Subdomain Aware Contour Trees and Contour Evolution in Time-Dependent Scalar Fields

Andrzej Szymczak
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332-0280, USA
email: andrzej@cc.gatech.edu

Abstract

For time-dependent scalar fields, one is often interested in topology changes of contours in time. In this paper, we focus on describing how contours split and merge over a certain time interval. Rather than attempting to describe all individual contour splitting and merging events, we focus on the simpler and therefore more tractable in practice problem: describing and querying the cumulative *effect of the splitting and merging events over a user-specified time interval. Using our system one can, for example, find all contours at time t_0 that continue to two contours at time t_1 without hitting the boundary of the domain. For any such contour, there has to be a bifurcation happening to it somewhere between the two times, but, in addition to that, many other events may possibly happen without changing the cumulative outcome (e.g. merging with several contours born after t_0 or splitting off several contours that disappear before t_1).*

Our approach is flexible enough to enable other types of queries, if they can be cast as counting queries for numbers of connected components of intersections of contours with certain simply connected domains. Examples of such queries include finding contours with large life spans, contours avoiding certain subset of the domain over a given time interval or contours that continue to two at a later time and then merge back to one some time later.

Experimental results show that our method can handle large 3D (2 space dimensions plus time) and 4D (3D+time) datasets. Both preprocessing and query algorithms can easily be parallelized.

1. Introduction

Isosurfaces play an important role in visualization and analysis of scalar fields. An isosurface is the set of all points whose value is equal to a user-specified constant (isovalue). The problem of finding a suitable isovalue resulting in a meaningful isosurface has been recognized in [2], where the authors propose a system helping the user to select an 'interesting' isovalue based on several quantitative characteristics of the isosurfaces like surface area, volume and number of connected components.

Time dependent datasets pose similar challenges. Computational fluid dynamics simulation data is often analyzed in terms of features and their behavior in time, in particular their correspondence to features in other time frames and the way the features interact with each other (in particular, how they merge and split) [18, 17, 16]. Features are often obtained by thresholding scalar fields and are therefore closely related to contours (connected components of isosurfaces). Mimicking the static case, one may pose a question of finding 'meaningful' isovalues and contours that could lead to more complete understanding of the data. This paper aims to provide tools for finding contours that evolve and interact with other contours in a way consistent with a number of constraints. In the most general form, each of the constraints requires that a contour intersects a certain simply connected (space-time) subset at a user-specified number of connected components.

Contour trees have been widely used to describe the topological structure of contours [8, 20, 19, 14, 15]. A number of algorithms for computing contour trees have been proposed [20, 19, 8, 6, 7]. In most practical cases, contour trees can be computed in $O(n + m \log m)$ time, where n is the size of the grid and m is the number of critical points. Critical points can be defined as nodes v of the grid at which contour topology undergoes a local change. Time-Varying Reeb Graphs of [10] precisely describe and classify changes that can happen to contour trees of time slices as time progresses. The Safari interface described in [11] allows to interactively explore time-dependent scalar fields. It represents the structure of the isosurfaces computed for a specific isovalue at a specific time by sampling the space of

all pairs (t, c) (t-time, c-isovalue) and then presenting the isosurface properties to the user as an intuitive and easy to navigate through height field parametrized by these two parameters. Note that by doing this, the Safari interface does not capture information how contours deform in time into each other, it only represents properties of entire isosurfaces for time slices.

In this paper we describe an algorithm for finding certain kinds of user-specified behaviors of contours in regularly sampled time-dependent scalar fields. We assume that the data is given in the form of a sequence of regularly sampled scalar fields, defining time sections corresponding to integer times. Let $D \times [T_0, T_1]$ (with T_0 and T_1 integer) be the domain of the input scalar field. In 2D or 3D, D is a rectangle or a parallelepiped (respectively). We first preprocess the input scalar field by computing contour trees of its restrictions to several subdomains. With the precomputed contour trees, we can efficiently find the contour tree of the scalar field restricted to $D \times [t_0, t_1]$ (where $[t_0, t_1] \subset [T_0, T_1]$ and label its every edge e with a vector whose entries are numbers of connected components of the intersection of the contour corresponding to e with integer time sections and $\delta D \times [t_0, t_1]$. In particular, this allows to find all contours which intersect user-specified integer time sections at a prescribed number of connected components. Using the count of connected components in the intersection with $\delta D \times [t_0, t_1]$, one can additionally require that the contours do not intersect the boundary.

One can interpret our approach as a description of merging and splitting events for contours. However, unlike [10], we do not attempt to describe *all* structural changes that happen to the contours. Rather than doing that, we attempt to describe, measure and represent the cumulative effect of all such events between the consecutive time slices. In time-dependent scalar fields, contours can undergo topological changes *between* the consecutive time slices through the data (i.e. at non-integer times) even if the topology on consecutive integer slices is preserved. This is illustrated in Figure 1. In particular, the contour tree for time slice t can change as t increases not only at integer but also at fractional times. Our approach does not attempt to capture all such changes, instead focusing only on the 'total' change between two consecutive time slices. We believe that in practical cases this loss of accuracy is in fact desirable since it allows to eliminate the cost of finding and describing topology changes whose effect is too short lived to be important. Such topology changes may be regarded as temporal topological noise.

