Output-Sensitive Construction of Reeb Graphs

Harish Doraiswamy and Vijay Natarajan, *Member, IEEE*

Abstract—The Reeb graph of a scalar function represents the evolution of the topology of its level sets. This paper describes a near-optimal output-sensitive algorithm for computing the Reeb graph of scalar functions defined over manifolds or non-manifolds in any dimension. Key to the simplicity and efficiency of the algorithm is an alternate definition of the Reeb graph that considers equivalence classes of level sets instead of individual level sets. The algorithm works in two steps. The first step locates all critical points of the function in the domain. Critical points correspond to nodes in the Reeb graph. Arcs connecting the nodes are computed in the second step by a simple search procedure that works on a small subset of the domain that corresponds to a pair of critical points. The paper also describes a scheme for controlled simplification of the Reeb graph and two different graph layout schemes that help in the effective presentation of Reeb graphs for visual analysis of scalar fields. Finally, the Reeb graph is employed in four different applications – surface segmentation, spatially-aware transfer function design, visualization of interval volumes, and interactive exploration of time-varying data.

Index Terms—Computational topology, scalar functions, Reeb graphs, level set topology, simplification, graph layout.

1 Introduction

The Reeb graph of a scalar function is obtained by mapping each connected component of its level sets to a point. Level set components that contain critical points of the function map to nodes of the graph. The abstract representation of level-set topology within the Reeb graph enables development of simple and efficient methods for modeling objects and visualizing scientific data. Reeb graphs and their loop-free version, called contour trees, have a wide variety of applications including computer aided geometric design [1], [2], [3], [4], topology-based shape matching [5], topological simplification and cleaning [6], [7], [8], surface segmentation and parametrization [9], [10], [11], and efficient computation of level sets [12]. They serve as an effective user interface for selecting meaningful level sets [13], [14] and for designing transfer functions for volume rendering [15], [16], [17], [18].

Fast computation of the Reeb graph and its efficient representation is key to its successful application to the above-mentioned problems. In this paper, we focus on the problem of computing Reeb graphs with the goals of efficiency and applicability.

1.1 Related work

Several algorithms have been proposed for constructing Reeb graphs. However, only a few produce provably correct Reeb graphs. We restrict our discussion to such algorithms. Shinagawa and Kunii proposed the first algorithm for constructing the Reeb graph of a scalar function defined on a triangulated 2-manifold [19]. Their algorithm explicitly tracks connected components of the level sets and has a running time of $O(n^2)$, where n is the number of triangles in the input. Efficient storage and manipulation of connected components of level sets leads to fast construction of Reeb graphs. Cole-Mclaughlin et al. [20] adopted this approach to obtain an efficient algorithm for 2-manifolds. They improved the running time to $O(n \log n)$ by maintaining the level sets using dynamically balanced search trees. Patanè et. al. [21] focused on 2-manifolds and proposed a contouring approach to compute the Reeb graph in $O(ns)$ time, where s is the number of saddles in the input. This algorithm has a good time complexity when $s = O(\log n)$, but this is rarely the case in real-world data due to the presence of noise. When the number of saddles is large, the running time degenerates to $O(n^2)$.

Doraiswamy and Natarajan [22] stored the connected components of level sets using dynamic connectivity data structures resulting in an algorithm that computes the Reeb graph of a scalar function defined on a 3-manifold in $O(n \log n + n \log g (\log \log g)^3)$ time. Here g is the maximum genus over all level sets of the input function. They extended this approach to higher dimensional manifolds and designed a $O(n \log n (\log \log n)^3)$ time algorithm. This algorithm has the best known theoretical bound on the running time. However, in practice, the sophisticated data structures used in the algorithm do not lend themselves to efficient implementations.

For the special case of loop-free Reeb graphs, called contour trees, Carr et al. [23] described an elegant $O(v \log v)$ algorithm that works in all dimensions, where v is the number of vertices in the input. The algorithm makes two passes over the data to compute the join and split tree, whose union is the contour tree. Chiang et al. [24] proposed an output sensitive approach that computes join and split trees using monotone paths. The presence of loops in the Reeb graph implies that such a decomposition may not exist. Tierny et al. [25] performed a surgery on the 3-manifold domain that cut open all handles of the domain's boundary, thereby reducing the problem to the computation of contour trees. This led to a very efficient

- *Harish Doraiswamy is with Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, INDIA. E-mail: harishd@csa.iisc.ernet.in*
- *Vijay Natarajan is with Department of Computer Science and Automation & Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, INDIA. E-mail: vijayn@csa.iisc.ernet.in*

algorithm. However, the algorithm is restricted to 3-manifolds. More specifically, it works only when the 3-manifold has a single boundary component. Note that 3-manifolds may have more than one boundary component or no boundary at all.

Pascucci et al. [26] proposed an online algorithm that constructs the Reeb graph for streaming data. Their algorithm takes advantage of the coherency in the input to construct the Reeb graph efficiently. In a streaming model, where triangles are processed during a single pass through the triangles in the input mesh, the algorithm essentially attaches the straight line Reeb graph corresponding to the current triangle with the Reeb graph computed so far. Even though the algorithm has a $O(n^2)$ behavior in the worst case, it performs very well in practice for 2-manifolds. However, the optimizations that result in fast incremental construction of Reeb graphs for 2-manifolds do not provide a performance benefit in higher dimensions. We adopt a simple but different approach to compute Reeb graphs that traces connected components of interval volumes, the volume between two level sets. This approach results in an algorithm that exhibits good worst-case behavior and works well in practice. While we obtain good running times for 2-manifolds, our algorithm performs better than the online algorithm for 3-manifolds.

Other algorithms for computing Reeb graphs follow a sample based approach that produces potentially inaccurate results [5], [27].

1.2 Results

We present an efficient two-step algorithm† for computing the Reeb graph of a piecewise-linear (PL) function in $O(n+l+t\log t)$ time, where n is the number of triangles in the input mesh, t is the number of critical points of the function, and l is the total size (number of edges) of all critical level sets. The algorithm has various desirable properties. It is

- *output-sensitive*: the running time depends on the number of critical points of the function, which is equal to the number of nodes in the Reeb graph, and the size of critical level sets, which is indicative of the importance of features in the data.
- *near-optimal*: the size of critical level sets is usually $O(n)$ in practice. So, the worst-case running time is close to the lower bound $\Omega(n+t\log t)$ [12].
- *generic*: the algorithm works, without any modifications, for functions defined on d-manifolds and for non-manifolds. Our implementation can handle data sampled on both unstructured and structured grids.
- *simple*: the algorithm is simple to implement. It consists of a sorting operation followed by a series of tree search operations.

We present experimental results that demonstrate the efficiency of our algorithm. We also address two important issues that are crucial for the application of Reeb graphs to noisy or feature-rich data.

†. A preliminary version of this paper appeared in the Proceedings of the International Symposium on Algorithms and Computation [28]. This paper additionally presents algorithm optimizations that result in significant performance improvements, an embedded layout of the Reeb graph, and four applications of Reeb graphs.

- *Simplification*: We describe a method to simplify the Reeb graph based on the notion of extended persistence [29] that removes short leaves and cycles in the graph.
- *Layout*: We propose a feature-directed layout of the Reeb graph that serves as a useful interface for exploring and understanding three-dimensional scalar fields. We also describe a method to generate an embedded layout of the Reeb graph such that the embedding lies in the interior of the volume.

Finally, we describe four applications of Reeb graphs – segmentation of a surface mesh into meaningful parts, visualization of interval volumes, spatially-aware flexible transfer function design, and interactive exploration of time-varying data.

1.3 Outline

The rest of the paper is organized as follows: Section 2 introduces the necessary definitions and describes the structure and behavior of level sets. Section 3 describes our algorithm to construct the Reeb graph of a scalar function defined on a d-manifold. Section 4 presents experimental results. Section 5 describes techniques for simplification and visualization of Reeb graphs. Section 6 discusses four applications of Reeb graphs and Section 7 concludes the paper.

2 Background

Let $\mathbb{M}^d$ denote a d-manifold with or without boundary. A smooth, real-valued function $f:\mathbb{M}^d\to\mathbb{R}$ is called a *Morse function* if it satisfies the following conditions [20]:

1) All critical points of f are non-degenerate and lie in the interior of $\mathbb{M}^d$.
2) All critical points of the restriction of f to the boundary of $\mathbb{M}^d$ are non-degenerate.
3) All critical values are distinct *i.e.*, $f(p)\neq f(q)$ for all critical points $p\neq q$.

The above conditions typically do not hold in practice for PL functions. However, simulated perturbation of the function [30, Section 1.4] ensures that no two critical values are equal. A total order on the vertices helps in consistently identifying the vertex with the higher function value between a pair of vertices. In the remaining discussion, we assume that the above conditions are satisfied.

2.1 Input

We assume that the input manifold is represented by a triangulated mesh, the function is sampled at vertices, and linearly interpolated within each simplex. In the case of higher dimensional manifolds ($d\geq 3$), the algorithm requires only the 2-skeleton (vertices, edges, and triangles) of the mesh.

