isosurface cell search algorithm used for the lattice search index built in each node.

Given an isosurface query at time step t, we compute the isosurface by first locating the nodes in the tree that may contain the isosurface cells. This is done by recursively traversing from the root node N_i^j to one of its two child nodes, N_a^b, such that $a \leq t \leq b$ until the leaf node N_t^t is reached. Along the traversal path, we perform the isosurface cell search, using a method that will be described next, at each encountered node. The tree is constructed so that every cell in the field exists in one of the nodes in the traversal path. These cells have their representative extreme values, temporal or time-specific, as the approximation of their actual extreme values at time step t. Fig. 3 shows an example of the traversal path.

At every node along the traversal path, the lattice search index built at the node is used to locate the candidate isosurface cells. Given an isovalue V_{iso}, we first locate the lattice element with integer coordinates $[I, I]$ that contains the point (V_{iso}, V_{iso}) in the span space. The isosurface cells are then located in the upper left corner that is defined by the vertical line $x = V_{iso}$ and the horizontal line $y = V_{iso}$ as shown in Fig. 4. The candidate isosurface cells can be collected from the following three categories:

- **1**. For every list in the row $R, R = I + 1..L - 1$ that was sorted by the cells' minimum values, we collect the cells from the beginning of the list until the first cell is reached which has a representative minimum value that is greater than the isovalue.

- **2**. For the list in row *I* that was sorted by the maximum values, we collect the cells from the beginning of the list until the cell is reached which has a representative maximum value that is smaller than the isovalue.

- **3**. Collect the isosurface cells from the interval tree built at lattice element $[I, I]$. The method and its details are presented in [9].

After the candidate isosurface cells are located, we then use the cells' actual data at time step *t* to perform triangulation.

Our algorithm has optimal performance since the isosurface cells in categories **1** and **2** are collected without the need for any search. The number of cells in category **3** is usually small. Furthermore, the interval tree method has an optimal efficiency of $O(logN)$, where N is the number of cells in the field.

As mentioned previously, a candidate isosurface cell may not be an isosurface cell after all. These non-isosurface cells come from non-leaf nodes in our temporal hierarchical index tree since a cell's time-specific extreme values, min_t and max_t, may not contain the given isovalue even though the approximated extreme values, i.e., the temporal extreme values min_i^j and max_i^j, do contain the isovalue. Although this problem will not cause a wrong isosurface to be generated, since the triangulation routine will detect the case and create no triangles from these cells, it does incur performance overhead. Actually, this performance overhead is an expected consequence of using temporal extreme values as the approximated extreme values for cells, where we trade performance for storage space.

In fact, the performance overhead is bound by the resolution of the lattice subdivision in the span space. In our algorithm, we place a cell into the node N_i^j in the temporal hierarchical index tree in such a way that its representing points at different time steps within time interval $[i, j]$ always reside within an area of 2×2 lattice elements in the span space. Therefore, for any node N_i^j in the tree, the worst case for the number of the non-isosurface cells being visited is estimated as the number of cells in the two rows and two columns of the lattice elements at the boundary layers of the lattice elements that are searched for the candidate isosurface cells, as shown in the shaded area in Fig. 5. Therefore, the user-specified parameter L, in an $L \times L$ lattice subdivision becomes a control parameter that is used to determine the tradeoff factor between the storage space and the isosurface extraction time.

3.3 Node Fetching and Replacement

Ideally, if the entire temporal hierarchical index tree resides in main memory, there is no I/O required when the user randomly queries for isosurfaces at different time steps. However, the memory requirement is usually too high to make this practical. In our algorithm, the temporal hierarchical tree can be output to a file. When an isosurface at a time step is queried, our algorithm follows the traversal path as described previously and brings those nodes into main memory. Initially, all nodes on the traversal path need to be read in. Subsequently, if the user queries for an isosurface at a different time step, our algorithm traverses the search tree and brings in only those nodes that are not already in main memory. In fact,

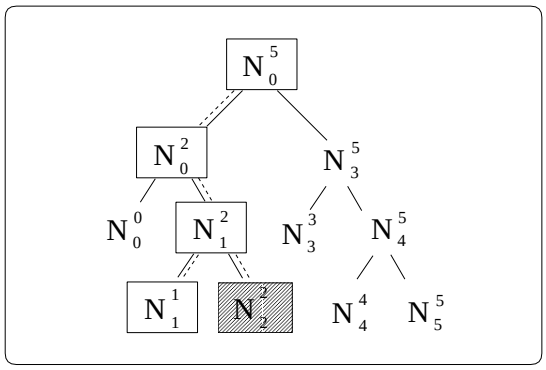


Figure 6: In this case, if the user changes the isosurface query from time step 1 to time step 2, only the node N_2^2 needs to be brought in from the disk.