As an example, suppose one asks for all contours in $D \times [t_0, t_1]$ (where t_0 and t_1 are integers) whose t_0 time slice has one connected component and whose t_1 time slice has two connected components. The contours that our algorithm would hand in could in fact merge with an arbitrary number (restricted only be the grid size) of contours that are born between t_0 and t_1. Similarly, an arbitrary number of contours that later disappear before t_1 can split off. However, for all these contours the cumulative effect of all these events will be the same: one contour splitting into two. One can refine the search by asking for all contours which intersect slices for times $t_0, t_0+1, t_0+2, \ldots, t_0+k$ at one connected component and slices at $t_0+k+1, t_0+k+2, \ldots, t_1$ at two connected components for some integer k between 0 and t_1-t_0-1. However, this still does not guarantee that the contour undergoes just one splitting event: many more splitting and merging events may happen between the slices.

2. Contour Trees

For a scalar function f defined on a simply connected domain X, its contour tree $\mathcal{T}$ is the quotient space $X/\equiv$, where $\equiv$ is the equivalence relation such that $x \equiv y$ if and only if x and y belong to the same contour. By a contour we mean a connected component of an isosurface. $\mathcal{T}$ is indeed a tree (connected one-dimensional simplicial complex with no loops) in all practical cases, like piecewise linear scalar fields on simplicial domains or scalar fields defined using multi-linear interpolation on regular grids. Note that, since f is constant on every contour, it induces a scalar function on the contour tree $\mathcal{T}$ which we call the *height function*. In practice, one usually wants to compute and use contour trees to describe the topology of discrete isosurfaces computed using a variant of the Marching Cubes algorithm [12], which does not exactly fit into the above framework (for example, isosurfaces for regular grid based data are not really pre-images of a scalar value and they do not even have to be disjoint for different isovalues). As prior work shows, contour trees can nevertheless be defined correctly for a variety of flavors of Marching Cubes (see [5] for a discussion). Generally, different isosurface extraction methods lead to different isosurfaces (and isosurface topologies) and therefore also to different contour trees. Motivated by its simplicity and ease of extension to higher dimensions, we use the method of [4] as the isosurface extraction method for regularly sampled scalar fields. The overall scheme of this algorithm is the same as that of [12]: the vertices of the isosurface are points on the grid edges joining samples of different signs. Their locations are computed assuming that the value varies linearly over the edge. The vertices within a single voxel are then joined by triangles using the information stored in a lookup table. The lookup table is indexed by all possible combinations of states of the vertices of a voxel (two states for a vertex are 'below the isovalue' or 'above the isovalue') and is built as follows. Let H be the convex hull of the set of points consisting of all of the voxel's vertices having value below the isovalue and all centers of edges joining vertices having different state. All triangles of H that are not contained in the boundary of the voxel are put into the lookup table (Figure 2).

Efficient algorithms for computing contour trees have received a considerable amount of attention. Contour tree al-

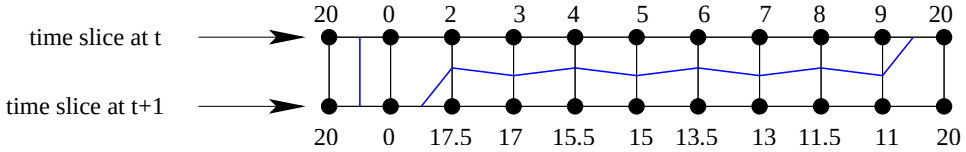

Figure 1. A one-dimensional example showing that a contour may undergo several topological changes between two consecutive slices while having the same topology at the slices. Numbers indicate the density at nodes, blue line is the isosurface (the isovalue is about 10). Notice that even though the topology of time sections for isosurfaces for isovalues close to 10 changes between the two slices, these changes cannot be 'observed' by examining just the topology of the two integer sections.

gorithms can be classified into two categories: sweep and divide and conquer. This paper builds upon both types of algorithms. For the rest of this section, we outline the prior work on computing contour trees and describe how they relate to the regular grid setting.

A divide and conquer algorithm for computing contour trees, readily applicable to the regular grid case, is discussed in [14, 15]. It recursively splits the domain into two parts of roughly equal size, computes contour trees of restrictions to each of the two parts and merges them to obtain the contour tree of the input scalar field. In this paper, we make use of the contour tree merging procedure when executing isosurface queries. The divide-and-conquer algorithm can be made run in $O(n + t \log n)$ time, where t is the number of critical points and n is the size of the grid [14].

The fastest algorithm in the worst case sense is the sweep algorithm described in [7] and based on the results of [6]. It runs in $O(n + m \log m)$ time, where n is the size of the grid and n is the number of critical points. Experiments involving our implementations of this and the divide-and-conquer algorithm indicate that the sweep algorithm tends to be faster in practice (at least within our framework) and therefore we have chosen to use the sweep algorithm to compute the contour trees in the pre-processing stage.