2.2 Critical points and level sets

Critical points of a smooth function are exactly where the gradient becomes zero. Banchoff [31] and later Edelsbrunner et al. [32] describe a combinatorial characterization for critical points of a PL function, which are always located at vertices of

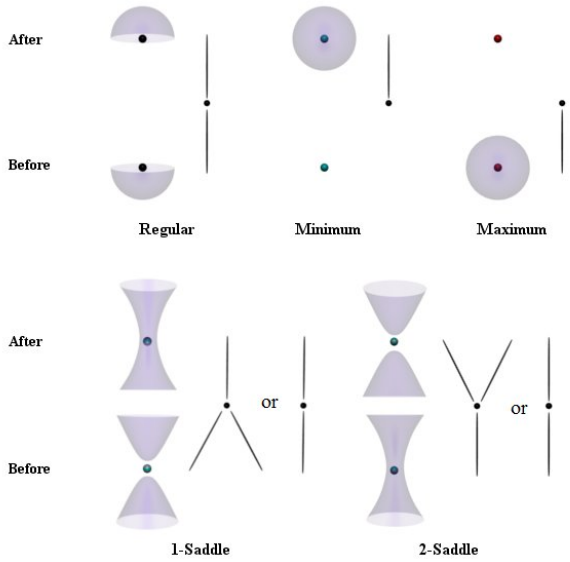


Fig. 1. Isosurfaces before ($f^{-1}(c-\varepsilon)$) and after ($f^{-1}(c+\varepsilon)$) passing through a point with function value c and the structure of the Reeb graph at the corresponding node. Topology of the isosurface changes when it evolves past a critical point. Genus modifying saddles and regular points are optionally included into the Reeb graph as degree two nodes.

the mesh. The *star* of a vertex consists of the incident edges, triangles, and higher-order simplices. All simplices in the star where the function value is greater than the vertex constitute the *upper star*. All simplices in the star where the function value is lower than the vertex constitute the *lower star*. The *link* of a vertex consists of all vertices adjacent to it and the induced edges, triangles, and higher-order simplices. Adjacent vertices with lower function value and their induced simplices constitute the *lower link*, whereas the adjacent vertices with higher function value and their induced simplices constitute the *upper link*. For functions defined on 2- and 3-manifolds, the critical points are classified based on the number of connected components, the zero-th Betti number, of the lower and upper link. Classification of all critical points in higher dimensions requires the computation of higher order Betti numbers.

The preimage of a real value is called a *level set*. The level set of a regular value is a $(d-1)$-manifold with or without boundary, possibly containing multiple connected components. We are interested in the evolution of level sets against increasing function value. Topological changes occur at critical points, whereas topology of the level set is preserved across regular points [33].

In the context of Reeb graphs, we are only interested in critical points that modify the number of level set components. So, it is sufficient to count the number of connected components of the lower and upper links for identifying these critical points. Given a critical point c_i, call the level set $f^{-1}(f(c_i))$ as a *critical level set*.

Consider the case when $d = 3$. A level set of a three-dimensional function is called an *isosurface*. Figure 1 illustrates the topology changes that occur at critical points in a 3-manifold. Specifically, the level set topology changes either by gaining / losing a component or by increasing / decreasing its genus. The isosurface gains a component when it evolves past a minimum and loses a component when it evolves past a maximum. The local pictures in Figure 1 indicate an apparent splitting of a component into two at a 2-saddle and merging of two components at a 1-saddle. Global behavior of the isosurface component will determine if this is indeed a split / merge or a reduction / increase in genus.

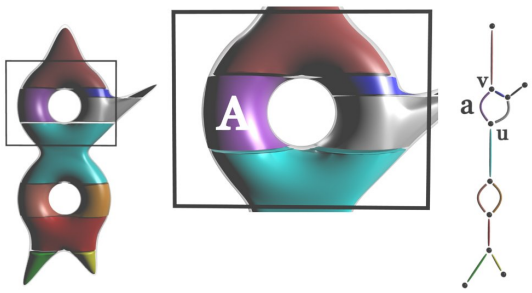


Fig. 2. Reeb graph of the height function defined on a solid 2-torus. The Reeb graph tracks the topology of level sets. Arc a in the Reeb graph maps to cylinder A, a collection of level set components.

2.3 Reeb graph

The *Reeb graph* of f is obtained by contracting each connected component of a level set to a point [34]. Formally, it is the quotient space under an equivalence relation that identifies all points within a connected component of a level set. The Reeb graph expresses the evolution of connected components of level sets as a graph whose nodes correspond to critical points of the function. Figure 2 shows the Reeb graph of the height function on a solid 2-torus. Nodes of the Reeb graph have degree one or three. Figure 1 illustrates the local structure of the Reeb graph at various types of nodes for a three-dimensional scalar function. Nodes corresponding to minima and maxima have degree one. A node that corresponds to a saddle has degree three if the saddle merges or splits level set components. Genus modifying saddles do not alter the number of level set components. They are optionally included into the Reeb graph as degree two nodes as described by Pascucci and Cole-McLaughlin [35].

The above description of the Reeb graph focuses on the mapping between individual level set components and nodes or points within arcs of the graph. We propose the use of an alternate but equivalent mapping, where nodes and arcs of the Reeb graph are mapped to components of critical level sets and equivalence classes of regular level set components respectively. The advantage of our proposed alternate map is that a simple and efficient algorithm to compute the Reeb graph follows immediately from the mapping. We illustrate this idea using an example in Figure 2. The arc a is mapped to *cylinder* A, a collection of regular level set components that are topologically equivalent to each other. The lower boundary

of A consists of a subset of the critical level set $f^{-1}(u)$, and the upper boundary of A consists of a subset of the critical level set $f^{-1}(v)$. The end point v of the arc originating at u can be computed by tracing A from the lower boundary component to the upper component. Different colors in the figure depict the cylinders corresponding to individual arcs of the Reeb graph.

3 The Reeb Graph Algorithm

We now describe an algorithm that computes the Reeb graph of a PL function f defined on a 3-manifold. The algorithm directly extends to d-manifolds ($d \geq 2$) and non-manifolds but in order to simplify the description, we will consider the case of $d = 3$ in this section. The algorithm follows from the alternate mapping described in Section 2.3. It consists of two steps:

1) Locating critical points in the domain and sorting them based on function value.
2) Identifying pairs of critical points that define cylinders and inserting the corresponding arcs in the Reeb graph.

The link of a vertex in a 3-manifold is a triangulation of a sphere. The vertex is *regular* if it has exactly one lower link component and one upper link component. All other vertices are *critical*. A critical point is a *maximum* if the upper link is empty and a *minimum* if the lower link is empty. Else, it is classified as a saddle. We count the number of components in the upper and lower links by performing a breadth first search in the graph formed by vertices and edges in the upper and lower links respectively.

3.1 Level sets and cylinders

The 3-manifold is represented by a tetrahedral mesh. A level set of the input scalar function is generically a surface that is represented by a collection of triangles. However, the algorithm requires only the 1-skeleton representation (vertices and edges) of the level sets in order to track its connectivity. The 1-skeleton of the level set can be extracted from the 2-skeleton representation (vertices, edges, and triangles) of the domain. An edge in a level set lies within a unique triangle of the input triangulation. So, the level set can be represented by the collection of corresponding triangles in the input mesh. Cylinders are also represented as a collection of mesh triangles. Specifically, the cylinder bounded by two critical level set components is represented by triangles that contain the intermediate level set components.

3.2 *LS*-graph

Tracking level set components requires maintaining and updating edges of the level set with changing function value. This maintenance becomes costly for three and higher dimensional data. To avoid such explicit tracking of level sets, we introduce a dual graph that stores triangle adjacencies and helps implicitly track level set components of individual cylinders. This directed graph $G_{LS}(V,E)$, called the *LS-graph,* is a directed graph whose nodes $V = \{t_1, t_2, \ldots, t_n\}$ corresponds to the n triangles $\{T_1, T_2, \ldots, T_n\}$ in the input mesh. Node t_i is assigned a cost equal to the maximum over function values at

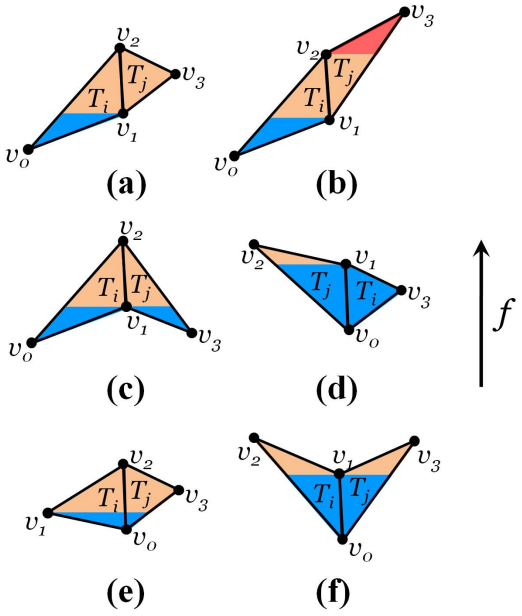


Fig. 3. Adjacent triangles in the input can have one of six possible configurations. The *LS*-graph contains an edge from t_i to t_j in all cases except the forbidden configuration in (f). Edges in the graph are directed towards the node with higher cost. The boundary between the blue and orange regions and the boundary between the orange and red regions indicate the location of the level set edges where the function value becomes greater than $f(v_1)$ and $f(v_2)$ respectively.

vertices of the triangle T_i, and is a representative of all level set components that pass through T_i. Traversing an edge from t_i to t_j in G_{LS} corresponds to moving to a level set at a higher function value. If this edge does not cross a critical value, then the traversal is equivalent to tracing a path within a cylinder. The graph G_{LS} contains an edge from vertex t_i to vertex t_j if triangles T_i and T_j are adjacent, with one exception shown in Figure 3. When t_i and t_j have the same cost, G_{LS} contains an edge from t_i to t_j, as well as from t_j to t_i.