Data Set	F-18	Delta Wing	Post
# of cells	1,662,290	658,944	123,039
# of nodes	1,764,711	686,147	131,072
Grid size	28.23	8.23	1.57
Solution size	7.05	2.74	0.53

Table 1: Density fields in three CFD simulation data sets were used in our experiments. Information listed here is for one time step, and the file sizes are in megabytes.

because the non-leaf nodes contain cell index entries that are shared by several time steps, they are very likely to be in memory already. In this case, only the differential nodes, a small portion of the index tree, need to be read in from the disk. As a result, the amount of I/O required for a subsequent isosurface query can be considerably smaller. Fig. 6 gives an example.

Although it is always desirable to retain as many nodes in memory as possible in case that the user needs to go back and forth in time when querying the isosurfaces, those nodes that are not in use have to be replaced when the memory limitation is exceeded. To determine which node needs to be replaced, we develop a node replacement policy that assigns a priority to every node, based on its *depth* in the tree. The smaller the depth of a node is, the higher is its priority. For example, the root of a tree has a depth of zero therefore it has the highest priority. The reason is that the root node contains search index entries to those cells that have the lowest temporal variations, and, thus, these index entries are used by many time steps. When a node has to be replaced, we select the node that has the lowest priority. If there are more nodes than one with the same priority, we remove the one that is the least recently used (LRU).

4 Results and Discussion

In this section, we present experimental results of isosurface extraction for time-varying scalar fields using the temporal hierarchical index tree. Three curvilinear gridded time-varying data sets generated from computational fluid dynamics (CFD) simulations were used [16, 17, 18], as shown in Table 1. The time and storage space measurements shown in the following for the Delta Wing and the Post data sets were performed on an SGI Onyx2 workstation with an R10000 microprocessor and 512 megabytes of memory. For the F-18 data set, the measurements were performed on an SGI Onyx2 RealityMonster with an R10000 microprocessor and four gigabytes

Data Set	F-18	Delta Wing	Post
Δ T	100	1	10
Sequence 1	10000-11900	750-769	12000-12190
Sequence 2	12000-13900	770-789	12200-12390
Sequence 3	14000-15900	790-809	12400-12590
Index Size (one time step)			
ISSUE	26.73	10.68	2.10
Interval Tree	26.61	10.55	1.97
Index Size (twenty time steps)			
ISSUE	534.6	213.6	42
Interval Tree	532.2	211	39.4

Table 2: The time sequences in the test data sets and the storage space (in megabytes) required for creating the search indices for one time step and for twenty time steps of data using the ISSUE and the Interval Tree algorithms.

F-18			
Lattice Resolution	10 × 10	40 × 40	80 × 80
Sequence 1	31.6	56.1	82.5
	5.9%	10.5%	15.4%
Sequence 2	32.9	67.2	102.5
	6.2%	12.6%	19.2%
Sequence 3	30.4	53.4	79.3
	5.7%	10%	14.8%

Table 3: The sizes (in megabytes) of the temporal hierarchical index trees for the F-18 data set using three different lattice resolutions.

of memory. We studied the characteristics of our algorithm and compared these characteristics with the regular Marching Cubes algorithm, the Interval Tree algorithm, and the ISSUE algorithm. All of these algorithms were implemented by the author.

In our tests, each temporal hierarchical index tree was built using twenty time steps of data. We performed our experiments at three different time sequences in each of the test data sets, as shown in Table 2; and we denote these sequences as *Sequence 1*, *Sequence 2*, and *Sequence 3*. To understand the storage overhead incurred by the existing value-partition techniques, the Interval Tree and the ISSUE algorithms were used to create search indices for data at every time step. Table 2 shows the sizes of search indices for one time step and the sizes of the search indices for twenty time steps. It is not a surprise that the size of the search index for one time step is much larger than the solution data itself because the cell search index needs to store each cell's minimum, maximum values, and the cell's identification.[1] For a time-varying field such as the F-18 data set, more than 500 megabytes of storage were required to index 20 time steps of data. This overhead is rather overwhelming.