The sweep algorithm of [6, 7] proceeds by first computing the *join tree* and the *split tree* and then combining them together to obtain the contour tree. Join tree describes how connected regions below the isovalue (we shall call their union the *sublevel set*) appear and merge with each other as the isovalue is increased. Split trees describe how connected regions above the isovalue (their union will be called a *superlevel set*) are created and merge with each other as the isovalue is decreased. Computing the join tree amounts to scanning the nodes in order of increasing value and, for each node, describing what components of the sublevel set merge as the isovalue reaches the value at that node. This can be implemented efficiently using the union find datastructure. The split tree can be computed using the dual procedure. A detailed description of this process can be found in [5, Section 14]. Instead of scanning all nodes, one could sweep over the critical nodes (i.e. those where local contour splitting or merging takes place), shooting monotonous

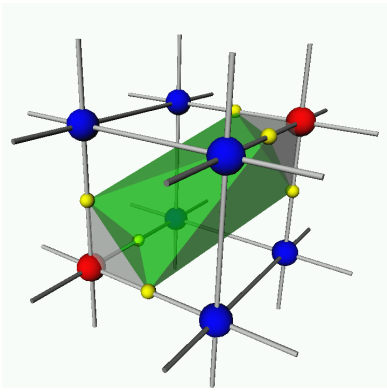

Figure 2. Example entry in the marching cubes lookup table. Samples of positive value are shown in blue and negative samples in red. The yellow spheres are put at the midpoints of edges joining samples of different signs. The triangles that are stored in the lookup table are shown in green.

paths from these nodes until they reach a local extremum or a previously traced path and keeping track of the connectivity of the union of paths as in [7].

The output of the contour tree algorithm is a tree whose vertices correspond to nodes of the grid. Vertices that have exactly one incident edge going up and exactly one incident edge going down will be called regular vertices of the contour tree. All other vertices will be called critical. Note that this notion of criticality is slightly different from the one we used with respect to the nodes of the grid (critical node is one where local connectivity changes happen to contours). In what follows, it will always be clear from context which notion of criticality we use.

We will often simplify contour trees by removing some or all regular vertices. When removing a regular vertex v, we remove its incident edges and connect the lower and upper neighbor of v with a new edge, preserving the tree structure. By removing all regular vertices, one can obtain the smallest possible representation of the contour tree. It

can be *augmented* with additional nodes if necessary. In our case, we will be interested in computing contour trees of restrictions of the input scalar field to certain subdomains. In order for our algorithm to work, we will need to make sure that certain vertices present in certain trees are also present in other trees. This is further explained in the subsequent sections.

3. Finding contours corresponding to contour tree edges

Contours corresponding to a point x of the contour tree can be found using the path seed algorithm of [5]. Assume that x is on edge e with endpoints p_0 and p_1, the height of p_i is h_i ($i = 0, 1$), with $h_0 < h_1$ and the height of x is $h \in (h_0, h_1)$. To find the contour corresponding to x, one can either follow an ascending path from the grid node corresponding to p_0 or a descending path from the grid node corresponding to p_1 until it crosses an isosurface for isovalue of h. The contour can be extracted by contour propagation (as in [21]) from the last edge on the path. However, not every path descending from p_1 or ascending from p_0 would lead to the right contour: we have to make sure that the corresponding path in the contour tree descends or ascends along e. As [5] observes, to find such a path (called the path seed corresponding to e) it is sufficient to follow the *first* edge correctly and then extend the path in any way. We compute path seeds for all edges while building the contour tree using the approach of [5]. This requires remembering the starting edges of monotonous paths used while computing the split and join trees together with their edges and then transferring this information to edges of the contour tree when combining the split and join trees. See [5] for more details.

4. Subdomain Aware Contour Trees

A subdomain aware contour tree is a contour tree equipped with information describing how individual contours intersect a simply connected subdomain $Y \subset X$. More precisely, each edge of the subdomain aware contour tree has a label that represents the number of connected components of the intersection of any contour corresponding to that edge with Y. Below we briefly describe formal properties of subdomain-aware contour trees and the procedure for computing the edge labels.

Let us denote by f_Y the scalar field f restricted to Y. For the scalar field f_Y we also have a contour tree (called the *restricted contour tree* later on), which we denote by $\mathcal{T}_Y$. Since the inclusion map of Y into X maps contours of f_Y into contours of f, one can define the inclusion-induced map j of the restricted contour tree $\mathcal{T}_Y$ into the full contour tree $\mathcal{T}$. Note that j preserves height. In what follows, we require that vertices of $\mathcal{T}_Y$ are also present in $\mathcal{T}$.

The label of an edge e of the contour tree is equal to the number of points in $j^{-1}(p)$ for any point p in the interior of the edge e (i.e. belonging to e but different from its endpoints). This is clear from the definitions: points in the preimage of p correspond to contours of the restricted scalar field that are contained in the contour of the full scalar field corresponding to p.

Each edge in $\mathcal{T}_Y$ is mapped into an ascending path in the full tree $\mathcal{T}$ (in what follows, we call such paths edge paths). A label of an edge e counts how many such paths traverse e. A linear time algorithm of computing the edge labels motivated by the procedure for computing the topology of contours of [14] has been proposed in [3]. We outline it below.

Knowing $\mathcal{T}$, $\mathcal{T}_Y$ and the mapping j, for each vertex v of $\mathcal{T}$, one can easily compute $\Delta(v)$, the sum of all labels of edges joining v with a higher vertex minus the sum of all labels of edges joining v with a lower vertex. For every vertex v of $\mathcal{T}$ which is not a vertex of $\mathcal{T}_Y$, $\Delta(v) = 0$: this is because no edge paths start or end at v. For vertices v that are also present at $\mathcal{T}_Y$, $\Delta(v)$ is equal to the difference of the number of edge paths that start at v and the number of edge paths that end at v (recall edge paths go up). Therefore, $\Delta(v)$ is equal to the number of edges that join v with a higher vertex in $\mathcal{T}_Y$ minus the number of edges that join v with a lower vertex in $\mathcal{T}_Y$.