The exception, shown in Figure 3(f), is a configuration where the level set components represented by triangles T_i and T_j possibly belong to different cylinders. Let $\langle v_0, v_1, v_2 \rangle$ with $f(v_0) < f(v_1) < f(v_2)$ be vertices of triangle T_i and $\langle v_0, v_1, v_3 \rangle$ with $f(v_0) < f(v_1) < f(v_3)$ be vertices of triangle T_j. Triangles T_i and T_j share the edge (v_0, v_1) and the cost of t_j (=$f(v_3)$) is greater than $f(v_1)$. Figure 3(f) shows this configuration where the level set component possibly splits into two at $f(v_1)$ during an upward sweep. Inserting an edge from t_i to t_j could allow a graph traversal to jump from one cylinder to another. We do not insert this edge into the *LS*-graph because we are interested in tracking individual cylinders.

3.3 Connecting the critical points

We compute arcs in the Reeb graph by tracing paths in the *LS*-graph. Let $\langle c_1, c_2, \ldots, c_t \rangle$ be the ordered list of critical points with function values $\langle f_1, f_2, \ldots, f_t \rangle$ and $f_x < f_y$ whenever $x < y$.

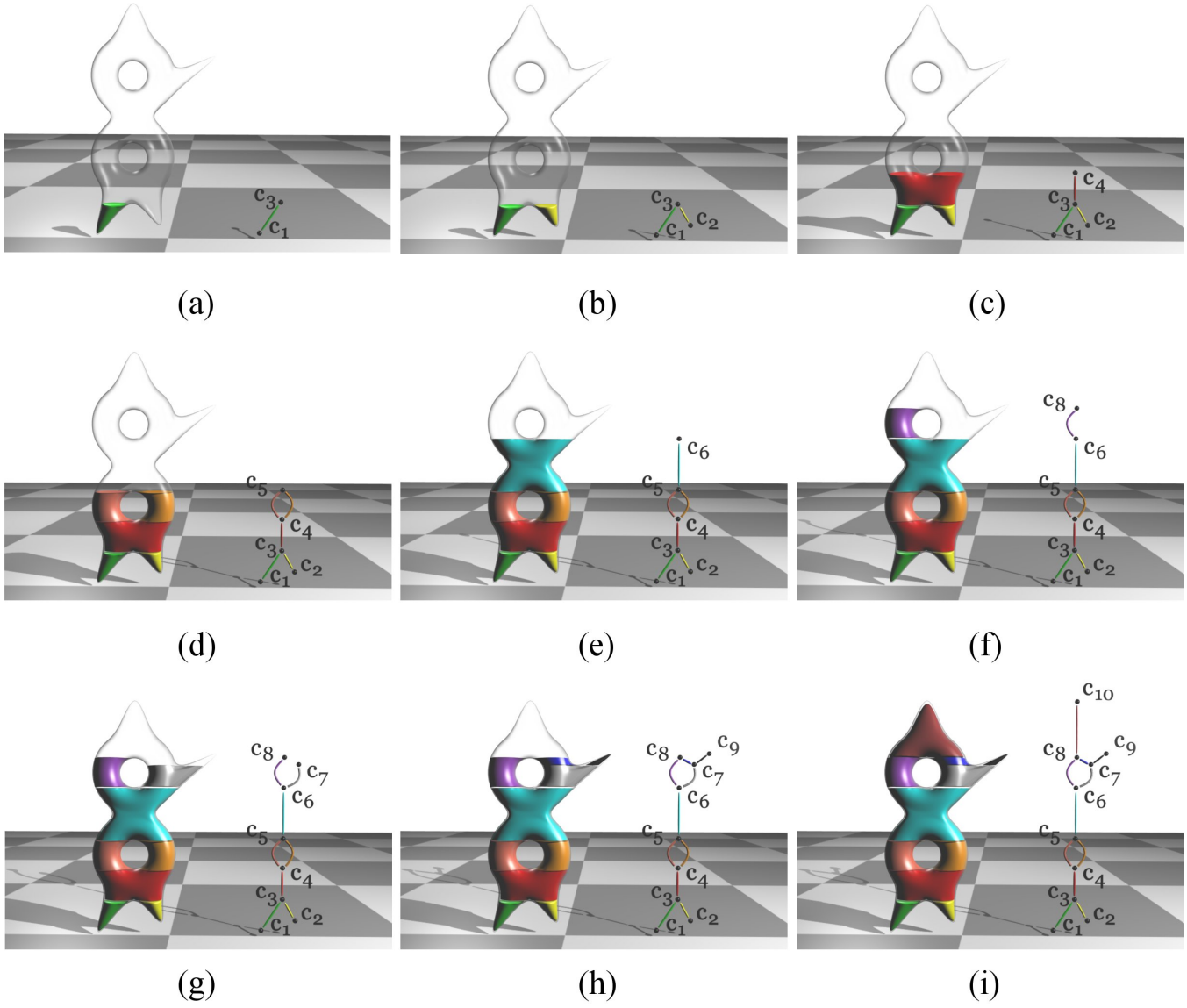


Fig. 4. Illustration of the two-step algorithm computing the Reeb graph of the height function defined on a solid 2-torus. This model has ten critical points, including two minima, two maxima, and six saddle points. The critical points are first sorted in increasing order of function value. Let $c_1, c_2, ..., c_{10}$ be the ten critical points in sorted order. (a) Beginning with a triangle in the upper star of c_1, the algorithm traces the green cylinder to reach L_3 and inserts (c_1, c_3) into the Reeb graph. (b) The search from c_2 also reaches L_3, but a different component as compared to the previous trace. So (c_2, c_3) is inserted into the Reeb graph. (d) The upper star of c_4 has two components. A search is initiated from each component to obtain the two parallel arcs (c_4, c_5) of the Reeb graph. (f) While tracing the cylinder from c_6, the search procedure reaches a triangle with cost greater than f_7 that does not belong to L_7. The search procedure next reaches L_8 and the arc (c_6, c_8) is inserted into the Reeb graph. (i) The Reeb graph of the input function is computed after all critical points are processed.

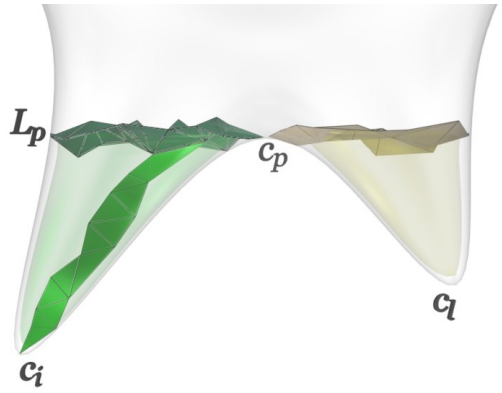


Fig. 5. Connecting critical points. The set of green triangles shows the path traced in the *LS*-graph by the search procedure initiated at c_i. The search terminates when it reaches a triangle in L_p. Similarly the search initiated at c_l also terminates at L_p.

Let L_i denote the set of triangles containing the components of the level set $f^{-1}(f_i - \varepsilon)$ that pass through the lower star of c_i. The i^{th} iteration of the algorithm connects c_i with a set of critical points c_p, where $f_p > f_i$. Figure 4 illustrates the different iterations of the algorithm applied on the height function defined on a solid 2-torus.

The upper star of c_i can possibly contain multiple connected components. Each component of the upper star corresponds to a potential new arc in the Reeb graph that connects c_i with a higher critical point. Figure 4(d) illustrates the case when the upper star of c_4 has two components. We trace the cylinders bounded below by a level set component of c_i in the i^{th} iteration of the algorithm. We initiate a tree search in G_{LS} from a node t that is dual to a triangle T in the j^{th} component of the upper star of c_i. Nodes in the *LS*-graph that belong to this cylinder are labeled $[i, j]$. In each step of the search, we traverse to a higher cost node in G_{LS} and terminate the search when we reach a node t' whose cost is greater than or equal to f_{i+1}. We insert an arc into the Reeb graph between nodes corresponding to c_i and c_{i+1} iff the triangle T' dual to t' belongs to L_{i+1}.

If T' does not belong to L_{i+1}, we continue the search procedure until we reach a node whose cost is greater than or equal to f_{i+2} and test if the dual triangle belongs to L_{i+2}. We repeat the search until we find the set L_p that bounds the cylinder. This operation is shown in Figure 4(f). Figure 5 illustrates the search initiated at two critical points c_i and c_l that terminate in L_p. Triangles in L_p are shaded green and yellow indicating the disjoint components of the level set $f^{-1}(f_p - \varepsilon)$.

If a search initiated from the j^{th} component of the upper star of c_i reaches a node with label $[i, j']$, $j \neq j'$ or if it reaches a node whose dual triangle belongs to a level set component visited during a previous search, then c_i is declared a genus modifying saddle. In either case, the Reeb graph remains unaffected. The search initiated from c_i can never reach a node with label $[i', j]$, $i \neq i'$, for any j because this would imply that two level set components merged at a regular vertex.

We use the triangle-edge data structure [36] to store the input triangulation. The *LS*-graph is implicitly stored in this data structure because each triangle-edge pair stores a reference to neighboring triangle-edge pairs. We traverse from a dual triangle by comparing the function value at vertices of the adjacent triangle. The Reeb graph is stored as an adjacency list whose nodes correspond to critical points of the function. An arc from c_i to c_p is inserted if the search initiated at c_i finds a triangle in L_p. After all critical points are processed, the adjacency list represents the Reeb graph of f.

3.4 Analysis

We first prove that our algorithm indeed computes the Reeb graph of the input scalar function f and then analyze its worst case running time.

