Reeb Graphs: Computation, Visualization and Applications

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE FACULTY OF ENGINEERING

by

Harish D

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2012

To Pati

Acknowledgements

It has been a long and memorable journey at IISc, which began when I enrolled as a Master's student. This journey would not have been possible without the support from my family. I will always be grateful to them for encouraging me to follow my own path. A special thanks to my little niece Aditi for cheering me up whenever I needed it.

I would like to thank my advisor Dr. Vijay Natarajan for introducing me to the field of Visualization and providing valuable guidance and encouragement. Working with Vijay has been a pleasure and has helped me improve both technically and professionally. I would like to sincerely acknowledge his invaluable support in all forms throughout my Ph.D.

I was introduced into the world of research by Prof. Jayant Haritsa. His occasional nudges together with his enthusiasm towards research is what inspired me to pursue a Ph.D. He has been a major influence in my research career, and I will always be grateful for his continued support and encouragement.

It has been possible to meet and interact with a lot of wonderful researchers during my Ph.D. In particular, I would like to thank Dr. Yusu Wang for her valuable advice when working on Topological Saliency. I am grateful to the former chairman of our department, Prof. Narasimha Murty, and our current chairman Prof. Narahari for their support throughout my stay at IISc. I would also like to thank the office staff, Mrs. Lalita and Mrs. Suguna, for efficiently handling all the required paperwork.

I have enjoyed my stay at IISc thanks to my many friends. They were always present either for an afternoon stroll for coffee, random discussions on myriad topics, or for an evening game of cricket. It has also been fun to experience some semi-professional cricket with the IISc cricket team. I will always cherish the memories of the time spent with my friends at IISc.

I was supported by Infosys Technologies Ltd., Bangalore, under the Infosys Fellowship Award from January 2009 to July 2010, and was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award from August 2010. This work was partially supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/048/2007, and partially supported by DST Center for Mathematical Biology, IISc, under Grant SR/S4/MS:419/07.

Publications based on this Thesis

Conference and Journal publications

1. Harish Doraiswamy and Vijay Natarajan, “Efficient output-sensitive construction of Reeb graphs”, *ISAAC ’08: Proc. Intl. Symp. Algorithms and Computation, LNCS 5369, Springer-Verlag*, 2008, pgs. 557-568.

2. Harish Doraiswamy and Vijay Natarajan, “Efficient algorithms for computing Reeb graphs”, *Computational Geometry: Theory and Applications*, 42, 2009, pgs. 606-616.

3. Harish Doraiswamy and Vijay Natarajan, “Output-sensitive construction of Reeb graphs”, *IEEE Transactions on Visualization and Computer Graphics*, 18(1), 2012, pgs. 146-159.

4. Harish Doraiswamy and Vijay Natarajan, “Computing Reeb graphs as a union of contour trees”, *IEEE Transactions on Visualization and Computer Graphics*, 19 (2), 2013, pgs. 249-262.

5. Harish Doraiswamy, Nithin Shivashankar, Vijay Natarajan, and Yusu Wang, “Topological Saliency”, *Computers & Graphics*, to appear.

Posters and Videos

1. Harish Doraiswamy, Aneesh Sood, and Vijay Natarajan, “Constructing Reeb graphs using cylinder maps”, *ACM Symposium on Computational Geometry*, Video / Multimedia track, 2010.

2. Harish Doraiswamy and Vijay Natarajan, “Computing Reeb graphs as a union of contour trees”, Poster at *IEEE Visualization*, 2011.

Abstract

Level sets are extensively used for the visualization of scalar fields. The Reeb graph of a scalar function tracks the evolution of the topology of its level sets. It is obtained by mapping each connected component of a level set to a point. The Reeb graph and its loop-free version called the contour tree serve as an effective user interface for selecting meaningful level sets and for designing transfer functions for volume rendering. It also finds several other applications in the field of scientific visualization.

In this thesis, we focus on designing algorithms for efficiently computing the Reeb graph of scalar functions and using the Reeb graph for effective visualization of scientific data. We have developed three algorithms to compute the Reeb graph of PL functions defined over manifolds and non-manifolds in any dimension. The first algorithm efficiently tracks the connected components of the level set and has the best known theoretical bound on the running time. The second algorithm, utilizes an alternate definition of Reeb graphs using cylinder maps, is simple to implement and efficient in practice. The third algorithm aggressively employs the efficient contour tree algorithm and is efficient both theoretically, in terms of the worst case running time, and practically, in terms of performance on real-world data. This algorithm has the best performance among existing methods and computes the Reeb graph at least an order of magnitude faster than other generic algorithms.

We describe 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. We also employ the Reeb graph in four different applications – surface segmentation, spatially-aware transfer function design, visualization of interval volumes, and interactive exploration of time-varying data.

Finally, we introduce the notion of topological saliency that captures the relative importance of a topological feature with respect to other features in its local neighborhood. We integrate topological saliency with Reeb graph based methods and demonstrate its application to visual analysis of features.

Contents

List of Tables

List of Figures

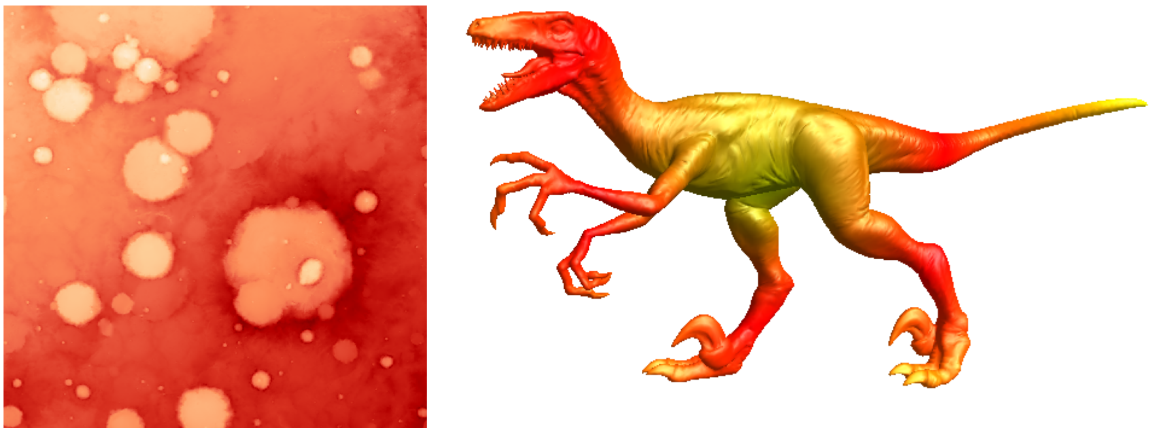
Chapter 1

Introduction

Visual interpretation of scientific data enhances comprehension of the data. Scientific data is typically obtained from scientific instruments such as sensors and imaging devices, and from simulations. Advances in computational resources has primarily resulted in increasing the size of data that is available for analysis. Constructing a meaningful abstraction of such data results in not just decreasing its size, but also helps ease the analysis. In this thesis, we study one such abstraction known as Reeb graphs, which abstracts the topology of the underlying data. We explore techniques for the efficient construction of Reeb graphs and its effective application to visualize scientific data.