Having computed $\Delta(v)$ for every vertex of $\mathcal{T}$, we start computing the edge labels. If e is an edge such that, for one of its endpoints v, all labels of edges out of v other than e have been computed (in what follows, we shall call such an edge *simple*), then the label of e can be computed from these labels and $\Delta(v)$. Our algorithm assigns labels to simple edges until all edges are labeled. It maintains a queue of simple edges and a counter $c(v)$ for each vertex v of $\mathcal{T}$. Initially, the queue holds all leaf edges of $\mathcal{T}$ and $c(v)$ is set to the degree of v in $\mathcal{T}$. Every time a label is assigned to an edge e joining u and w, $c(u)$ and $c(w)$ have to be decremented. Whenever the value of a counter $c(v)$ becomes 1, we find the edge incident to v that does not have a label and insert it into the queue. This procedure has to terminate with all edges labeled (all unlabeled edges form a tree, and leaf edges of that tree are simple). The total amount of time spent on computing the labels is $O(n)$, where n is the size of the tree $\mathcal{T}$ ($\mathcal{T}_Y$ has to be smaller since all its vertices must be vertices of $\mathcal{T}$).

The same algorithm can be used to label edges with n-dimensional vectors representing the number of connected components at which contours intersect k simply connected subdomains $Y_1, Y_2, \ldots, Y_k$. Its running time (assuming the full contour tree and contour trees of all restrictions are already computed) can be bounded by $O(nk)$, where n is the number of vertices in the full tree (augmented with all vertices of restricted trees).

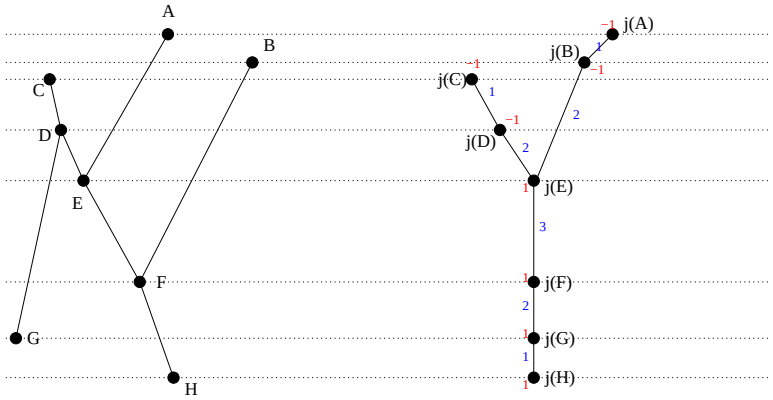

Figure 3. Example: a contour tree of a restricted scalar field (left) and the full contour tree (right) with edge labels. The mapping j is indicated by the vertex labels for the tree on the right. Edge labels count connected components of the intersection of contours with the subdomain that the left tree is based upon. $\Delta(v)$ are shown in red, next to vertices of the restricted tree. Note that we show here only the part of the full tree that is traversed by the edge paths; typically, many edges and vertices of the full tree are not visited by the edge paths at all. The edge labels can be computed in the following order: $j(A)j(B)$, $j(B)j(E)$, $j(C)j(D)$, $j(D)j(E)$, $j(E)j(F)$, $j(F)j(G)$, $j(G)j(H)$.

5. Preprocessing

We preprocess the dataset by computing contour trees of restrictions of the scalar input field to several subdomains. Assume the domain of the input scalar field is $D \times [T_0, T_1]$, where T_0 and T_1 are integers and D is either a rectangle (in the 2D case) or a parallelepiped (in the 3D case). As in Section 1, we assume that the data is represented as a sequence of $T_1 - T_0 + 1$ scalar fields defined on D and corresponding to integer time slices. We compute contour trees for restrictions of the input scalar field to the following subsets, shown in Figure 4:

1. Time slices, $D \times \{t\}$ for all integer $t \in [T_0, T_1]$
2. Thick time slices, $D \times [t, t+1]$ for all integer $t \in [T_0, T_1 - 1]$
3. Thick boundary slices, $(\delta D) \times [t, t+1]$, where $t \in [T_0, T_1 - 1]$ is an integer; Note that thick boundary slices are homeomorphic to the Cartesian product of the two-dimensional sphere and a closed interval and hence simply connected in 3D. However, in the 2D case, the thick boundary slices are not simply connected. In this case, we break them into simply connected parts, for example as shown in Figure 4.

To compute the contour trees, we use the sweep algorithm described in Section 2). We keep the precomputed trees on a hard disk so that they can be conveniently accessed during query execution.

In order to allow queries to run more efficiently, we ensure that the contour trees computed during preprocessing stage have the smallest possible number of vertices that would allow queries to run without flaws. We do that by first computing the contour trees for time slices and simplifying them by removing all their regular vertices. Let R_t be the set of vertices of the tree corresponding to the time slice for time t.