Correctness. Let c_i, c_p with $f_i < f_p$ be critical points such that there is an arc from c_i to c_p in the Reeb graph. When we track a level set component beginning at a function value infinitesimally above f_i, the topology of that level set component remains unchanged until the function value reaches f_p. This collection of level set components is exactly a cylinder between c_i and c_p. Consider a triangle T that contains the level set component when the tracking begins. As we increase the function value past the cost of the dual node t, the level set component passes through a triangle adjacent to T whose dual node has a cost greater than that of t. This is equivalent to the search in the *LS*-graph as performed by our algorithm. The algorithm proceeds until we reach a node t' with cost greater than or equal to f_p. Since the cost of the preceding node is less than f_p, a level set component at a function value infinitesimally below f_p will pass through the triangle T' dual to t'. This level set component is a subset of L_p because we have essentially traced the cylinder between c_i and c_p. Our algorithm observes that the triangle T' belongs to L_p and correctly declares (c_i, c_p) to be an arc in the Reeb graph.

Running time. Let n be the number of triangles in the input and t be the number of critical points of the input PL function. Triangles adjacent to a given triangle, that is required for the *LS*-graph traversal, can be found in $O(1)$ time using the triangle-edge data structure. Critical points are located by computing the number of connected components of the lower and upper links, which also takes $O(n)$ time using the triangle-edge data structure. Sorting the critical points takes $O(t \log t)$ time.

The set L_i is extracted by marching through the triangles that contain $f^{-1}(f_i - \varepsilon)$. This task takes $O(l + n)$ time, where l is the total size of all critical level sets. This is because the size of the set L_i is equal to the size of the critical level set $f^{-1}(f_i)$, plus the number of triangles in the lower star of c_i which when summed over all critical points is bounded by $O(n)$. Though it is possible in theory that $l = O(n^2)$, we notice that l is usually $O(n)$ in our experiments.

Each node t_i in the *LS*-graph has at most six neighbors because each triangle is incident to at most two tetrahedra. So, the number of edges in the *LS*-graph is $O(n)$. During the search procedure, each node is labeled exactly once and visited at

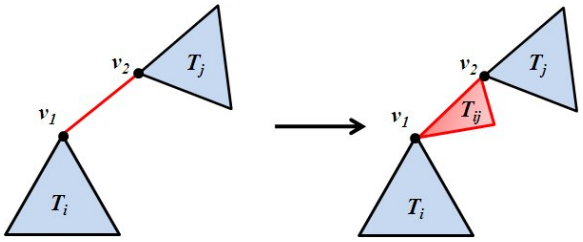


Fig. 6. Replace edge v_1v_2 that is not incident on any triangle with triangle T_{ij}. Our algorithm works on this modified input to compute the Reeb graph. The function value at the new vertex of T_{ij} is set equal to the average of $f(v_1)$ and $f(v_2)$.

most six times, once from each of its labeled neighbors. Thus the traversal of the graph is accomplished in $O(n)$ time. Each update of the adjacency list representation takes constant time. The total number of such updates is equal to the number of arcs in the Reeb graph. A conservative bound for the number of arcs in the Reeb graph is given by the number of triangles in the input. Hence, maintaining the Reeb graph takes $O(n)$ time. Combining the above steps, we obtain an $O(n+l+t\log t)$ bound on the running time of our algorithm.

3.5 d-manifolds and non-manifolds

The level set of a regular value for a Morse function defined on a d-manifold is a $(d-1)$-manifold. The connectivity of a level set is represented by its 1-skeleton. Therefore, tracking the connected components of the level set requires only the edges of the level set, which, as mentioned earlier in Section 3.1, can be extracted from the 2-skeleton of the input mesh. Also, the vertices and edges of the upper links and lower links can be obtained from the triangles of the input. Tracking the cylinders corresponding to each arc of the Reeb graph is accomplished as before using the *LS*-graph, which also requires only the triangles of the input. So, the algorithm works directly on the 2-skeleton representation of d-manifolds.

In the case of non-manifolds, the algorithm again works on the 2-skeleton representation without any modification. Since the algorithm expects the input to be a collection of triangles, edges that are not incident on any triangle in the input are replaced by a triangle as shown in Figure 6. The function value at the additional vertex is equal to the average of the function values of the two end points of the edge. This operation does not affect the Reeb graph of the input because the newly introduced vertex is a regular point. Candidate critical points are again located by counting the number of connected components of the lower and upper link. In case a regular vertex has two link components, we extend the *LS*-graph by appropriately adding edges between triangles present in the two components. When the function value is equal to that of such a vertex, then the level set component will be just a point. Figure 6 shows two such vertices v_1 and v_2, where the level set component becomes a point. The *LS*-graph as defined for a manifold input will not have an edge connecting triangles T_i and T_{ij}, and the *LS*-graph traversal will terminate at t_i. Inserting the new edge overcomes this difficulty.

4 Experiments

The Reeb graph construction algorithm was implemented in Java[§] and tested on a machine with a 64-bit 2.0 GHz Intel Xeon E5405 quad-core processor and 16 GB main memory. Our implementation accepts a function sampled at vertices of a simplicial mesh as input, computes the Reeb graph, and stores it as an edge list. Since our implementation requires the data to be in RAM, for large datasets it may not be possible to store the sets L_i of all critical points found in Step 1 of the algorithm. We develop two optimization strategies that allow us to handle large datasets.

4.1 Optimizations

We noticed in our experiments that level set components do not merge or split at several saddle points. We discard these vertices and compute the sets L_i only for the potential critical vertices and hence reduce the memory footprint considerably. We march through triangles containing the level set $f^{-1}(f_i \pm \varepsilon)$ in a breadth first manner beginning from a triangle in an upper or lower star component of c_i. If this traversal reaches triangles labeled by a traversal from a different upper or lower star component, then we recognize that the level set consists of a single component and stop processing the vertex. This filtering step, in the worst case, requires the additional computation of the level set $f^{-1}(f_i+\varepsilon)$ for all potential critical vertices c_i whose upper link has two components. This overhead is small in practice and many of the false positives are quickly eliminated, and thus the overall time taken to compute all sets L_i is significantly reduced.

As a second memory optimization, instead of processing all critical vertices in a single step, we process critical points in batches of a predetermined granularity. Let g denote this granularity. After sorting the critical points in increasing order of function value, we begin by computing the sets L_i for the first g critical points. Next, we find the arcs, one of whose end points belongs to this set. The second end point of the arc may not lie within the current set of g critical points. In this case, we store the partially traced paths in a queue after the g^{th} critical point is processed. Paths in the queue are traced first when we process the next set of g critical points. By splitting Step 2 of the algorithm into batches, we only need to store the sets L_i for g critical points at any instant of time. The memory required can thus be limited by varying the value of g.

4.2 Results

Table 1 shows the time taken by our implementation to compute the Reeb graph for various models. We compare the performance of our algorithm with the online algorithm of Pascucci et al. [26]. While the online algorithm performs well for 2D data, our algorithm performs substantially better for 3D data. We expect that the algorithm will also be efficient in practice for higher dimensional input. The running time depends on the number of critical points, clearly indicating the output sensitivity of our algorithm. Specifically, for the

§. The source code together with test data is available at *http://vgl.serc.iisc.ernet.in/software/software.php?pid=001*

	Model	#Triangles	#Critical points	Time taken (sec) Our algorithm	Online algorithm [26]
2D	bunny	40000	217	0.7	0.06
	Laurent Hand	99999	92	1.0	0.24
	Neptune	998840	1757	9.0	1.76
3D	Engine	27252	160	1.2	0.26
	Solid 8-torus	34832	18	0.52	0.1
	PMDC	237291	902	2.7	4.9
	Blunt fin	451601	827	23.3	118.8
	Liquid Oxygen Post	1243200	132	13.0	481.1
	Bucky Ball	2524284	4378	197.9	9887.0 ‡
	Plasma	2646016	2852	396.3	11983.2 ‡
	SF Earthquake	4198057	11888	598.1	15949.1 ‡
Non-manifold	Crank (2D)	100056	253	1.5	0.56
	Armadillo-nonmanifold (2D)	331904	462	3.3	2.2
	Fighter (3D)	143881	3618	123.8	44.7

TABLE 1
Reeb graph computation time for various 2D and 3D input. For all 2D models and solid 8-torus, the Reeb graph was computed for the height function. In all other cases, the function is available with the data set.

fighter data, the Reeb graph computation time is high due to the presence of a large number of critical points.

To test the scalability of our algorithm in higher dimensions, we generated Sierpinski simplexes in four, five, and six dimensions and computed the Reeb graph for the height function. In all three cases, the Sierpinski simplexes were generated by repeated subdivision until the number of triangles became greater than 10 million. Figure 7 shows the running times for various cases. Note that the running time scales almost linearly with the number of input triangles, independent of the dimension of the input mesh.

For datasets that do not fit in memory, our implementation can be extended similar to the contour tree algorithm proposed by Chiang et al. [24] by storing the input data using a streaming layout. The layout ensures that triangles adjacent to a given triangle are stored close to each other. Therefore, traversing triangles can be efficiently accomplished in a spatially coherent fashion thus reducing the number of disk accesses.

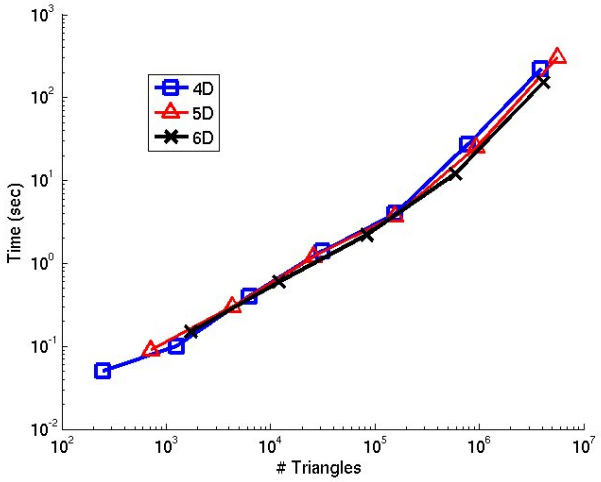


Fig. 7. A log − log plot of time taken to compute the Reeb graph of Sierpinski simplexes of various sizes in different dimensions. Note the near-linear behavior of the algorithm.