1.1 Scalar functions and level sets

A function is a unique mapping from members of a domain set to members of a co-domain set. Scientific data is often represented as scalar functions that assign real values to points in a geometric domain. Figure 1.1 and Figure 1.2 shows two examples of two-dimensional and three-dimensional scalar functions respectively. Figure 1.1(a) shows a region of Mars, the scalar value at each point being the height of the terrain, and Figure 1.1(b) shows the average geodesic function defined on the surface of a triangulated mesh as the average distance from a given point to all other points on the surface. Figure 1.2(a) shows a volume rendering of the CT scan of the head of the visible human male. The scalar value at each point in the domain is



(a) (b)

Figure 1.1: Example of 2-dimensional scalar functions. **(a)** The height function defined on a region on the surface of Mars. **(b)** The average geodesic function defined on a surface mesh representing a raptor.

its radiodensity. Figure 1.2(b) shows a volume rendering of the fuel data set [3] obtained from a simulation of combustion of fuel injected into a fuel chamber. The scalar value at each point in the combustion chamber is the density of fuel at that point. The scalar functions in these figures are visualized by mapping the function values to color and opacity.

A level set consists of all points where the function attains a given value, called the iso-value. Level sets are used extensively to visualize three and higher dimensional scientific data. Visualization approaches using level sets focus on choosing a suitable iso-value that captures the interesting features of the data. The level set of a three-dimensional scalar function forms a surface called the isosurface. Figure 1.3 shows a few interesting isosurfaces of the scalar functions shown in Figure 1.2.

As the data becomes more detailed and feature-rich, it becomes difficult to search for a meaningful set of iso-values. Thus, there arises a need for methods to automatically analyze and extract important features from the data.

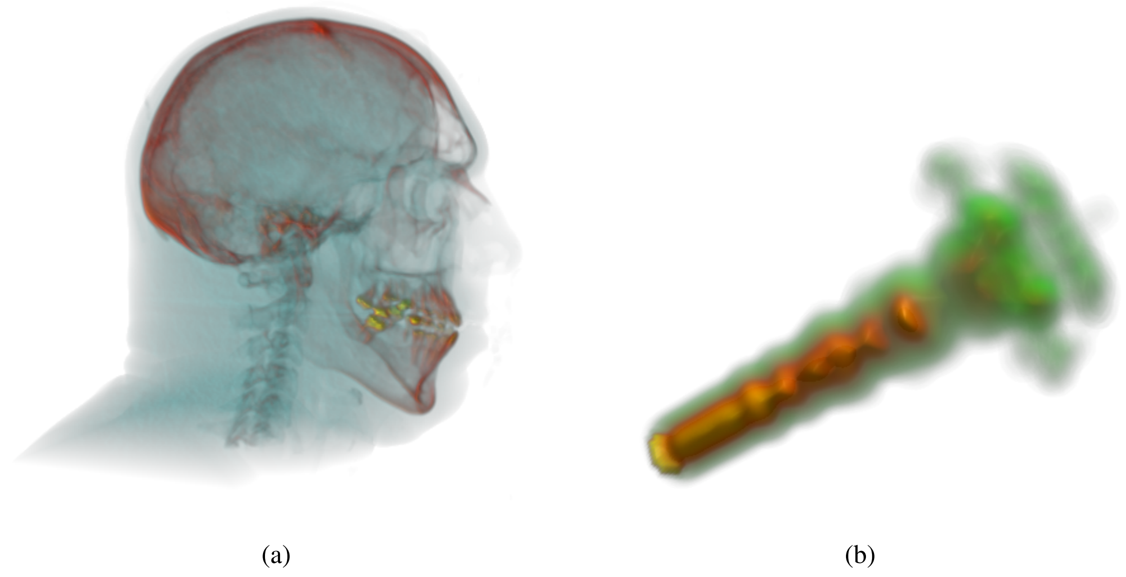


Figure 1.2: Example of 3-dimensional scalar functions. **(a)** Head of the visible human male. **(b)** Simulation of combustion of fuel injected into a fuel chamber.

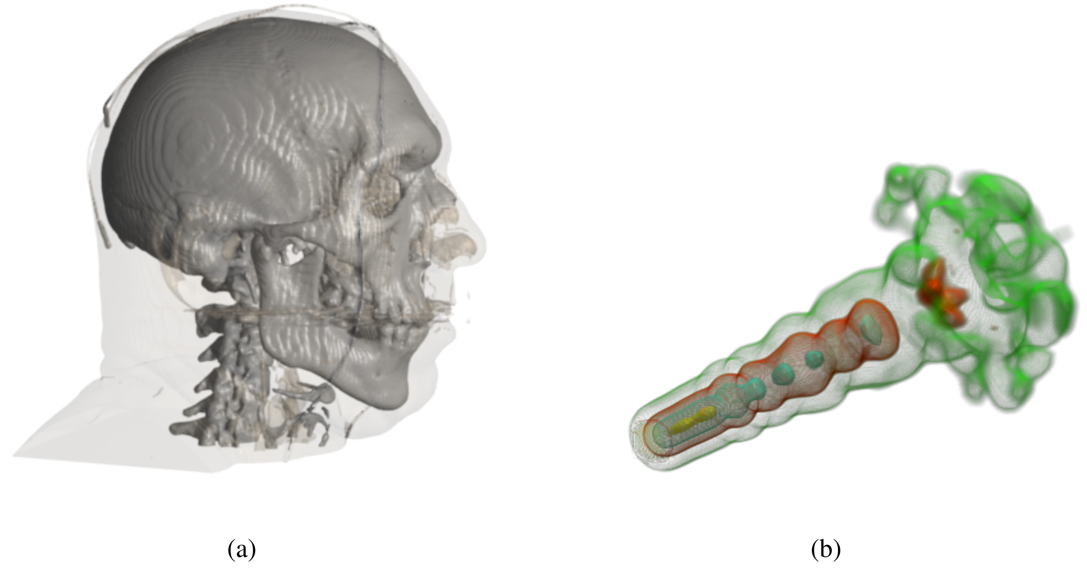


Figure 1.3: Use of isosurfaces for visualizing scientific data. **(a)** Two isosurfaces corresponding to the skin and skull of the human are shown. **(b)** Four isosurfaces of the fuel data are shown.

1.2 Reeb graph

The Reeb graph tracks topology changes in level sets of a scalar function, and therefore provides a good abstraction for the given data. The abstract representation of the level set topology in the Reeb graph facilitates the development of methods for modeling objects and visualizing scientific data. Reeb graphs and their loop-free version, called contour trees, have a variety of applications including computer aided geometric design [46,61,67,78], topology-based shape matching [42], topological simplification and cleaning [16, 36, 66, 77], surface segmentation and parametrization [41,51,80], and efficient computation of level sets [73]. The Reeb graph serves as an effective user interface for selecting meaningful level sets [5, 14], for designing transfer functions for volume rendering [33, 65, 75, 81] and for exploring high dimensional data [37,53].

Rapidly increasing data sizes and the interactivity requirement in the above-mentioned applications necessitate the development of algorithms for fast computation of Reeb graphs that are capable of handling relatively large input sizes. Further, in several cases, the domain is not simply connected, is non-manifold, and may be high-dimensional. While an efficient and fast algorithm is available for computing contour trees in all dimensions, such an algorithm for computing Reeb graphs has remained elusive.

1.3 Contributions

In this thesis, we target on the following three aspects that are essential for the use of Reeb graphs for visualization of scientific data.