Then, we compute contour trees for the thick boundary slices. For two consecutive thick boundary slices $(\delta D) \times [s-1, s]$ and $(\delta D) \times [s, s+1]$, we compute all vertices in the s-time slice that are critical in either one of the two trees. Let B_s be the set of all such vertices. We simplify the contour tree for thick boundary slice $(\delta D) \times [t, t+1]$ (for all t) by removing all vertices that are not in $B_t \cup B_{t+1}$ (such vertices have to be regular).

Finally, we proceed to compute contour trees for the thick time slices. Let H_s be all vertices in the s-time slice that are critical in the contour trees for thick time slices $D \times [s-1, s]$ or $D \times [s, s+1]$. We simplify the contour tree for a thick time slice $D \times [t, t+1]$ by removing all its vertices except for those in $R_t \cup R_{t+1} \cup B_t \cup B_{t+1} \cup H_t \cup H_{t+1}$ (all such vertices have to be regular).

By obeying the above rules, we can guarantee that:

1. If two thick boundary slices share a vertex and that vertex is critical in the contour tree of one of these thick boundary slices, then it is present in the contour tree for the other one.
2. If two thick time slices share a vertex and that vertex is critical in the contour tree of one of these thick time slices, then it is present in the contour tree for the other one.
3. If a vertex is critical in a contour tree for a time slice or a contour tree for a thick boundary slice, it is present in the contour tree for any thick time slice it belongs to.

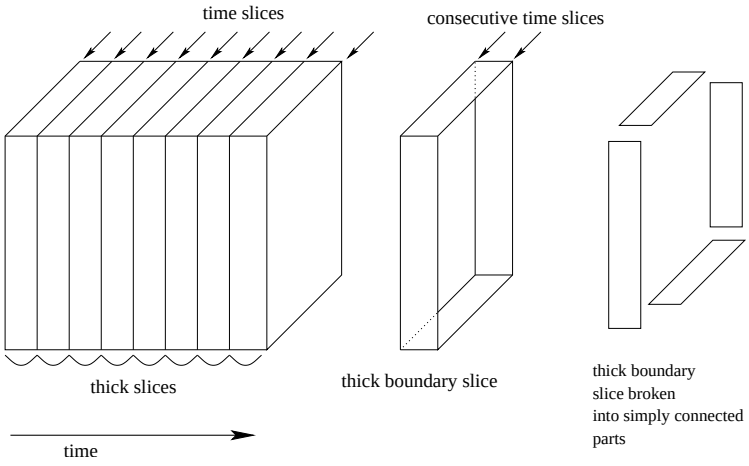

Figure 4. Thick slices, time slices and thick boundary slices for a time-dependent 2D scalar field. In the 2D case, thick boundary slices are not simply connected and therefore cannot be used as subdomains for subdomain aware contour trees. Therefore, we break it into four simply connected parts (right).

These conditions will ensure that certain key contour trees can be derived from the precomputed ones when executing queries.

6. Queries

In this section we discuss the query algorithm in more detail. We focus on queries allowing to find isosurfaces of the scalar field restricted to $D \times [t_0, t_1]$ (where t_0 and t_1 are integers such that $T_0 \leq t_0 \leq t_1 \leq T_1$) which intersect the time slices at t_0 at t_1 at a user-specified numbers of connected components n_0 and n_1. In addition, we allow the user to restrict search to isosurfaces which do not intersect the boundary of the dataset in any time slice, i.e. are disjoint with $\delta D \times [t_0, t_1]$.

6.1. Query execution

In order to execute a query, we first compute the contour trees of the restriction of the scalar field to the following subsets:

(a) $D \times [t_0, t_1]$

(b) $\delta D \times [t_0, t_1]$ (in the 3D case; the 2D case is discussed later).

Both contour trees can be obtained by merging the trees computed in the preprocessing stage using the algorithm of [14, 15]. This naturally leads to a recursive procedure which computes the contour tree of the restriction to $D \times [t_0, t_1]$ by recursively computing trees for restrictions to $D \times [t_0, \lfloor \frac{t_0+t_1}{2} \rfloor]$ and $D \times [\lfloor \frac{t_0+t_1}{2} \rfloor, t_1]$ and then merging the resulting trees if $t_1 - t_0 \geq 2$ or just returning the precomputed contour tree for a thick slice if $t_1 - t_0 = 1$. The tree for the restriction to $\delta D \times [t_0, t_1]$ can be computed in the same way. Since the tree merging operation is linear time [14, 15], the total running time of this algorithm is $O(n(1 + \log(t_1 - t_0)))$, where n is the maximum size of a contour tree for a thick boundary slice or thick time slice. Having computed the two trees, we label each edge e of the tree $\mathcal{T}$ corresponding to $D \times [t_0, t_1]$ with connected component counts of intersections of the corresponding contours with $\delta D \times [t_0, t_1]$ and the time slices $D \times \{t_0\}$ and $D \times \{t_1\}$ (note the contour trees corresponding to restrictions to the two time slices have been computed in the preprocessing stage). Before labeling the edges, the contour tree for $\delta D \times [t_0, t_1]$ is simplified by removing all regular vertices, leading to $\mathcal{T}_\delta$ and $\mathcal{T}$ is simplified by removing all vertices that are not present in $\mathcal{T}_\delta$ or the two trees for time slices at t_0 or t_1. Ultimately, one might want to run simplification after each contour tree merging operation while computing the two trees **(a)** and **(b)**.