5 Visualization of Reeb Graphs

Effective presentation of Reeb graphs is crucial for its application to interactive exploration of scalar fields. Prior to its visualization, simplification of Reeb graphs is necessary for effective visualization of large and feature rich data. Simplification aids in noise removal and creation of feature-preserving multiresolution representations. In this section we first describe a method to simply the Reeb graph, and then provide two layout schemes for visualizing them.

5.1 Simplification of Reeb graphs

A topological feature in the input is represented by a pair of critical points, typically an arc in the Reeb graph. Unimportant features in the data can be removed by repeated cancellation of low persistence critical point pairs [37], which also leads to a multiresolution representation of the input scalar field. Features can also be ordered and removed based on geometric measures like hypervolume [14]. Existing algorithms for contour tree simplification remove critical point pairs that create and destroy a level set component. We simplify the Reeb graph using a notion of extended persistence [29] that pairs all critical points. In particular, it finds pairs for critical points that are creators without a corresponding destroyer.

Our approach to Reeb graph simplification is similar to the one used to simplify contour trees [14]. In addition to the *leaf pruning* and *node reduction* simplification operations, we perform an additional *loop pruning* operation on the Reeb graph. Leaf pruning removes a leaf and the incident arc from the Reeb graph. A leaf connecting to a split or merge saddle is not pruned. Node reduction removes a degree-2 node by merging the two adjacent arcs. The loop pruning operation removes a loop defined by adjacent nodes connected by two

‡. These timings were reported for the online algorithm of Pascucci et al. [26] by Tierny et al. [25]. The experiments were run on a machine with a similar configuration.

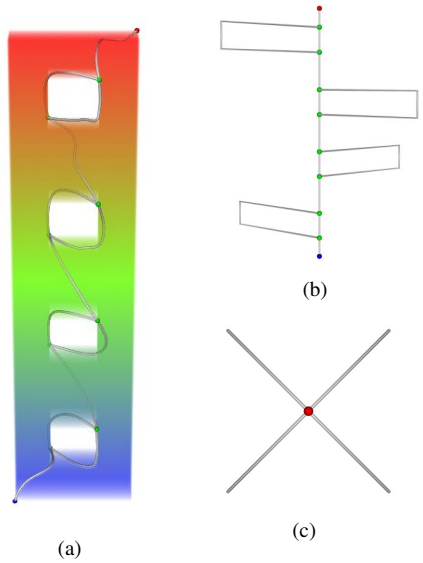


Fig. 8. Reeb graph of the height function on a 4-torus. (a) A volume rendering of the solid 4-torus with the embedded Reeb graph. (b) Side view of the radial layout of the Reeb graph. (c) Top view of the radial layout of the Reeb graph.

parallel arcs. This operation is equivalent to removing one of the parallel arcs and performing node reduction on the pair of adjacent nodes.

We simplify the Reeb graph using repeated application of the three mentioned operations:

1) Perform node reduction where possible.
2) Choose the least important leaf / loop and prune it.

Leaves and loops that can be pruned are stored in a priority queue ordered based on the persistence of the corresponding critical point pair. If a pruning operation results in a reducible node, then node reduction is performed immediately. All new leaves and prunable loops created by the above operations are in turn inserted into the priority queue. Note that we use the simplification process as an aid for visualizing Reeb graphs and not to modify the input function. Realizing the function representing the simplified Reeb graph may require changing the topology of the input.

5.2 Reeb graph layout

We now describe two different layout schemes for visualizing the Reeb graph. The first scheme embeds the Reeb graph within the input domain, whereas the second scheme generates an abstract visual representation of the hierarchical structure of the topological features in the data.

Embedded Reeb graph layout. Each arc of the Reeb graph is obtained by tracking the corresponding monotone cylinder using the *LS*-graph. The path thus obtained has the property that it lies entirely within the input domain, specifically in the interior of its corresponding cylinder. These paths constitute an embedded layout of the Reeb graph with the property that all arcs lie within the input domain.

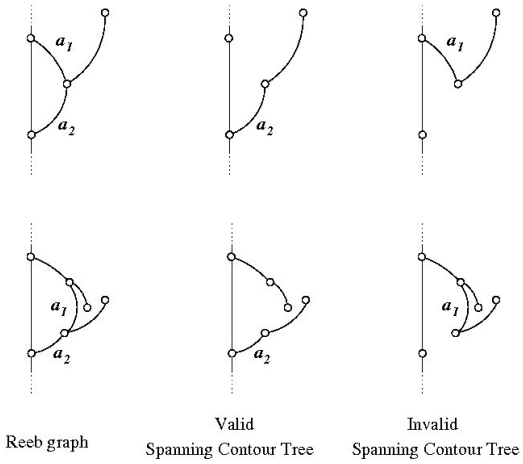


Fig. 9. The spanning contour tree of a Reeb graph is structurally similar to a contour tree. Removal of a_1 results in a spanning contour tree. Removing a_2 results in a tree with an invalid degree-2 node.

Figure 8(a) shows the Reeb graph of a solid 4-torus embedded within its volume. The Reeb graph is computed for the height function defined on the input. Minima are shown in blue, maxima in red, and saddles in green.

Feature directed radial layout. We build upon the orrery layout proposed for contour trees [38] to obtain a layout for Reeb graphs. The extension to Reeb graphs is non-trivial because of the presence of loops. We overcome this difficulty by designing a four step layout scheme:

- First, extract a *spanning contour tree* of the Reeb graph.
- Second, compute a branch decomposition of this spanning tree.
- Third, use a radial layout scheme to embed the spanning tree in 3D.
- Finally, add the non-tree arcs to the layout.

The spanning contour tree is a spanning tree of the Reeb graph that satisfies the structural properties of a contour tree, namely all degree-2 nodes in this spanning tree have exactly one neighbor node with higher function value and one neighbor node with lower function value. Not all spanning trees satisfy this property. For example, in the two graphs shown in Figure 9, removing arc a_1 results in a spanning contour tree. Removal of a_2 also results in a spanning tree, but one that does not correspond to a valid contour tree.

A branch decomposition is an alternate representation of a contour tree that explicitly stores the topological features and their hierarchical relationship [38]. A branch is a path between two leaves of the contour tree or a path that connects a leaf to an interior node of another branch.

All branches of the spanning contour tree are drawn as L-shaped polylines and the y-coordinate corresponds to function value. The (x,z) coordinates are computed for each branch using a radial layout scheme. The root branch is located at the origin and others are placed on concentric circles centered at the origin. All branches that connect to an interior node

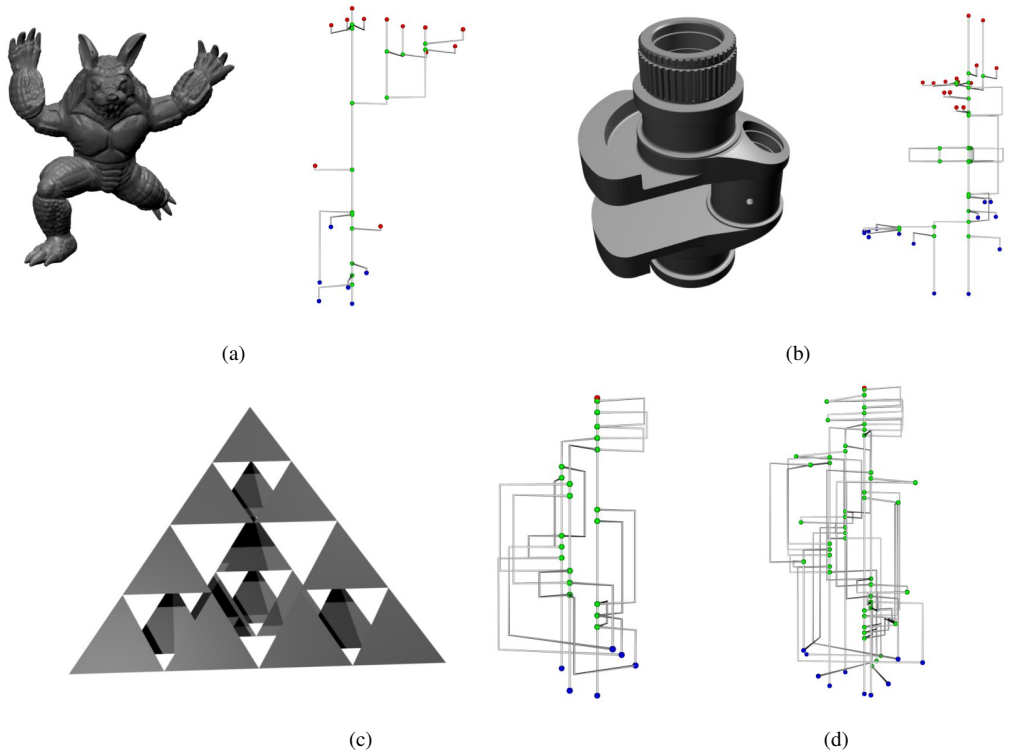


Fig. 10. Reeb graphs computed for the height function defined on non-manifold meshes. (a) Non-manifold Armadillo dataset, (b) the Crank dataset, (c) a 3D Sierpinski simplex subdivided twice, and (d) a 4D Sierpinski simplex subdivided twice.

of the root branch are equally spaced around the origin at a fixed distance from it. Branches that connect to an interior node of a first-level branch are placed in the second concentric circle within a wedge centered at a level-one branch. The angle subtended is proportional to the number of descendant branches. In order to avoid intersections when the non-tree arcs are inserted, we include a dummy branch for each loop arc before calculating the angular wedge subtended at each branch. Figures 8(b) and 8(c) show the layout for the Reeb graph computed for the height function defined on a 4-torus. Figure 10 shows the radial layout for Reeb graphs of a few non-manifold meshes.