1. Efficient computation of Reeb graphs

2. Effective visualization of Reeb graphs

3. Applications of Reeb graphs

1.3.1 Computation of Reeb graphs

The primary focus of this thesis is to establish an efficient and fast algorithm to compute Reeb graphs. To that effect, we propose the following three algorithms that computes the Reeb graph of a scalar function defined on a mesh in any dimension.

- **The Sweep algorithm** [20]: This algorithm uses an efficient tree-cotree decomposition to maintain connected components of level sets of a 3-dimensional input and computes the Reeb graph in $O(n \log n + n \log^2 g)$ time, where n is the number of triangles in the input mesh and g is the maximum genus over all level sets of the scalar function. We extend this approach to maintain connected components of level sets in higher dimension using a dynamic connectivity algorithm. This results in an $O(n \log^2 n)$ time algorithm that computes the Reeb graph of scalar functions in any dimension. Using a randomized approach to maintain the above data structures improves the time complexity to $O(n \log n + n \log g (\log \log g)^3)$ expected time for a 3-dimensional input, and $O(n \log n (\log \log n)^3)$ expected time for an input in any dimension. This was the first sub-quadratic time algorithm and is a significant improvement over the previously known $O(n^2)$ algorithm. It currently has the best known theoretical bound[1] on the running time.

- **The Cylinder Map algorithm** [21]: The use of complex data structures to maintain level sets in the sweep algorithm does not result in an efficient implementation. The cylinder map algorithm avoids explicitly maintaining level sets by using an alternate definition of the Reeb graph that considers equivalence classes of level sets instead of individual level sets. This approach results in an algorithm that is simple to implement and has a running time of $O(n + l + t \log t)$, where t is the number of critical points of the input, and l is the size of all critical level sets. We also show that this algorithm performs upto an order of magnitude faster than the previously known generic algorithms.

- **The Recon algorithm** [22]: This algorithm builds upon the cylinder map algorithm by aggressively employing the efficient algorithm that computes loop-free Reeb graphs. It

[1] at the time of writing this thesis

splits the input into a set of subvolumes that have loop-free Reeb graphs, and constructs the Reeb graph of the input by merging the Reeb graphs of all the subvolumes. The algorithm has a running time of $O(N\log N + sn)$, which is close to the lower bound $\Omega(N\log N + n)$. Here, N is the number of vertices and s is the number of saddles in the input. This approach outperforms current generic algorithms by a factor of up to two orders of magnitude, and has a performance on par with the state-of-the-art algorithms that are catered to restricted classes of input. The algorithm is also amenable to handle large data that do not fit in memory.

1.3.2 Visualization of Reeb graphs

Effective visualization of Reeb graphs is crucial for its application to noisy or feature-rich data. We accomplish this by addressing the following two important issues.

- **Simplification**: We develop a method to simplify the Reeb graph based on the notion of extended topological persistence [4] that removes short leaves and cycles in the graph. The advantage of this approach is that, it easily extends to handle other notions of importance such as hyper-volume [14].

- **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 develop a method to generate an embedded layout of the Reeb graph such that the embedding lies within the interior of the volume.

1.3.3 Application of Reeb graphs

We demonstrate the use of Reeb graphs in four applications – 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.

We also introduce *topological saliency* [23], a notion of importance that captures the relative importance of a topological feature with respect to other features in its local neighborhood. We demonstrate the advantage of using topological saliency together with the Reeb graph in several applications including key feature identification, scalar field simplification, and feature clustering.

1.4 Organization

This thesis is organized as follows. Chapter 2 introduces the necessary definitions and Chapter 3 surveys existing literature on Reeb graphs. Chapters 4-6 describe the algorithms for computing Reeb graphs. Chapter 7 presents experimental results and comparative analysis with existing algorithms. Chapter 8 describes techniques for simplification and visualization of Reeb graphs, and Chapter 9 discusses four applications of Reeb graphs. Chapter 10 introduces the notion of topological saliency and Chapter 11 concludes this thesis.

Chapter 2

Background

In this chapter, we briefly introduce some of the necessary definitions and refer the reader to books on computational topology, algebraic topology, and Morse theory [25, 39, 48, 50] for more detailed definitions and discussions of these concepts.

2.1 Scalar functions and manifolds

A *scalar function* is a function that maps points in a spatial domain to the set of real values $\mathbb{R}$. One particular class of spatial domains we consider in this thesis is called manifolds.

Consider a function $f : X \to Y$ between two spaces X and Y. The function f is a *homeomorphism* if it is bijective, continuous and the inverse of f is continuous. The space X is *homeomorphic* to the space Y, if there exists a homeomorphism between the two spaces. A *d-manifold* is defined as a space in which every point has a neighborhood that is homeomorphic to $\mathbb{R}^d$. Intuitively, a d-manifold locally resembles the d-dimensional Eucleadean space. Examples of 1- and 2-manifolds are shown in Figure 2.1.

A particular function of interest used in this thesis is the *height function*. The height function of a point along a given line is equal to the projection of that point onto the line. Unless otherwise specified, we use the y-axis to define the height function throughout this thesis.

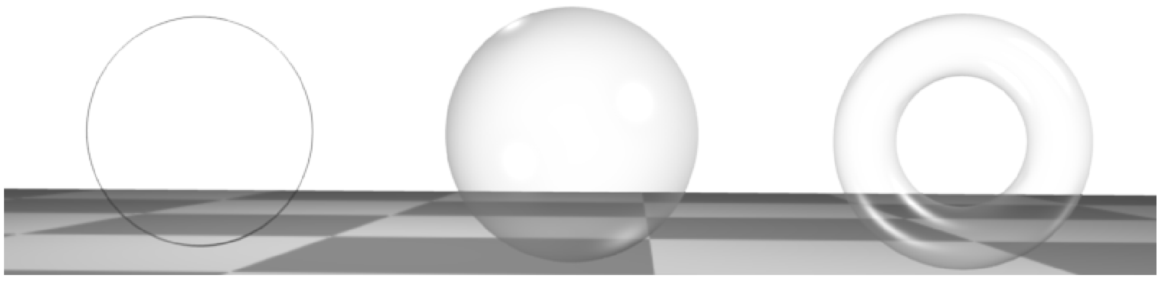


Figure 2.1: Examples of 1- and 2-manifolds. A circle (left) is a 1-manifold. A hollow sphere (center) and a hollow torus (right) are 2-manifolds.

2.2 Critical points and Morse functions

Let $\mathbb{M}$ denote a d-manifold with or without boundary. Given a smooth, real-valued function $f : \mathbb{M} \to \mathbb{R}$ defined on $\mathbb{M}$, the *critical points* of f are exactly where the gradient becomes zero. A critical point c_p is *non-degenerate* if the Hessian at c_p is non-singular.

The function f is called a *Morse function* if it satisfies the following conditions [17]:

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

Critical points of a Morse function can be classified based on the behavior of the function within a local neighborhood.