In the 2D case thick boundary slices are not simply connected, and the preprocessing stage computes four trees for parts of thick boundary slices shown in figure 4. We can use the contour trees for the parts to compute the contour trees for restrictions of the scalar field to four faces of $\delta D \times [t_0, t_1]$. Then, we compute four edge labels corresponding to the four trees and sum them up to get the a_e label for every edge e. In this case, a_e is not equal to the number of connected components of the intersection of a contour in $D \times [t_0, t_1]$ with $\delta D \times [t_0, t_1]$. However, it still can be used to determine if the contour intersects $\delta D \times [t_0, t_1]$ or not: a_e is nonzero if and only if it does.

7. Experimental results

We have tested our algorithm on two input datasets: simulation of 2D spiral waves [9] obtained using `ezspiral` code [1] and a 3D fluid mixing simulation from Lawrence

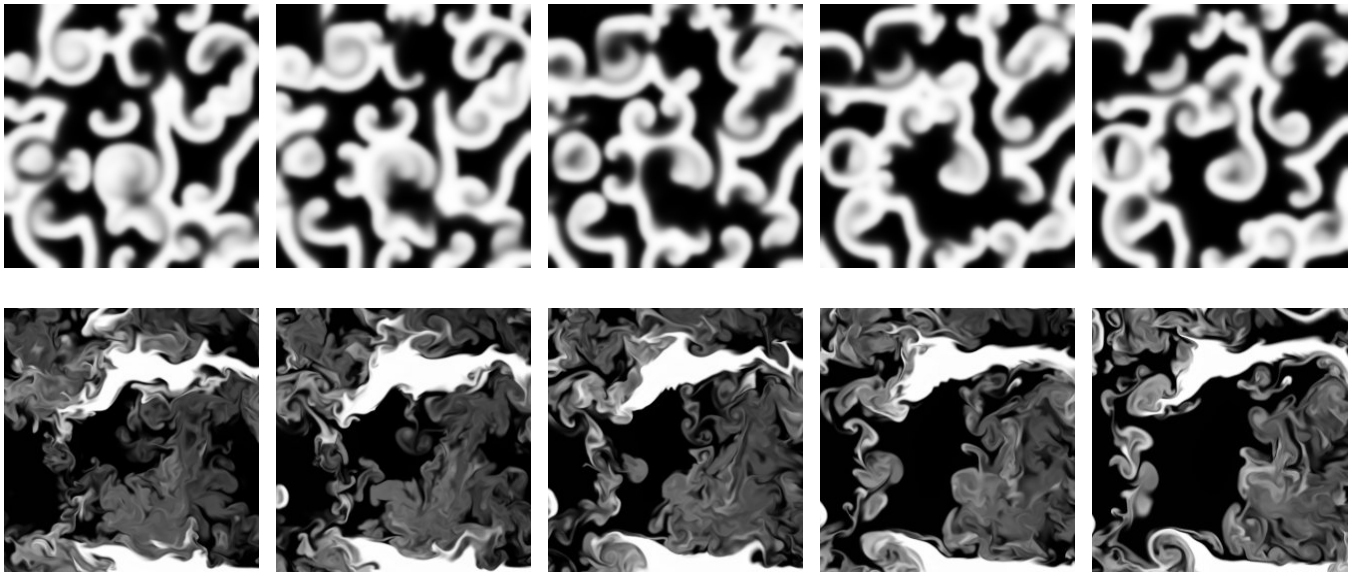

Figure 5. Example time slices for spiral waves dataset (top row) and 2D slices through the fluid simulation dataset (bottom row).

Livermore National Laboratory described in [13]. Example slices through these datasets are shown in Figure 5. The resolution of the 2D dataset is 512×512 with 700 time slices, while the 3D dataset is of resolution $256 \times 256 \times 128$ with 165 time slices. Since the fluid simulation dataset contains numerous high frequency features, which makes it hard to visualize the isosurfaces, we smoothed it by applying the box filter 5 times before using it in our experiments. This substantially reduced the number of contour tree edges resulting from our queries, since a number of them resulted from high frequency in the dataset.

We preprocessed both datasets as described in Section 5. Then, we ran the query algorithm to find isosurfaces that do not hit the boundary and that intersect the first and the last section in a user-selected time range at one or more connected components (i.e. all those that can be continue throughout that time interval). Examples of contours found by our system for the spiral waves dataset are shown in Figure 6. The examples were found by executing the query as described above for time interval equivalent to 50 consecutive slices. The number of edges output was typically small, in fact for many time intervals no contours conforming to our criteria have been found.

Examples of time-sections found in the fluid simulation dataset using the same criteria are shown in Figure 7. In this case, the output consisted of a large number of edges (around 300). The examples we selected to show here are contours whose time sections have a large surface area and which arise from isovalues in the middle of the dynamic range for the input scalar field.

The preprocessing time was about twenty minutes per slice for the fluid simulation dataset and less than 2 minutes per time slice for the spiral waves dataset. The query time (including finding seeds for isosurfaces but without isosurface extraction time) is about a minute for time window equivalent to 50 slices for the spiral waves dataset and about two minutes for the fluid simulation dataset (for time interval equivalent to 10-20 time frames). All time measurements have been made on a Pentium III-850MHz workstation.