Three different resolutions of lattice subdivisions were used in our experiments to build temporal hierarchical index trees. A coarse resolution of lattice structure indicates that more cells are characterized as having low temporal variations. As a result, the temporal hierarchical index tree will have a smaller size since more cells in the time-varying field are placed into the non-leaf nodes in the tree. The tradeoff is that the search index tree that results from a coarse lattice subdivision will be relatively less efficient in extracting iso-

[1] In our experiments, we intentionally chose not to cluster multiple cells to form meta cells for building the index as in [2, 14], or use the nice chess-board approach as suggested in [9], so we can more easily study the behavior of the underlying algorithms. However, these techniques can be equally well applied to all the methods, including our new algorithm, discussed in this section.

Delta Wing			
Lattice Resolution	10×10	40×40	80×80
Sequence 1	14.4 6.7%	36.2 16.9%	58.4 27.3%
Sequence 2	14.5 6.8%	35.5 16.6%	56.9 26.6%
Sequence 3	14.6 6.8%	37.1 17.3%	59.4 27.8%

Table 4: The sizes (in megabytes) of the temporal hierarchical index trees for the Delta Wing data set.

Post			
Lattice Resolution	10×10	40×40	80×80
Sequence 1	11.9 28.3%	18.5 44%	23.1 55%
Sequence 2	4.8 11.4%	12.7 30.2%	18.9 45%
Sequence 3	4.9 11.7%	12.9 30.7%	19.1 45.5%

Table 5: The sizes (in megabytes) of the temporal hierarchical index trees for the Post data set.

surfaces. Table 3 shows the sizes of the temporal hierarchical index trees built for the F-18 data set. The percentages shown in the table are the ratios of the tree sizes to the overall space required by the ISSUE algorithm, in a period of twenty time steps, as listed in Table 2. The test results from the three different time sequences consistently showed that the storage overhead was significantly reduced, namely from more than 500 megabytes to about 30 megabytes in the 10×10 lattice, and to about 100 megabytes in the 80×80 lattice; the disk space savings amount to more than 80%. Table 4 and Table 5 list the results for the Delta Wing and the Post data sets. The Post data set has a higher scalar variation in time. However, even with a high resolution of lattice subdivision we still had about 50% saving in storage; for the smaller resolutions of lattice subdivision, we achieved about $75\% - 90\%$ space savings.

Table 6 shows the performance of isosurface extraction using the temporal hierarchical index tree for the F-18 data set. We also show the performance of the regular Marching Cubes algorithm (denoted as MCs), the Interval Tree method (denoted as Int. Tree), and the ISSUE algorithm. We chose two representative isovalues at each of the three representative time steps. Among the techniques, the Interval Tree and the ISSUE algorithms have optimal performance, which can save about $80\% - 95\%$ isosurface extraction time compared with the regular Marching Cubes algorithm. Using the temporal hierarchical index tree, it can be seen that when a high resolution lattice such as the 80×80 subdivision was used, the performance of isosurface extractions was very close to the optimal performance gained from using the Interval Tree or the ISSUE algorithms, while only about 20% of the storage space used by the Interval Tree or the ISSUE algorithm was needed for storing the temporal hierarchical index tree. For the low resolution lattice such as the 10×10 subdivision, although the performance was slightly lower, it was still significantly faster than the regular Marching Cubes algorithm. Considering that less than 10% of space was required to store the search index compared with a full set of ISSUE or Interval Tree indices, this tradeoff can be very beneficial for certain applications. Table 7 and Table 8 show the results for the Delta Wing and the Post data sets, which had very similar characteristics. Table 9 shows the number of non-isosurface cells that were visited with lattice subdivisions of different resolutions. The percentage numbers are the ratios to the total number of cells in the field. It can be seen that even with a low resolution subdivision such as 10×10, the overhead is fairly small.

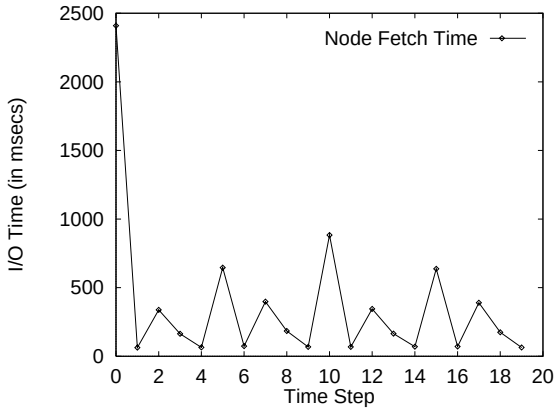


Figure 7: The time (in milliseconds) for restoring tree nodes from the disk when the user sequentially queries the isosurface in time. The F-18 data set was used.