6 Applications of Reeb graphs

We now describe four applications of Reeb graphs to visualization and graphics.

6.1 Segmentation of surface meshes

The cylinders partition the input mesh into potentially interesting features. A minor extension of our two-step algorithm also traces the cylinders. While computing the arcs of the Reeb graph, instead of tracing a single monotone ascending path within a cylinder, we trace all monotone ascending paths in the cylinder. This is accomplished by performing a depth first traversal or a breadth first traversal in the *LS*-graph beginning from a node dual to a triangle in the upper star of the critical point c_i. The set of triangles dual to *LS*-graph nodes visited during this traversal constitute the cylinder formed by the arc (c_i, c_p).

In order to find interesting features on the surface, we compute the average geodesic function on the input mesh [5], and use the Reeb graph computed on this function to segment the surface. Figure 11(a) shows the Olivier hand model partitioned using the Reeb graph. Figure 11(b) shows a segmentation of the raptor model into its key features such as the main body, tail, legs and talons, jaws, and tongue. In both examples, we use the simplified Reeb graph to identify the segments and appropriately color each segment. We group arcs of the simplified Reeb graph into different clusters based on the location of the segment corresponding to it, and assign different colors to each cluster. This operation is currently done manually, but can be automated with further geometric processing.

6.2 Reeb graphs of interval volumes

Scientific simulation data and measurements from imaging devices are often available as scalar values sampled on a three dimensional rectilinear grid. The scalar values in the interior of a cell is computed using trilinear interpolation. Since the input volume may have an irregular shape, it is quite likely that several cells in the rectilinear grid are not present in the original volume. These cells are padded with a scalar value of zero or a suitable constant. This results in a loss of the original topology of the input domain, which now becomes simply connected. We study the input scalar field by computing the Reeb graph of interval volumes [39], [40], which is the preimage of a given range of scalar values.

We first convert the rectilinear grid into an unstructured

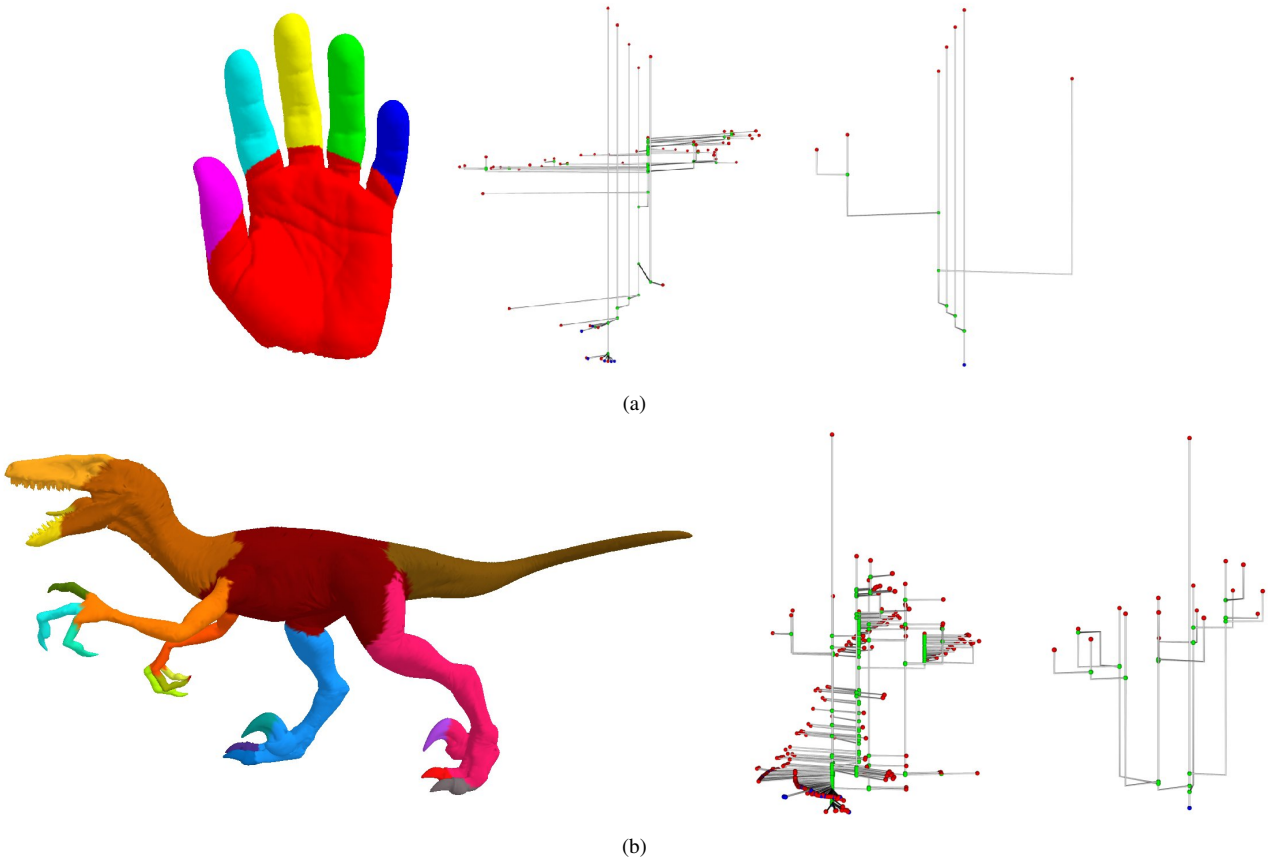


Fig. 11. Using Reeb graphs to segment surfaces into features of interest. (a) Segmentation of the Olivier hand. (b) Partition induced by the Reeb graph on the Raptor model. The Reeb graph and the simplified Reeb graph used to segment the surface are also shown beside the models.

mesh by decomposing each cube into a set of tetrahedra following the method outlined by Sohn [41]. This decomposition preserves the topology of all isosurfaces. Assuming trilinear interpolation, the value of the input scalar function at a point (x,y,z) within a unit cube is

$$\begin{aligned} f(x,y,z) = & f_{000}(1-x)(1-y)(1-z) + f_{001}(1-x)(1-y)z \\ & + f_{010}(1-x)y(1-z) + f_{011}(1-x)yz \\ & + f_{100}x(1-y)(1-z) + f_{101}x(1-y)z \\ & + f_{110}xy(1-z) + f_{111}xyz, \end{aligned}$$

where f_{ijk} is the value of the function at the vertex (i,j,k) of the cube. Similar to PL functions defined on tetrahedral meshes, maxima and minima of the piecewise-trilinear function occur at vertices of the grid. However, a saddle point may be located on a face or within the body of the cube. Saddle points can be located by equating the partial derivatives of f to zero and applying the necessary boundary conditions. Each cube is then decomposed into a constant number of tetrahedra depending on the number of face and body saddles [41]. If a tetrahedron thus created contains the boundary of the isovolume, we first split the tetrahedron along the boundary into a smaller tetrahedron and a prism. We retain the smaller tetrahedron or the subdivided prism depending on which lies in the interior of the isovolume.

We run our two-step algorithm on the tetrahedral mesh obtained from the above-described decomposition. Generating the mesh takes time linear in size of the grid. Also, the number of tetrahedra in the generated mesh is linear in the number of grid nodes. Thus, the time complexity for computing the Reeb graph for a structured mesh remains unchanged. Figure 12(a) shows a volume rendered image of the silicium data set along with its Reeb graph embedded within the volume. The Reeb graph was computed for the original dataset. Figures 12(b) and 12(c) and the accompanying video show the Reeb graphs of an interval volume extracted from the data. The Reeb graph for the height function of the original input would be a straight line, while the Reeb graph computed after removing the padding exhibits loops as shown in Figure 12(c).