2.3 Level set topology

The preimage $f^{-1}(c)$ of a real value c is called a *level set*. The real value c is called the *iso-value*. The level set of a regular value is a $(d-1)$-manifold with or without boundary, possibly containing multiple connected components. A *sub-level set* of a real value c is the preimage of the interval $(-\infty, c]$, while the *super-level set* of c is the preimage of the interval $[c, +\infty)$. The *interval volume* between two real values c_1 and c_2 is defined as the preimage of the interval

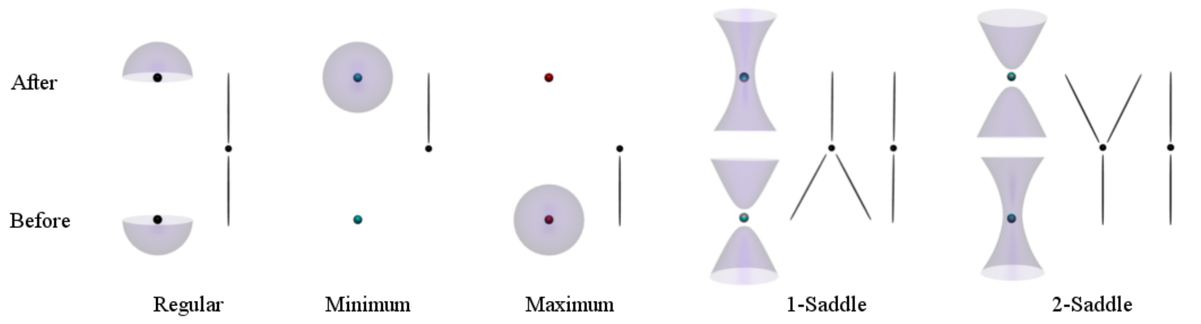


Figure 2.2: Isosurfaces before ($f^{-1}(c-\varepsilon)$) and after ($f^{-1}(c+\varepsilon)$) passing through a point with function value c and the structure of the Reeb graph at the corresponding node. Topology of the isosurface changes when it evolves past a critical point. Genus modifying saddles and regular points are optionally included into the Reeb graph as degree two nodes.

$[c_1, c_2]$. 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 [48].

Consider the case when $d = 3$. A level set of a three-dimensional function is called an *isosurface*. Figure 2.2 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 2.2 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.

2.4 Reeb graphs

The *Reeb graph* of f is obtained by contracting each connected component of a level set to a point [57]. 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 of a simply connected domain has no loops and is called a *contour tree*.

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 2.3 shows the Reeb graphs

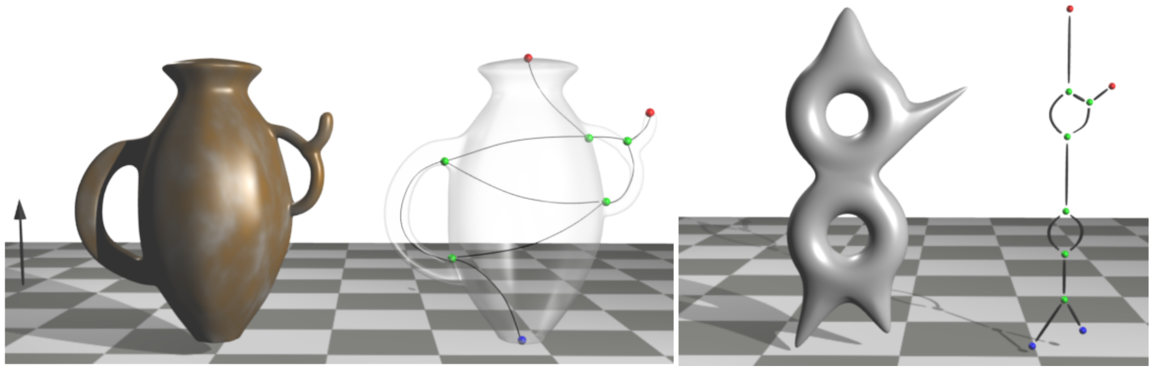


Figure 2.3: Reeb graphs of the height function defined on a solid vase and a solid 2-torus respectively.

of the height function defined on a solid vase and a solid 2-torus respectively. Nodes of the Reeb graph have degree one, three, or greater. Figure 2.2 illustrates the local structure of the Reeb graph at various types of nodes present in a 3-dimensional scalar function. Nodes corresponding to minima and maxima have degree one. A node that corresponds to a *simple saddle* has degree three if the saddle merges or splits level set components. Other simple saddles do not alter the number of level set components and are optionally included into the Reeb graph as degree-2 nodes. Higher degree nodes correspond to *multi-saddles* that are not present in Morse functions. Regular vertices and degree-2 saddles are often inserted into the Reeb graph as degree-2 nodes to obtain the *augmented Reeb graph*.

2.5 Piecewise-linear functions

Scientific data is typically available as a sample over a domain of interest. The domain is represented using a mesh and the function is interpolated within elements of the mesh. Banchoff [6] and Edelsbrunner et al. [26] extend the ideas defined on smooth functions to PL functions, which we cover in this section.

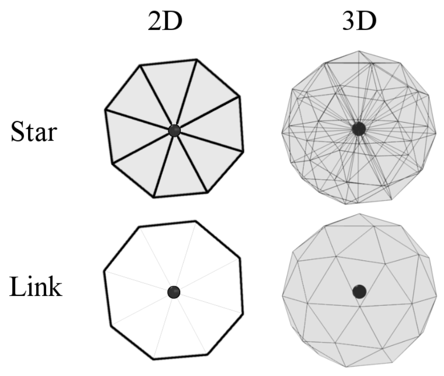


Figure 2.4: Local neighborhood of a vertex in a 2D and 3D mesh. The star of the vertex represents its local neighborhood and consists of all simplicies incident on it. The link of a vertex in 2D is a triangulation of a circle, and in 3D, it is a triangulation of a sphere.

2.5.1 Simplicial complex

A *d-simplex* σ is the convex hull of $d+1$ affinely independent points. A simplex τ is a *face* of σ if it is the convex hull of a subset of the $d+1$ points and is denoted as $\tau \leq \sigma$. A simplex σ is called the *coface* of τ if τ is a face of σ. A simplicial complex K is a finite collection of non-empty simplices that satisfies the following two properties:

1. $\sigma \in K$ and $\tau \leq \sigma$ implies $\tau \in K$
2. $\sigma_1, \sigma_2 \in K$ implies that $\sigma_1 \cap \sigma_2$ is either empty or a face of both σ_1 and σ_2.

The input scalar function corresponding to the scientific data is a triangulated mesh represented by a simplicial complex K together with a PL function $f : K \to \mathbb{R}$. The function is defined on the vertices of K. Any point x in the interior of a d-simplex can be uniquely represented as a convex sum $x = \sum_{i=0}^{d} \lambda_i v_i$, where v_i are the vertices of the simplex and $\sum_{i=0}^{d} \lambda_i = 1$. The function at x is obtained by linearly interpolating f within the simplex, that is, $f(x) = \sum_{i=0}^{d} \lambda_i f(v_i)$.

The neighborhood of a vertex is defined by its star and link. The *star* of a vertex v consists of the set of cofaces of v. All simplices in the star in which the function values are greater than at v constitute the *upper star*. All simplices in the star in which the function values are lower than at v constitute the *lower star*. The *link* of a vertex v is the set of all faces of simplicies of its star that are disjoint from v. It consists of all vertices adjacent to v 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*. Figure 2.4 shows the star and link for vertices in two- and three-dimensional meshes.

2.5.2 Critical points in PL functions

Banchoff [6] and Edelsbrunner et al. [26] describe a combinatorial characterization for critical points of a PL function, which are always located at vertices of the mesh. Critical points are characterized by the number of connected components of the lower and upper links. 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*.