In our algorithm, the nodes in the temporal hierarchical index tree are read into main memory only when necessary. In the case when a user roams a time-varying data set back and forth in time, many non-leaf nodes containing search indices that are shared by consecutive time steps can be retained in memory. As a result, only nodes that are specific to the time step for the current isosurface query need to be brought into main memory and placed into the tree. This can result in a substantially smaller amount of I/O. Fig. 7 shows our experimental results. In our tests, we used the F-18 data set and queried the isosurfaces for a fixed isovalue of 0.99 from time step 10000 to 11900 in ascending order. As shown in the figure, at the first time step, no node in the traversal path was in main memory, so a higher amount of I/O was required. However, in the subsequent time steps, only the nodes that are not resident in main memory needed to be brought in. The amount of time for fetching the nodes shown in the figure is proportional to the number of nodes specific to each time step.

Finally, the color plate shows images of isosurfaces extracted from the test data sets.

5 Conclusions and Future Work

We have presented a new isosurface extraction algorithm for time-varying scalar fields. In the algorithm, we characterize the cells in the field based on their extreme values and the extreme values' variations over time. For a cell that has a low temporal variation, its extreme values at consecutive time steps are coalesced, and the overall extreme values are used to refer to a cell at many time steps. We adaptively compute the representative extreme values for every cell in the time-varying field and place the cells into a search structure called Temporal Hierarchical Index Tree. This index tree can efficiently locate isosurface cells in a time-varying field, while the size of the tree for a series of time steps is substantially smaller than the space required by the search indices of the existing isosurface extraction algorithms. Our algorithm allows flexible control of the tradeoff between performance and storage space and, thus, can be used for data with different characteristics in different computing environments. We have tested our algorithm using three large-scale time-varying data sets from CFD simulations. The space savings

can amount to more than 80%, while the isosurface extraction performance remains nearly optimal. In addition, using the temporal hierarchical index tree, the amount of I/O for accessing the search indices at different time steps can be greatly reduced.

Future work includes devising an out-of-core algorithm for creating and accessing the temporal hierarchical index tree. The method we described in section 3.3 is a coarse out-of-core model since a whole node is fetched into main memory at a time. In fact, it is also desirable to devise a finer grind out-of-core algorithm for accessing the temporal hierarchical index tree so that only the subset of the nodes' lattice needed for the current isovalue is brought into main memory at a time. In addition, we would like to investigate a combination of the space- and value-partition algorithms. Furthermore, developing time-varying methods for surface-propagation schemes is also an interesting research subject.

Acknowledgments

This work was supported in part by NASA contract NAS2-14303. We would like to thank Ken Gee, Neal Chaderjian, and Dennis Jespersen for providing their data sets. Special thanks to Randy Kaemmerer and David Ellsworth for their meticulous proofreading of this manuscript and valuable suggestions. We also thank Tim Sandstrom and other members in the Data Analysis Group at NASA Ames Research Center for their helpful comments and technical support.

References

[1] W.E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[2] J. Wilhelm and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[3] Y. Livnat, H.-W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1), March 1996.

[4] H.-W. Shen, C.D. Hansen, Y. Livnat, and C.R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proceedings of Visualization '96*, pages 287–294. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[5] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4), Dec. 1995.

[6] T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume thinning for automatic isosurface propagation. In *Proceedings of Visualization '96*, pages 303–310. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[7] C.L. Bajaj, V. Pascucci, and D.R. Schikore. Fast isocontouring for improved interactivity. In *1996 Symposium for Volume Visualization*, pages 39–46. IEEE Computer Society Press, Los Alamitos, CA, Oct. 1996.

[8] M. van Kreveld, R. van Oostrum, C.L. Bajaj, D.R. Schikore, and V. Pascucci. Contour trees and small seed sets for isosurface traversal. In *Proceedings of 13th ACM Symposium on Comp. Geom.*, pages 212–219, 1997.

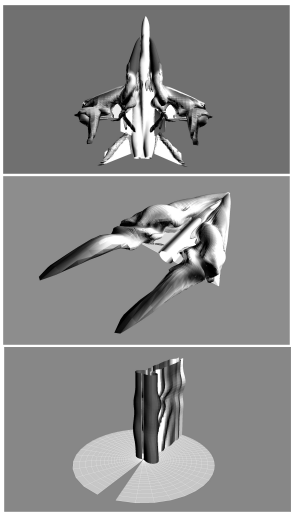


Figure 8: Color Plate – Isosurfaces of density fields in the F-18, Delta Wing, and Post data sets. The surfaces are colored by velocity magnitudes, with red being a high magnitude and blue being a low magnitude.