6.3 Spatially-aware transfer function design

Reeb graphs can be used to design effective transfer functions for volume rendering [15], [17]. Each cylinder can be accessed using arcs of the Reeb graphs and assigned individual colors and opacity, thereby creating a volume rendered image that distinctly highlights the user-specified areas of the volume. We propose a procedure that allows the user to identify and highlight regions of the volume that are characterized by its geometric feature. The user could specify a different transfer function for a specific geometric feature of interest as compared to the rest of the volume. The main idea here is to choose a region of interest with ease in the volume rendered image based on the geometry of the input. We describe this

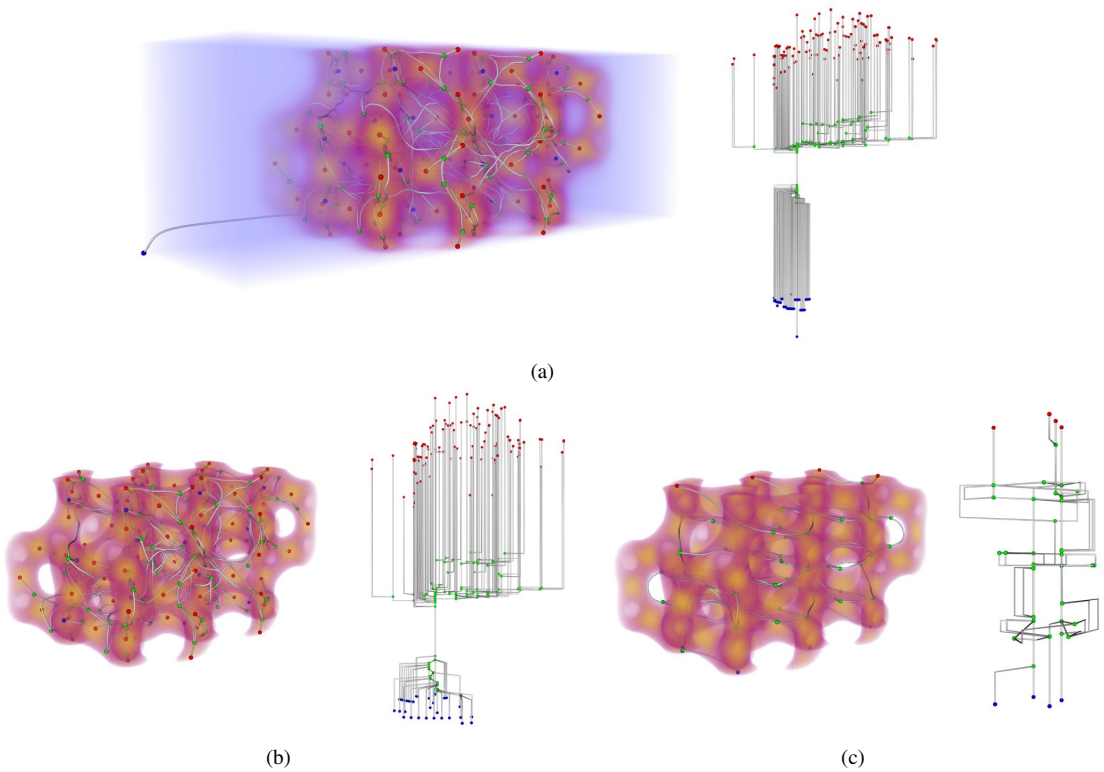


Fig. 12. Visualization of the Silicium dataset: volume rendered images with the embedded Reeb graph and the radial layout. (a) The rectilinear volume. (b) Interval volume, and (c) Height function (y-coordinate) defined on the interval volume.

procedure below:

1) Compute the Reeb graph for a geometric function defined on the volume.
2) Select the required feature using this Reeb graph. The feature might correspond to a loop in the graph or a collection of arcs.
3) Design a transfer function that highlights the selected feature when compared to the rest of the volume. The cylinders corresponding to the selected feature are rendered using this transfer function.
4) Design a transfer function for the rest of the volume possibly using the Reeb graph of the input scalar function.

Figure 13(a) shows a volume rendering of the silicium dataset. We use the interval volume obtained by removing the padding, and compute the Reeb graph for the height function (y-coordinate) defined on this volume. We highlight two atoms in the data set by selecting a loop in the Reeb graph and designing a different transfer function for the corresponding cylinders. Figure 13(b) highlights a single atom in the silicium data set. In this case, we select one arc from the loop.

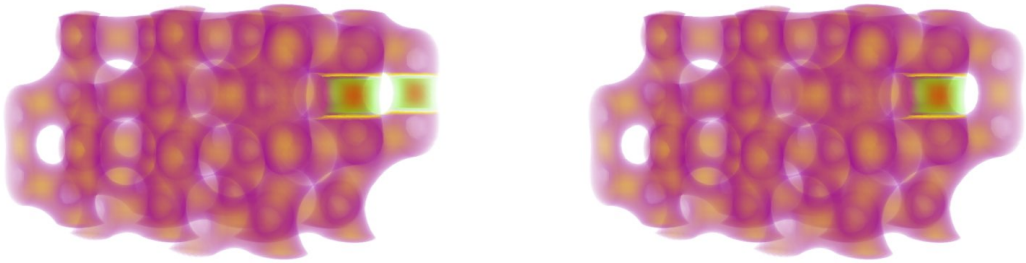
6.4 Interactive exploration of time-varying data

Time-varying data can be considered as a four dimensional scalar field defined on a 4D grid. We decompose each 4D hypercube in the grid into a set of pentatopes or 4-simplices. We use this triangulated mesh as input and compute the Reeb graph. By providing an interface to select arcs of the Reeb graph, we are able to interactively view the corresponding cylinders and explore the given time-varying data.

Figure 14 shows results of our experiment on the pressure field in the hurricane Isabel data set. Figure 14(a) shows the input as a set of volumes at four different time steps. The Reeb graph corresponding to the input time-varying function is shown on the right. Notice that by selecting an arc in the Reeb graph, we are able to focus on different features of the input. The arc selected in Figure 14(b) tracks the eye of the hurricane across the different time steps. The arc corresponding to the cylinder having the maximum function range, shown in Figure 14(c) corresponds to region of the hurricane surrounding the eye. Figure 14(d) shows how the user can select a region of interest, namely the eye and the neighboring region over time, by selecting multiple arcs of the Reeb graph.

7 Conclusions

We have described a simple output-sensitive near-optimal algorithm that constructs the Reeb graph of a piecewise-linear scalar function. Compared to prior algorithms that run in $O(n^2)$ time, our algorithm has a worst case running time of $O(n+l+t\log t)$, where n is the number of triangles in the input mesh, t is the number of critical points of the function and l is size of all critical level sets. The algorithm works without any modification for functions defined on manifolds in any dimension, and for non-manifold domains. We also outlined a method to simplify the Reeb graph based on an extended notion of persistence. Our embedded layout and



(a) (b)

Fig. 13. Reeb graph computed on y-coordinate function in the silicium data set is used to highlight interesting geometric features. (a) Highlight the volumetric region corresponding to a loop in the Reeb graph by designing a different transfer function. (b) Highlight an individual atom by selecting one arc in the Reeb graph and designing a different transfer function for the corresponding cylinder.

feature-directed layout of the Reeb graph serve as useful interfaces for exploring and understanding three-dimensional scalar fields.

We believe that the Reeb graph will soon become a standard tool for exploring scalar data and will supplement existing techniques like level sets, volume rendering, and contour spectrum. In this paper, we discussed how Reeb graphs can be used to segment surfaces and design transfer functions for volume rendering. We described the computation of Reeb graph for interval volumes and time-varying function and how they can be used to interactively study different regions of interest in the data. We have demonstrated through various experiments that our algorithm performs efficiently in practice. Since the iterations of the algorithm are independent of each other, there is an inherent scope for parallelization of our code.

Acknowledgements. Harish Doraiswamy was supported by Infosys Technologies Ltd., Bangalore, under the Infosys Fellowship Award. This work was supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/048/2007. The silicium dataset used in the experiments is courtesy of the volvis repository [42]. The PMDC dataset is courtesy of the TetView software [43]. All other models are courtesy of the AIM@SHAPE repository [44].

REFERENCES

[1] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien, "Surface coding based on Morse theory," *IEEE Comput. Graph. Appl.*, vol. 11, no. 5, pp. 66–78, 1991.

[2] H. S. Y. Shinagawa, T. L. Kunii and M. Ibusuki, "Modeling contact of two complex objects: with an application to characterizing dental articulations," *Computers and Graphics*, vol. 19, no. 1, pp. 21–28, 1995.

[3] S. Takahashi, Y. Shinagawa, and T. L. Kunii, "A feature-based approach for smooth surfaces," in *SMA '97: Proceedings of the fourth ACM symposium on Solid modeling and applications*. New York, NY, USA: ACM, 1997, pp. 97–110.

[4] F. Lazarus and A. Verroust, "Level set diagrams of polyhedral objects," in *SMA '99: Proceedings of the fifth ACM symposium on Solid modeling and applications*. New York, NY, USA: ACM, 1999, pp. 130–140.

[5] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii, "Topology matching for fully automatic similarity estimation of 3d shapes," in *Proc. SIGGRAPH*, 2001, pp. 203–212.

[6] I. Guskov and Z. Wood, "Topological noise removal," in *Proc. Graphics Interface*, 2001, pp. 19–26.

[7] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder, "Removing excess topology from isosurfaces," *ACM Trans. Graph.*, vol. 23, no. 2, pp. 190–208, 2004.

[8] S. Takahashi, G. M. Nielson, Y. Takeshima, and I. Fujishiro, "Topological volume skeletonization using adaptive tetrahedralization," in *GMP '04: Proceedings of the Geometric Modeling and Processing 2004*. Washington, DC, USA: IEEE Computer Society, 2004, p. 227.

[9] M. Mortara and G. Patané, "Affine-invariant skeleton of 3d shapes," in *SMI '02: Proceedings of the Shape Modeling International 2002 (SMI'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 245.

[10] F. Hétroy and D. Attali, "Topological quadrangulations of closed triangulated surfaces using the Reeb graph," *Graph. Models*, vol. 65, no. 1-3, pp. 131–148, 2003.

[11] E. Zhang, K. Mischaikow, and G. Turk, "Feature-based surface parameterization and texture mapping," *ACM Trans. Graph.*, vol. 24, no. 1, pp. 1–27, 2005.

[12] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore, "Contour trees and small seed sets for isosurface traversal," in *Proc. Symp. Comput. Geom.*, 1997, pp. 212–220.

[13] C. L. Bajaj, V. Pascucci, and D. R. Schikore, "The contour spectrum," in *Proc. IEEE Conf. Visualization*, 1997, pp. 167–173.

[14] H. Carr, J. Snoeyink, and M. van de Panne, "Simplifying flexible isosurfaces using local geometric measures," in *Proc. IEEE Conf. Visualization*, 2004, pp. 497–504.