Consider a sweep of the input in decreasing order of function value. A vertex is a *join saddle* if two level set components merge at that vertex during the sweep. It is a *split saddle* if a single level set component splits into two components at that vertex. Other simple saddles that have at most two lower link and two upper link components, and do not modify the number of connected components of the level set. In the context of Reeb graphs, we are interested only in critical points that modify the number of level set components. Figure 2.5 shows the characterization of critical points using the upper and lower links for a given vertex.

Degeneracy in PL functions results in the presence of saddles that have degree greater than three in the Reeb graph. The higher degree saddles can be split into multiple simple saddles as shown by Edelsbrunner et al. [26] and Carr et al. [12]. The conditions for a Morse function typically do not hold in practice for PL functions. Higher degree saddles are indeed present in practice but they can be handled explicitly. Simulated perturbation of the function [27, Section 1.4] ensures that no two critical values are equal. The simulated perturbation imposes a total order on the vertices, which helps in consistently identifying the vertex with the higher function value between a pair of vertices.

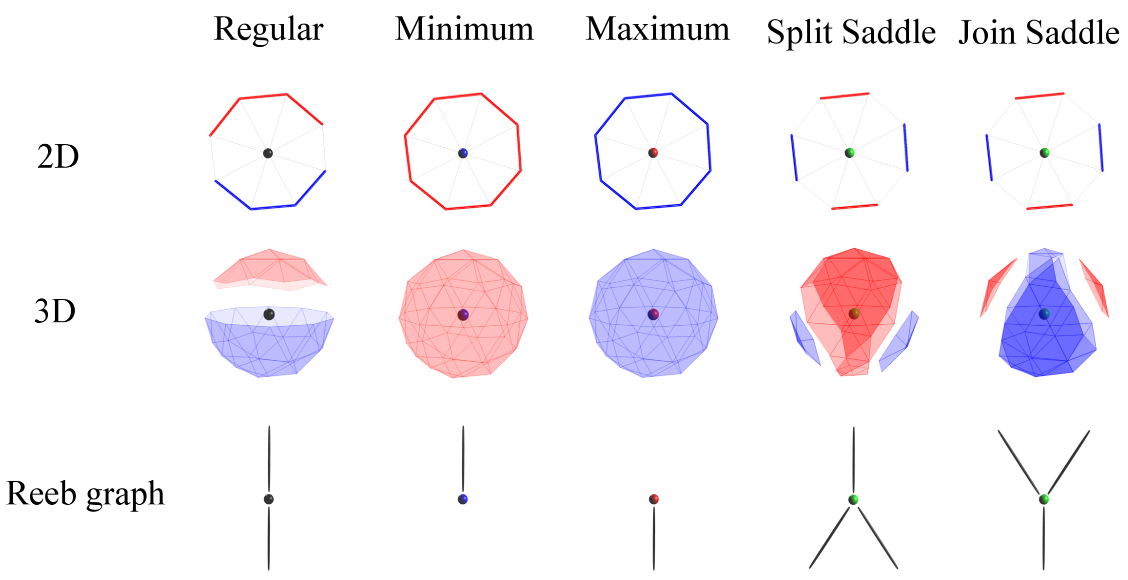


Figure 2.5: Classifying a vertex as regular, minimum, maximum or saddle using the topology of its local neighborhood. The lower link of a vertex is colored blue; its upper link is colored red. The structure of the Reeb graph is shown in the bottom row. The connectivity of the level sets further classifies the saddle as a split or join saddle.

2.6 Topological persistence

Topological persistence measures the topological importance of the critical points present in a scalar function [4, 29]. Consider a sweep of the input scalar function f in increasing order of function value. As mentioned earlier in Section 2.3, topology of the level set changes at critical points during this sweep. In particular, at a critical point, either new topology is created or some topology is destroyed, where topology is quantified by a class of 'cycles'. A non-bounding *k-cycle* is a k-dimensional simplicial complex with zero boundary that does not bound a $k+1$-dimensional simplicial complex. A 0-dimensional cycle represents a connected component, a 1-dimensional cycle is a loop that represents a tunnel, and a 2-dimensional cycle bounds a void. A critical point is a creator if new topology appears and a destroyer otherwise. It turns out that one can pair up each creator c_1 uniquely with a destroyer c_2 which destroys the topology created at c_1. The persistence value of c_1 and c_2 is defined as $f(c_2) - f(c_1)$, which intuitively indicates the lifetime of the feature created at c_1, and thus the importance of c_1 and c_2. In case of Reeb graphs, the set of creators and destroyers are restricted to critical points where the number of connected components of the level set change.

2.7 Input Representation

The algorithms presented in this thesis use a modified version of the triangle-edge data structure [52] to represent the input simplicial complex together with the PL function defined on it. In this section, we provide a brief overview of this data-structure, and describe the modifications made to it in order to support higher dimensional manifolds, as well as non-manifolds.

2.7.1 Triangle-edge data structure

The triangle-edge data structure essentially stores the set of triangles in the input along with information about their adjacencies. Consider the set of edges of the input. Each edge has a set of triangles incident on it. This set, called the *triangle fan*, can be naturally ordered when the input is embedded in the Euclidean space $\mathbb{R}^3$. The data structure maintains this ordering for each edge. This is accomplished by storing the references of the previous and next triangle in the ordering for each triangle-edge pair. In addition to the above ordering, we additionally store for each vertex the scalar value associated with it.

2.7.2 d-manifold input

The data structure described above stores the triangles of the input ordered around an edge. While this is feasible for two and three-dimensional input, there is no natural order among triangles for higher dimensional data. When working with higher dimensional data sets, our algorithms use the triangle-edge adjacencies only to identify the edges of a level set. Since this operation does not depend on the ordering of the triangles around an edge, the triangles are arbitrarily ordered around each edge.

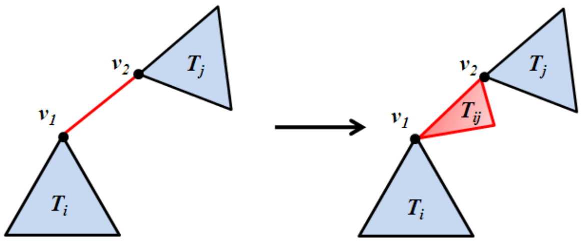


Figure 2.6: Replace edge v_1v_2 that is not incident on any triangle with triangle T_{ij}. Our algorithms works on this modified input to compute the Reeb graph. The function value at the new vertex of T_{ij} is set equal to the average of $f(v_1)$ and $f(v_2)$.

2.7.3 Non-manifold input

The algorithms presented in this thesis expects the input to be a collection of triangles. While this is true for a manifold input, it is possible for non-manifolds to have isolated edges incident of vertices. Figure 2.6 shows such an example where an edge connects two triangles of the input. In such situations, an edge that has no triangle incident on it is replaced by a triangle as shown in Figure 2.6. The function value at the additional vertex is set to be 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 a regular. Note that the size of the input remains unaffected asymptotically.

Several stages of the algorithms presented in this thesis require a traversal of the star of a vertex. The star of a vertex in a manifold mesh can be obtained by traversing adjacent triangles in the triangle-edge data structure, since it consists of a single component. However, the star of a vertex in a non-manifold input can consist of multiple components. Figure 2.6 shows two examples of such vertices (v_1 and v_2). The star cannot be computed in such cases from the triangle adjacencies. Hence, all triangles in the star of each vertex is also stored in the triangle-edge data structure.