The size of the contour trees for each of the thick time slices is about 13000 nodes for the spiral wave dataset and 95000 for the fluid simulation dataset. The contour trees for time slices were typically slightly less than half the size of the thick time slices. The contour trees for the thick boundary slices were much smaller. In both cases, the trees were considerably smaller than the number of nodes in one time slice, which reduces the memory footprint and running time of the query algorithm when compared to simply computing the contour trees from scratch.

8. Summary and future work

We described a system allowing to query large time-dependent datasets for time-space isosurfaces having a prescribed number of connected components in user-specified time sections. While the query times reported in this paper are not interactive, they are low enough to make our algorithm a useful tool for exploring time-dependent data. Furthermore, they easy to implement efficiently on parallel architectures. We believe that the most important task for future research is to incorporate robustness measures in

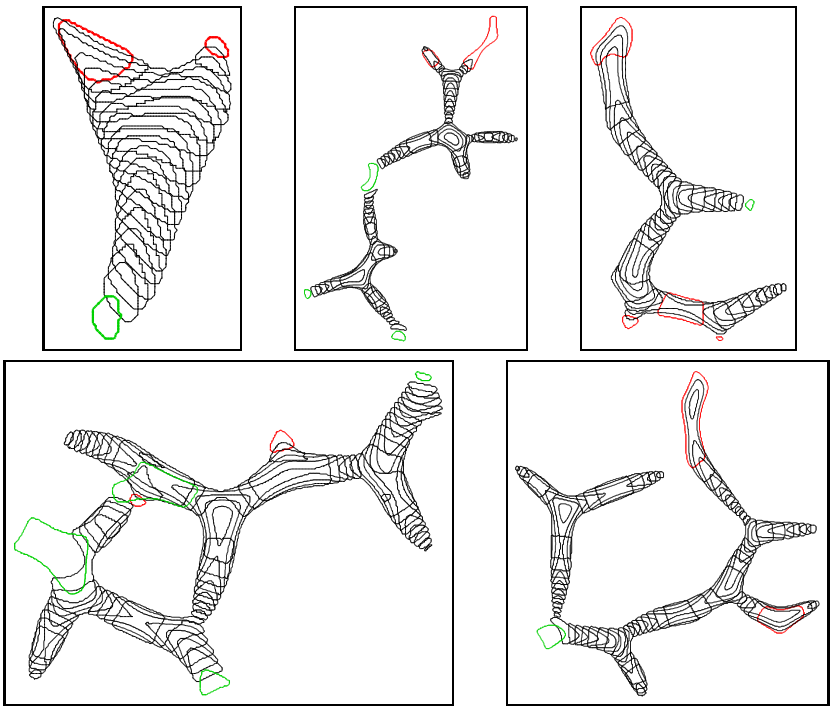

Figure 6. Examples of complex contour evolution found by our algorithm in the spiral wave dataset. Each picture shows several time sections of a space-time contour; the red and green curves are sections in the initial and final time sections (respectively).

our system, allowing to filter out contours (or contour sections) that can be removed by a small perturbation of the data. In particular, we are interested in extending the contour tree simplification algorithms in [5] to our setting. The need for such filters has been demonstrated by our experiments: in some cases we examined, for the fluid simulation dataset the query algorithm reported over 300 edges satisfying the constraints. A number of these edges were short and existed only because of very small connected components created at the first or last frame of the time interval of interest. Such components can be regarded as topological noise and should be omitted from the output. We hope that an effective simplification algorithm of this kind might allow to visualize the topology of complex datasets using contour trees with edge labels in a comprehensible manner.

Another interesting future research topic is to explore the relationship between our work and [10]. For example, given all contour trees that we computed in the preprocessing stage, is there a way to determine the simplest way the contour trees can evolve in time while being consistent with the information carried by these trees? That would lead to a simplified description of contour tree changes in time which could be more comprehensible than the exact description suggested by [10].

References

[1] http://www.maths.warwick.ac.uk/ barkley/ez_software.html, 1999.

[2] C. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *Proc. IEEE Visualization 1997*, pages 167–175, 1997.

[3] N. Berglund and A. Szymczak. Making contour trees subdomain-aware. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04)*, pages 188–191, 2004.

[4] P. Bhaniramka, R. Wenger, and R. Crawfi s. Iso-contouring in higher dimensions. In *IEEE Visualization 2000*, pages 267–273, 2000.

[5] H. Carr. *Topological Manipulation of Isosurfaces*. PhD thesis, University of British Columbia, 2004.

[6] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24:75–94, 2003.

[7] Y.-J. Chiang and X. Lu. Simple and optimal output-sensitive computation of contour trees. Technical Report TR-CIS-2003-02, Polytechnic University, June 2003.

[8] M. deBerg and M. vanKreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18:306–323, 1997.

[9] M. Dowle, R. M. Mantel, and D. Barkley. Fast simulations of waves in three-dimensional excitable media. *Int. J. of Bifurcation and Chaos*, 7(11):2529–2545, 1997.