[9] P. Cignoni, P. Marino, E. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), June 1997.

[10] M. Giles and R. Haimes. Advanced interactive visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[11] R. S. Gallagher. Span filter: An optimization scheme for volume visualization of large finite element models. In *Proceedings of Visualization '91*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[12] H.-W. Shen and C.R. Johnson. Sweeping simplices: A fast isosurface extraction algorithm for unstructured grids. In *Proceedings of Visualization '95*, pages 143–151. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[13] Y.-J. Chiang and C.T Silva. I/O optimal isosurface extraction. In *Proceedings of Visualization '97*, pages 293–300. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[14] Y.-J. Chiang, C.T Silva, and W.J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of Visualization '98*. IEEE Computer Society Press, Los Alamitos, CA, 1998.

[15] A. Finkelstein, C.E. Jacobs, and D.H. Salesin. Multiresolution video. In *Proceedings of ACM SIGGRAPH '96*, pages 281–290, 1996.

[16] K. Gee, S. Murman, and L. Schiff. Computation of F-18 tail buffet. *Journal of Aircraft*, 33(6), Dec. 1996.

[17] D. Jespersen and C. Levit. Numerical simulation of flow past a tapered cylinder. *RNR Technical Report RNR-90-021*, October 1990.

[18] N. Chaderjian and L. Schiff. Navier-Stokes analysis of a delta wing in static and dynamic roll. *AIAA-95-1868*, 1995.

F-18						
Time Step	11000		13000		15000	
Isovalue	0.99	0.93	0.99	0.93	0.99	0.93
# of Triangles	272,163	80,970	257,394	71,689	257,644	73,750
MCs	22.43	21.61	22.38	21.58	22.38	21.59
Int. Tree	4.23	1.21	4.0	1.08	4.0	1.11
ISSUE	4.18	1.18	3.96	1.05	3.96	1.09
Temporal Hierarchical Index Tree						
10×10	5.63	1.50	5.47	1.51	5.23	1.28
40×40	4.76	1.42	4.53	1.33	4.46	1.21
80×80	4.49	1.34	4.27	1.22	4.25	1.18

Table 6: The performance of isosurface extraction (in seconds) for the F-18 data set.

Delta Wing						
Time Step	760		780		800	
Isovalue	0.96	0.89	0.96	0.89	0.96	0.89
# of Triangles	50,962	17,288	52,728	16,760	47,842	17,990
MCs	7.86	7.72	7.87	7.72	7.85	7.73
Int. Tree	0.63	0.21	0.65	0.20	0.59	0.22
ISSUE	0.61	0.20	0.63	0.19	0.57	0.21
Temporal Hierarchical Index Tree						
10×10	1.39	0.35	0.14	0.36	0.13	0.36
40×40	0.82	0.27	0.84	0.28	0.75	0.30
80×80	0.70	0.25	0.73	0.25	0.66	0.27