[15] I. Fujishiro, Y. Takeshima, T. Azuma, and S. Takahashi, "Volume data mining using 3d field topology analysis," *IEEE Computer Graphics and Applications*, vol. 20, pp. 46–51, 2000.

[16] S. Takahashi, Y. Takeshima, and I. Fujishiro, "Topological volume skeletonization and its application to transfer function design," *Graphical Models*, vol. 66, no. 1, pp. 24–49, 2004.

[17] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann, "Topology-controlled volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 330–341, 2007.

[18] J. Zhou and M. Takatsuka, "Automatic transfer function generation using contour tree controlled residue flow model and color harmonics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1481–1488, 2009.

[19] Y. Shinagawa and T. L. Kunii, "Constructing a reeb graph automatically from cross sections," *IEEE Comput. Graph. Appl.*, vol. 11, no. 6, pp. 44–51, 1991.

[20] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Loops in Reeb graphs of 2-manifolds," *Disc. Comput. Geom.*, vol. 32, no. 2, pp. 231–244, 2004.

[21] G. Patanè, M. Spagnuolo, and B. Falcidieno, "A minimal contouring approach to the computation of the Reeb graph," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 4, pp. 583–595, 2009.

[22] H. Doraiswamy and V. Natarajan, "Efficient algorithms for computing reeb graphs," *Comput. Geom. Theory Appl.*, vol. 42, no. 6-7, pp. 606–616, 2009.

[23] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Comput. Geom. Theory Appl.*, vol. 24, no. 2, pp. 75–94, 2003.

[24] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Comput. Geom. Theory Appl.*, vol. 30, no. 2, pp. 165–195, 2005.

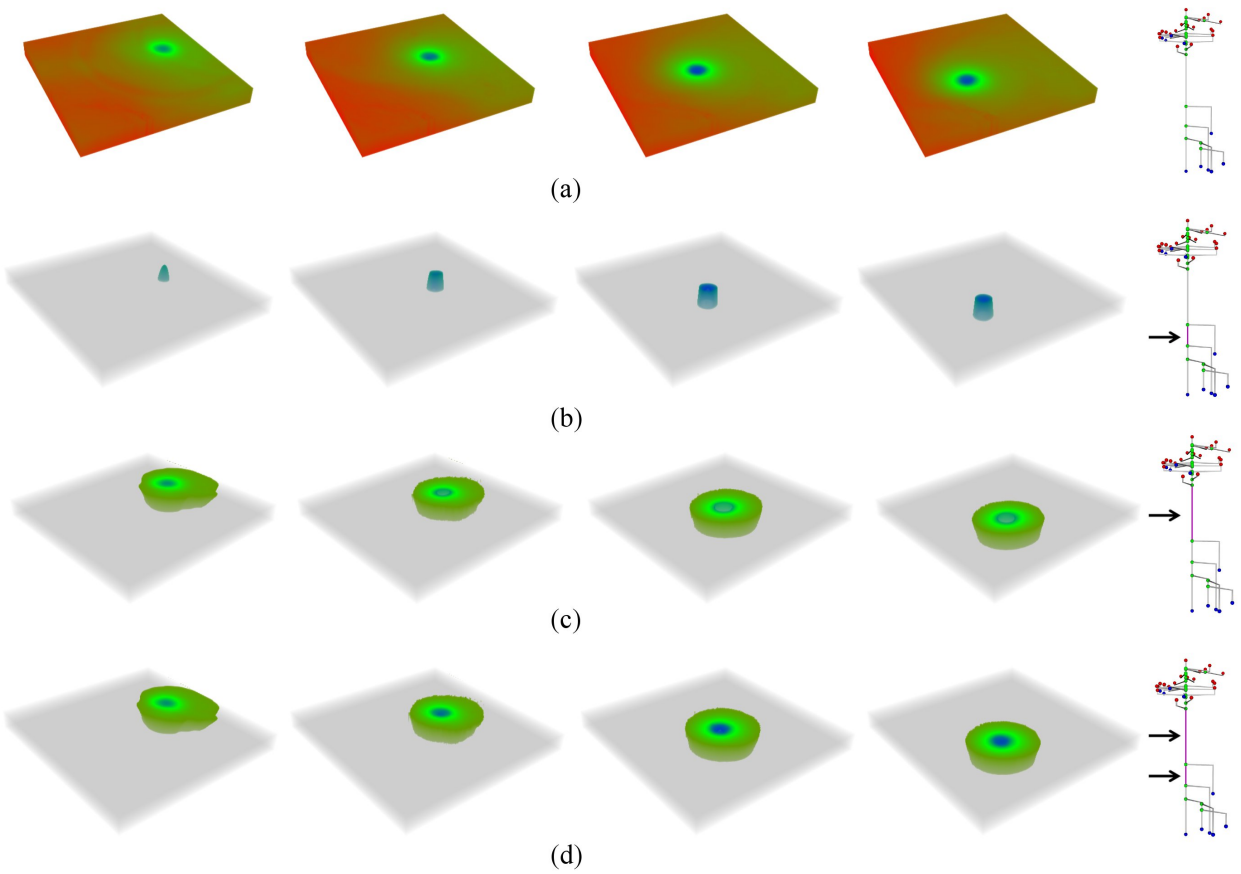


Fig. 14. Exploring the hurricane Isabel data using Reeb graphs. (a) The input is shown as a set of volumes at time steps $1, 16, 32, 40$ and the Reeb graph is shown on the right. (b) A transfer function is designed specifically for the cylinder corresponding to the arc selected in the Reeb graph. This allows highlighting of specific regions in the volume across different time-steps. The highlighted region in the resulting volume rendered image corresponds to the eye of the hurricane at different time steps. (c) The cylinder with the maximum range of function values corresponds to the region surrounding the eye of the hurricane over time. (d) Multiple arcs can be selected to interactively highlight the eye and the surrounding region.

[25] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci, "Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1177–1184, 2009.
[26] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas, "Robust on-line computation of reeb graphs: simplicity and speed," *ACM Trans. Graph.*, vol. 26, no. 3, p. 58, 2007.
[27] T. Tung and F. Schmitt, "Augmented reeb graphs for content-based retrieval of 3d mesh models," in *SMI '04: Proc Shape Modeling Intl.*, 2004, pp. 157–166.
[28] H. Doraiswamy and V. Natarajan, "Efficient output-sensitive construction of Reeb graphs," in *Proc. Intl. Symp. Algorithms and Computation*, 2008, pp. 557–568.
[29] P. K. Agarwal, H. Edelsbrunner, J. Harer, and Y. Wang, "Extreme elevation on a 2-manifold," *Disc. Comput. Geom.*, vol. 36, no. 4, pp. 553–572, 2006.
[30] H. Edelsbrunner, *Geometry and Topology for Mesh Generation*. England: Cambridge Univ. Press, 2001.
[31] T. F. Banchoff, "Critical points and curvature for embedded polyhedral surfaces," *Am. Math. Monthly*, vol. 77, pp. 475–485, 1970.
[32] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Morse-Smale complexes for piecewise linear 3-manifolds," in *Proc. Symp. Comput. Geom.*, 2003, pp. 361–370.
[33] Y. Matsumoto, *An Introduction to Morse Theory*. Amer. Math. Soc., 2002, translated from Japanese by K. Hudson and M. Saito.
[34] G. Reeb, "Sur les points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique," *Comptes Rendus de L'Académie ses Séances, Paris*, vol. 222, pp. 847–849, 1946.
[35] V. Pascucci and K. Cole-McLaughlin, "Efficient computation of the topology of level sets," in *Proc. IEEE Conf. Visualization*, 2002, pp. 187–194.
[36] E. P. Mücke, "Shapes and implementations in three-dimensional geometry," Ph.D. dissertation, Dept. Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1993.
[37] H. Edelsbrunner, D. Letscher, and A. Zomorodian., "Topological persistence and simplification," *Disc. Comput. Geom.*, vol. 28, no. 4, pp. 511–533, 2002.
[38] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli, "The TOPORRERY: computation and presentation of multi-resolution topology," in *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, ser. Mathematics and Visualization. Springer, 2009, pp. 19–40.
[39] I. Fujishiro, Y. Maeda, and H. Sato, "Interval volume: a solid fitting technique for volumetric data display and analysis," in *VIS '95: Proceedings of the 6th conference on Visualization '95*. Washington, DC, USA: IEEE Computer Society, 1995, p. 151.
[40] B. Guo, "Interval set: A volume rendering technique generalizing isosurface extraction," in *VIS '95: Proceedings of the 6th conference on Visualization '95*. Washington, DC, USA: IEEE Computer Society, 1995, p. 3.
[41] B.-S. Sohn, "Topology preserving tetrahedral decomposition of trilinear cell," in *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, 2007, pp. 350–357.
[42] Volvis repository. [Online]. Available: http://www.volvis.org/
[43] Tetview software. [Online]. Available: http://tetgen.berlios.de/tetview.html/
[44] Aim@shape shape repository. [Online]. Available: http://www.aimatshape.net/

Harish Doraiswamy is a Ph.D. candidate at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore. He received his B.E. degree from Visveswaraiah Technological University, and M.E. degree from Indian Institute of Science, both in Computer Science and Engineering. His research interests include scientific visualization and computational topology.

Vijay Natarajan is an assistant professor in the Department of Computer Science and Automation and the Supercomputer Education and Research Centre at the Indian Institute of Science, Bangalore. He received the Ph.D. degree in computer science from Duke University in 2004 and holds the B.E. degree in computer science and M.Sc. degree in mathematics from Birla Institute of Technology and Science, Pilani, India. His research interests include scientific visualization, computational geometry, computational topology, and meshing.