Chapter 3

Related Work

In this chapter, we briefly survey the current literature pertaining to the construction, simplification and visual presentation of Reeb graphs.

3.1 Reeb graph computation algorithms

Several algorithms have been proposed for computing the Reeb graph of a scalar function. In this section we restrict the discussion to those algorithms that produce provably correct Reeb graphs. We categorize the algorithms based on how they partition the input domain to analyze the topology of the level sets. Table 3.1 summarizes the properties and worst case running time of the best known Reeb graph computation algorithms. It also compares existing algorithms with the algorithms presented in this thesis.

The Reeb graph of a scalar function defined on a simply connected domain is called a contour tree. Carr et al. [12] describe an elegant $O(N \log N + n\alpha(n))$ algorithm for computing contour trees that works in all dimensions. Here, N is the number of vertices and n is the number of triangles in the input. This algorithm uses a series of union-find [18] operations to track the connectivity of the super-level sets and sub-level sets, and constructs a join tree and split tree, respectively. These two trees are merged to generate the contour tree. Chiang et al. [15] propose an output sensitive approach that first finds all component critical points using local neighborhoods and connects these critical points using monotone paths to obtain the join

Algorithm	Input Type	Time Complexity	Approach	Notes
Carr et al. 2003 [12]	simply connected d-dimensional simplicial complex	$O(N \log N + n\alpha(n))$	track components of sub-level sets and super-level sets using a series of union-find operations	1. Computes contour trees. 2. Requires data in memory.
Cole-Mclaughlin et al. 2004 [17]	2-manifolds	$O(n \log n)$	sweep and explicitly maintain level sets	1. Does not extend to 3- and higher dimensions. 2. Requires data in memory.
Pascucci et al. 2007 [55]	arbitrary simplicial complex	$O(n^2)$	explicitly maintain level sets	1. Has the best performance for 2D data. 2. Capable of handling large data sizes.
Patanè et. al. 2009 [56]	2-manifold	$O(ns)$	split and compute	1. Time complexity degenerates to $O(n^2)$ in the worst case. 2. It requires data in memory.
Tierny et al. 2009 [70]	3-manifolds embedded in $\mathbb{R}^3$	$O(N \log N + hn)$	split and compute	1. Has the best performance for such input. 2. Time complexity degenerates to $O(n^2)$. 3. Requires data in memory.
Harvey et al. 2010 [38]	arbitrary simplicial complex	$O(n \log n)$ expected	collapse triangles	1. Requires data in memory.
SWEEP [This thesis]	3-manifolds arbitrary simplicial complex	$O(n \log n + n \log^2 g)$ $O(n \log n + n \log g (\log \log g)^3)$ expected $O(n \log^2 n)$ $O(n \log n (\log \log n)^3)$ expected	sweep and explicitly maintain level sets	**1. Has best known theoretical bound on running time**. 2. Requires data in memory.
CMAP [This thesis]	arbitrary simplicial complex	$O(n + l + t \log t)$	split and compute	**1. Proposes a new approach using *Cylinder Maps***. 2. Requires data in memory. 3. Time complexity degenerates to $O(n^2)$ in the worst case.
RECON [This thesis]	arbitrary simplicial complex	$O(N \log N + sn)$	split and compute	**1. Outperforms existing algorithms**. **2. Works with large data**. 3. Time complexity degenerates to $O(n^2)$ in the worst case.

Table 3.1: Comparison of various algorithms for computing Reeb graphs based on the type of input they can handle, worst case running time, and approach. N is the number of vertices in the input, n is the number of triangles, t is the number of critical points, s is the number of saddles, l is the total size of all critical level sets, g is the maximum genus over all level sets, and h is the number of loops in the Reeb graph.

and split trees. This algorithm has a running time of $O(t \log t + n)$, where t is the number of critical points of the input.

Early algorithms for computing Reeb graphs followed the direct approach of tracking its level sets with increasing / decreasing function values during a sweep of the input. Shinagawa and Kunii proposed the first algorithm for constructing the Reeb graph of a scalar function defined on a triangulated 2-manifold [60] in $O(n^2)$ time. This algorithm explicitly tracks connected components of the level sets. Cole-Mclaughlin et al. [17] store the level sets using balanced search trees and improved the running time to $O(n \log n)$.

Pascucci et al. [55] propose an online algorithm that constructs the Reeb graph for streaming data. Their algorithm takes advantage of the input coherence to construct the Reeb graph efficiently. In a streaming model, where triangles are processed during a single pass through 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. The operations performed in order to keep track of the incomplete Reeb graph results in an $O(n^2)$ running time. Even though the algorithm has a $O(n^2)$ behavior in the worst case, it performs very well for two-dimensional scalar functions. However, the optimizations that result in fast incremental construction of Reeb graphs for 2D data do not provide a performance benefit in higher dimensions.

Harvey and Wang [38] propose a randomized algorithm that computes the Reeb graph of an arbitrary simplicial complex. They repeatedly collapse all triangles constituting the level set component of randomly chosen vertices resulting in a reduced input whose Reeb graph is equal to that of the original scalar function. This algorithm has an expected running time of $O(n \log n)$.

Other recent algorithms follow an approach that explicitly split the input, compute the Reeb graph for each subdomain, and stitch the graphs together to obtain the Reeb graph of the input. Patanè et. al. [56] focus on 2-manifolds and propose a contouring approach to compute the Reeb graph in $O(ns)$ time, where s is the number of saddles in the input.

Tierny et al. [70] perform a surgery on a 3-manifold domain that cuts through all handles on the domain's boundary, thereby reducing the problem to the computation of contour trees. This approach leads to a very efficient algorithm that computes the Reeb graph in $O(N \log N + hn)$ time, where h is number of loops in the Reeb graph. This algorithm however works only on 3-manifolds that are embedded in $\mathbb{R}^3$. Even though the worst case running times of this algorithm degenerates to $O(n^2)$, it has the best performance among all algorithms for a 3-manifold input. This can be attributed to the fact that this method achieves significant speed up by reducing the problem to that of computing contour trees. The above two approaches explicitly stores the split domain. This causes the memory required to increase linearly with the number of saddles and loops respectively, and results in a large memory overhead in practice when several triangles span a large function range and thus are repeatedly split.

Other algorithms for computing Reeb graphs follow a sample based approach that produces potentially inaccurate results [42, 72]. We refer the reader to the several surveys [7–10] for a detailed discussion of these approaches.

3.2 Simplification of Reeb graphs

Effective presentation of Reeb graphs requires the size of the graph to be relatively small, which is generally not the case with real world data due to the presence of noise. This indicates a need for meaningful methods to simplify the Reeb graph. Takahashi et al. [65] propose a method of simplifying contour trees that replaces degree-3 and degree-2 saddles from the contour tree based on the height (or persistence) of the edges adjacent on them. They extend this method in [66] to use the hyper-volume of the edges as a simplification measure. Carr et al. [14] extend this approach to preserve local information such as isosurface seeds and to compute arbitrary geometric measures of importance. They combine this with the flexible isosurface interface [13] which allows users to explore the dataset interactively.

Pascucci et al. [54] construct and store the contour tree in a multi-resolution data structure. This data structure stores the contour tree as a set of branches, which imposes a hierarchy and allows for an easy implementation of the simplification procedure described in [14].