[10] H. Edelsbrunner, J. Harer, A. Mascarenhas, and V. Pascucci. Time-varying reeb graphs for continuous space-time data. In

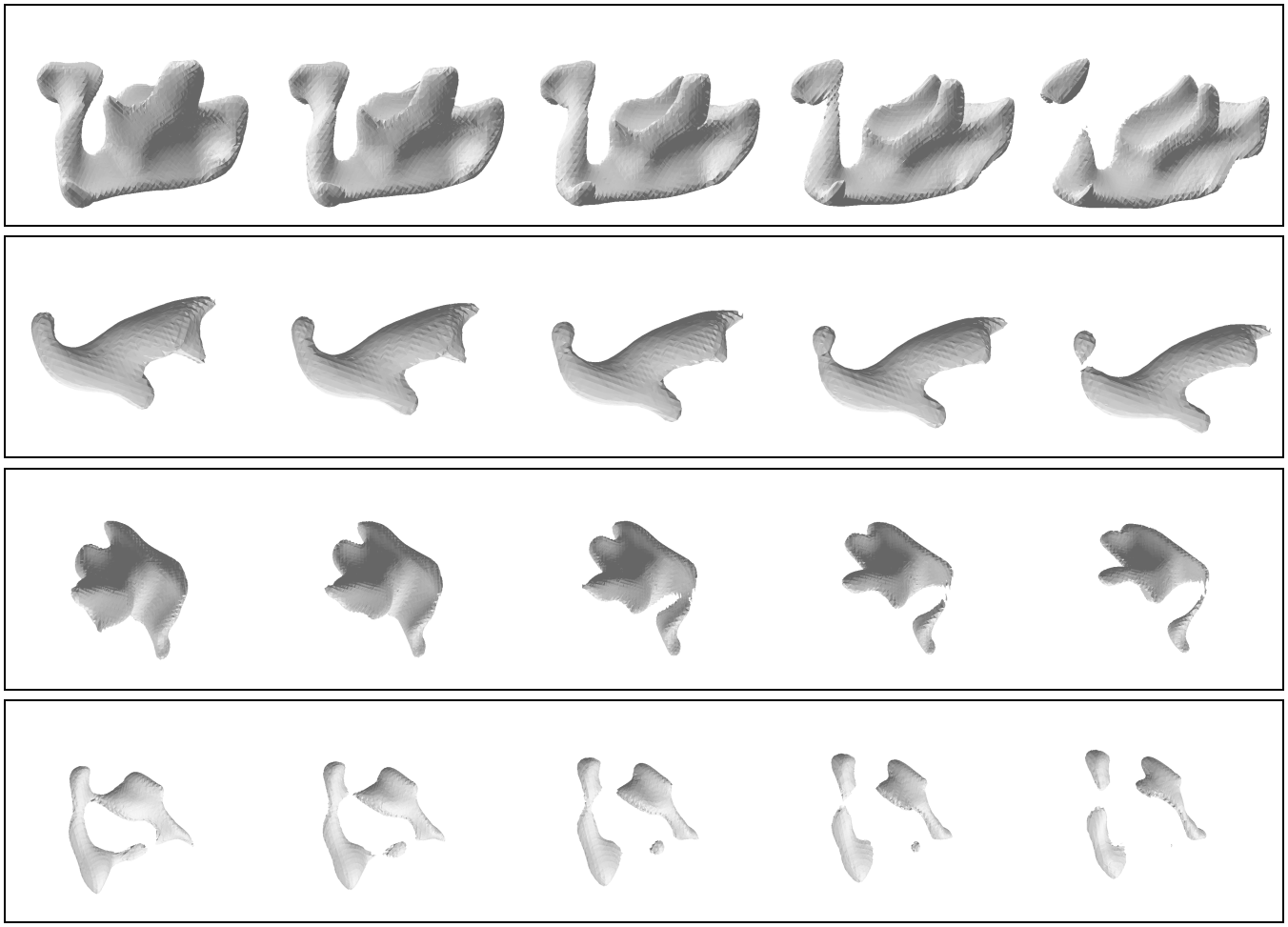

Figure 7. Examples of bifurcating contours evolution found by our algorithm in the fluid simulation dataset.

Proceeding of the 20-th ACM Symposium on Computational Geometry (SoCG), pages 366–372, 2004.

[11] L. Kettner, J. Rossignac, and J. Snoeyink. The safari interface for visualizing time-dependent volume data using isosurfaces and contour spectra. *Comput. Geom.*, 25(1-2):97–116, 2003.

[12] W. E. Lorensen and H. E. Cline. Marching cubes: A high-resolution 3d surface reconstruction algorithm. *ACM Computer Graphics*, 21(3):163–169, 1987.

[13] A. Mirin, R. Cohen, B. Curtis, W. Dannevik, A. Dimitis, M. Duchaineau, D. Eliason, D. Schikore, S. Anderson, D. Porter, P. Woodward, L. Shieh, and S. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing'99*, 1999.

[14] V. Pascucci and K. Cole-Mclaughlin. Effi cient computation of the topology of level sets. In *Proc. IEEE Visualization 2002*, pages 187–194, 2002.

[15] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(2):249–268, October 2003.

[16] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.

[17] F. Reinders, F. H. Post, and H. J. Spoelder. Visualization of time-dependent data using feature tracking and event detection. *The Visual Computer*, 17(1):55–71, 2001.

[18] R. Samtaney, D. Silver, N. Zabusky, and J. Cao. Visualizing features and tracking their evolution. *Computer*, 27(7):20–27, 1994.

[19] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface generation. March 2004.

[20] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th ACM Annual Symposium on Computational Geometry (SoCG)*, pages 212–220, 1997.

[21] B. Wyvill, C. McPheeters, and G. Wyvill. Animating soft objects. *The Visual Computer*, 2(4):235–242, 1986.