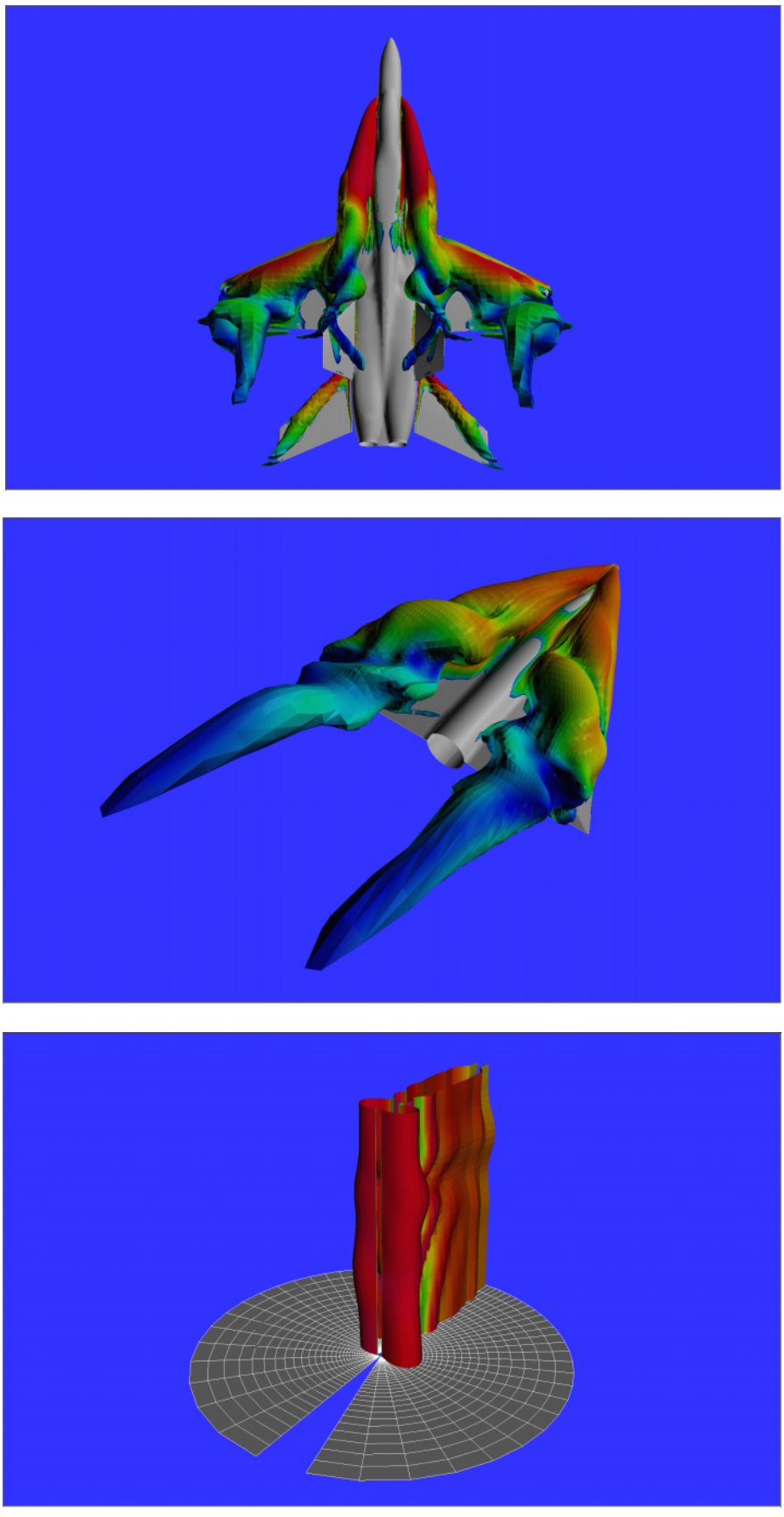
Table 7: The performance of isosurface extraction (in seconds) for the Delta Wing data set.

Post						
Time Step	12100		12300		12500	
Isovalue	1.00	0.98	1.00	0.98	1.00	0.98
# of Triangles	18,932	11,168	20,476	11,480	20,158	11,064
MCs	1.52	1.48	1.52	1.48	1.52	1.48
Int. Tree	0.22	0.13	0.24	0.13	0.23	0.13
ISSUE	0.22	0.12	0.24	0.13	0.23	0.12
Temporal Hierarchical Index Tree						
10×10	0.26	0.16	0.39	0.20	0.39	0.20
40×40	0.25	0.14	0.29	0.16	0.27	0.15
80×80	0.24	0.14	0.26	0.14	0.26	0.14

Table 8: The performance of isosurface extraction (in seconds) for the Post data set.

Lattice Resolution	10×10		40×40		80×80	
F-18	Time Step 13000					
Isovalue	0.99	0.93	0.99	0.93	0.99	0.93
Non-isocell Visited	62,551	20,193	22,972	12,031	11,973	7,186
Percentage	3.7%	1.2%	1.4%	0.7%	0.7%	0.4%
Delta Wing	Time Step 780					
Isovalue	0.96	0.89	0.96	0.89	0.96	0.89
Non-isocell Visited	48,261	9,738	11,289	4,586	4,531	2,918
Percentage	7.3%	1.5%	1.7%	0.7%	0.7%	0.4%
Post	Time Step 12300					
Isovalue	1.00	0.98	1.00	0.98	1.00	0.98
Non-isocell Visited	10,062	4,266	3,138	1,429	1,558	595
Percentage	8.2%	3.5%	2.6%	1.2%	1.3%	0.5%

Table 9: Number of non-isosurface cells that were visited with lattice subdivisions of different resolutions.



Color Plate: Isosurfaces of density fields in the F–18, Delta Wing, and Post data sets. The surfaces are colored by velocity magnitudes, with red being a high magnitude and blue being a low magnitude.