Hilaga et al. [42] simplify the Reeb graph of surface meshes by iteratively constructing a multi-resolution Reeb graph. They uniformly divide the input function into a set of intervals and use it to partition the input mesh into a set of interval volumes. The Reeb graph is then computed by tracking the connectivity of the components in this set. The level of coarseness is defined by the size of these intervals. The coarsest level corresponds to the entire input resulting in a single arc. The finest level results in the Reeb graph that correctly tracks components of level sets. This method strongly depends on the use of persistence as the simplification measure, and extension to geometric measures such as hyper-volume is therefore difficult.

Pascucci et al. [55] propose a method to simplify the Reeb graph based on the notion of extended persistence [4]. In addition to the traditional pairing of saddle-extremum pair, the extended persistence algorithm additionally pairs split-saddles with join-saddles representing a loop. The simplification procedure first computes the set of persistence pairs, and then cancels these pairs in increasing order of persistence. This process, however, does not guarantee that the pairs canceled form a single arc in the Reeb graph, due to which storing the Reeb graph in a multi-resolution manner is non-trivial. Moreover, implementing an interactive simplification interface, similar to that provided by contour tree simplification methods, becomes problematic.

3.3 Visual presentation of Reeb graphs

To the best of our knowledge, there exists no work on the visual layout of Reeb graphs. There are, however, some results available on the design of visual presentation of contour trees. We discuss this now.

Pascucci et al. [54] introduced the toporrery layout for displaying the contour tree. The contour tree is first hierarchically decomposed into a set of branches. These branches are then laid out radially around the root branch using L-shaped edges. The height of the L-shaped edges correspond to the function values of its end points. This layout not only provides an intuitive 3D orientation, but also serves as a good user interface to interact with the domain.

Weber et al. [76] introduced the concept of topological landscapes that provides an intuitive view of the data by displaying its topological features, abstracted by the contour tree, as a terrain. These landscapes were able to indicate both the branch hierarchy and the volume of the corresponding branch. Harvey et al. [37] extend this idea to generate a collection of landscapes from which the user can explore and identify an appropriate landscape.

Heine et al. [40] recently proposed an algorithm for drawing contour trees in a plane, that tries to minimize edge crossing while maintaining the branch hierarchy. Their method also allows incorporating geometric properties, such as volume of a branch, into the layout.

Chapter 4

The Sweep Algorithm

In this chapter, we present an algorithm that computes the Reeb graph of PL functions defined on three and higher dimensional simplicial meshes. The algorithm essentially follows from the definition of Reeb graphs. It computes the Reeb graph by tracking the evolution of level set components during a sweep of the input function.

4.1 Algorithm Outline

The level set of a PL function f defined on a d-dimensional mesh is a $(d-1)$-dimensional mesh. We are interested in tracking connected components of the level set. This is captured in the 1-skeleton (vertices and edges) of the level set. Therefore, it is sufficient to store edges and vertices of the level set. Vertices and edges that constitute a level set lie within edges and triangles of the input. Topology of a level set changes only when the sweep passes through critical points of f, which are restricted to vertices of the mesh [6,26].

The algorithm proceeds by processing a sequence of events during the sweep. An event is triggered when the level set passes through a vertex. The first step of the algorithm sorts the vertices on increasing function value. The event list is then populated with a set of events that correspond to the set of vertices to be processed. Processing an event includes updating the representation of the level set, its connected components and the Reeb graph. The algorithm maintains the level set at an isovalue infinitesimally above the function value of the processed

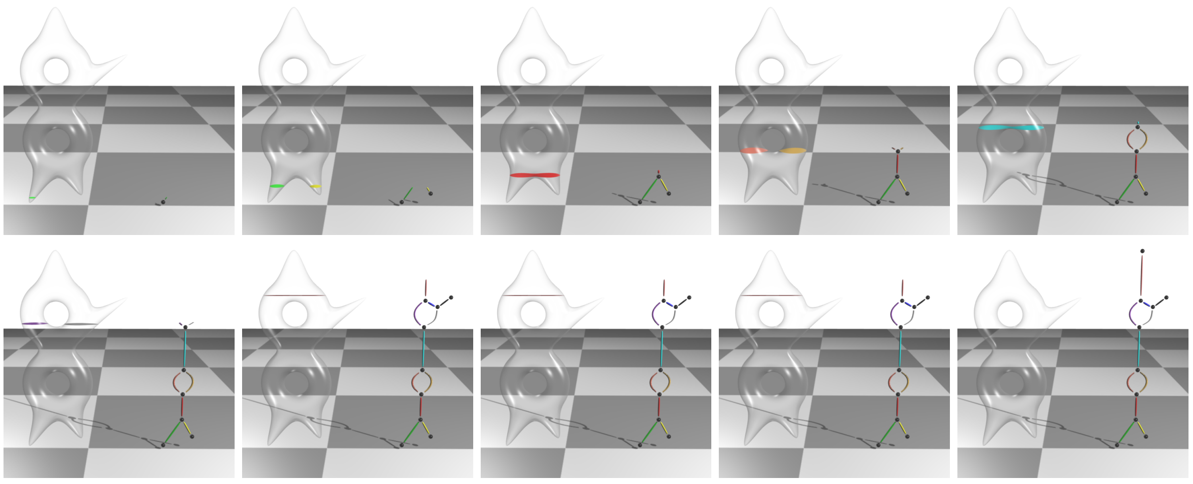


Figure 4.1: Illustration of the Sweep algorithm computing the Reeb graph of the height function defined on a solid 2-torus. Each image denotes a stage during the sweep immediately after a critical point is processed.

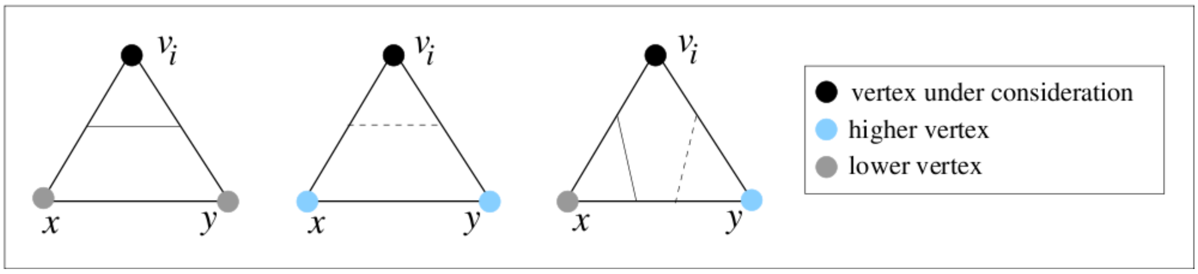


Figure 4.2: Edges in the level set before (solid line) and after (dashed line) processing a vertex v_i.

vertex. The Reeb graph is constructed incrementally based on the number of components of the level set. Figure 4.1 illustrates the algorithm for the solid 2-torus input shown in Figure 2.3. The different stages in this figure shows the various isosurface components and the partial Reeb graph that is constructed immediately after the sweep processes a critical point of the input. Procedure SWEEP outlines this algorithm.

End points of a single edge in the level set lie within two adjacent edges of a triangle in the input mesh. Figure 4.2 shows edges in the level set before and after processing a vertex event. The level set is updated locally depending on the relative function values at adjacent vertices of the triangle.

Procedure SWEEP

```
Input: Triangulated mesh $K$, PL function $f$.
Output: Reeb graph $R$
 1: Initialize Reeb graph $R = \emptyset$
 2: Initialize level set $L = \emptyset$
 3: Sort vertices of $K$ in increasing order of function value.
    Let $v_1, v_2, \ldots, v_k$ be the sorted list of vertices.
 4: for $i = 1$ to $k$ do
 5:     for each triangle $(v_i, x, y)$ incident on $v_i$ do
 6:         if $f(v_i) > f(x), f(y)$ then
 7:             remove edge $((v_i, x), (v_i, y))$ from $L$
 8:         end if
 9:         if $f(v_i) < f(x), f(y)$ then
10:             insert edge $((v_i, x), (v_i, y))$ into $L$
11:         end if
12:         if $f(x) < f(v_i) < f(y)$ then
13:             remove edge $((v_i, x), (x, y))$ from $L$
14:             insert edge $((v_i, y), (x, y))$ into $L$
15:         end if
16:         if $f(y) < f(v_i) < f(x)$ then
17:             remove edge $((v_i, y), (x, y))$ from $L$
18:             insert edge $((v_i, x), (x, y))$ into $L$
19:         end if
20:     end for
21:     Update $R$ to reflect change in the number of components of $L$.
22: end for
23: return $R$
```

4.2 Computing Reeb graph of a 3-dimensional input

Computing Reeb graphs using the algorithm outlined in the previous section requires efficient maintenance of both the level set of the input as well as the incremental Reeb graph. In this section we first describe the maintenance of level sets of a 3-dimensional input, also known as isosurfaces. Dynamic maintenance of the Reeb graph is then presented in Section 4.2.2, followed by the analysis of the algorithm in Section 4.2.3

4.2.1 Maintaining isosurfaces

A *map M* is an embedding of a graph on a 2-manifold such that the two-dimensional cells of the embedding are disks. The *dual map* M^* onto the same 2-manifold is constructed by creating a dual vertex t^* within each face t of the primal map M, and creating a dual edge e^* for each primal edge e. If e lies on the boundary of two faces t_1 and t_2, then e^* connects t_1^* and t_2^* by a path that crosses e exactly once and crosses no other primal or dual edge. The triangle mesh that represents an isosurface of f is a map whose two-dimensional cells are triangles. Planar graphs can be embedded on the sphere, which is a 2-manifold whose genus equals zero. Non-planar graphs cannot be embedded on the sphere, but they can be embedded on a higher genus 2-manifold.

Let T denote a spanning tree of M. Let C^* be a spanning tree of the dual map M^*, then we call $C = \{e | e^* \in C^*\}$ a *spanning cotree* of M. Given a map M with distinct edge weights, the minimum weight spanning tree and maximum weight spanning cotree of M are disjoint [30]. Here, the weight of a dual edge is the same as that of the corresponding primal edge.

Given a planar graph M, a *tree-cotree partition* of M is a triple (T, C, X), where T is the minimum spanning tree of M, C is the maximum spanning cotree of M, and X is the set of edges in M that are neither in the tree T nor in the cotree C. In the case of isosurfaces, since the edges of the mesh are unweighted, it is possible to use any edge disjoint spanning tree and spanning cotree and maintain the updated tree-cotree partition during the sweep process. Figure 4.3 shows a tree-cotree partition for a sphere and a torus. The cardinality of X, $|X|$, is equal to twice the *genus* of the 2-manifold. This follows from the fact that the Euler characteristic, $\chi = 2 - 2g$, of the 2-manifold can be expressed as the alternating sum of cells of M. Let $\#v, \#e$, and $\#t$ denote the number of vertices, edges, and faces of M, then

$$\begin{aligned}
\chi = 2 - 2g &= \#v - \#e + \#t \\
&= \#v_T - (\#e_T + \#e_C + |X|) + \#v_C \\
&= \#v_T - (\#v_T - 1 + \#v_C - 1 + |X|) + \#v_C \\
&= 2 - |X| \\
\Rightarrow |X| = 2g. &
\end{aligned}$$

The algorithm stores each isosurface component as a tree-cotree partition, see Figure 4.4.

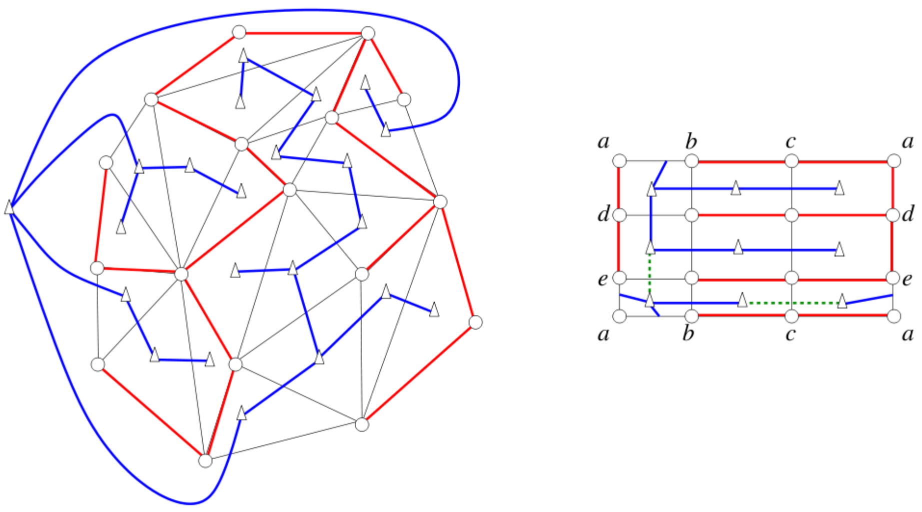


Figure 4.3: A tree-cotree partition of a graph embedding on a sphere (left) and a torus (right). Tree edges are red, cotree edges are blue, and edges from X are dotted and green in color. The 2-manifolds are “cut open” to make it easier to embed the graph. The surface is obtained by gluing along the boundary. In the case of the sphere, the tree and co-tree partition the edges of the graph. So, the set X is empty for the sphere whereas it contains two edges for the torus.

The set X is stored using a simple list data structure. To store the tree T and cotree C individually, we use a dynamic tree data structure [62] as modified in [31], known as the *edge-ordered tree*. The edge-ordered tree imposes a total order on edges incident on a tree/cotree node v, referred to as the *edge list* of v. Each node v is represented by a collection of *subnodes*, called a *node path*. A subnode v_e in the node path of v represents an edge e in the edge list of v. Subnode v_e is connected to the subnode of the predecessor and successor of e in the edge list of v. Subnode v_e is connected to subnode u_e if the edge e connects u and v. The edge-ordered tree supports *InsertEdge* and *DeleteEdge* operations, both requiring $O(\log n_v)$ amortized time per operation, where n_v is the number of nodes.

The ordering of edges around an isosurface vertex in this embedding is the same as the ordering of triangles around the corresponding edge in the input mesh. Given an edge to be inserted, its location with respect to the existing isosurface edges is determined by the corresponding triangle’s position in the input mesh. The ordered ring of mesh triangles around a mesh edge is obtained directly from the triangle-edge data structure. For efficient determination of the isosurface edge location, the algorithm stores the ordered set of mesh triangles around each isosurface vertex in a balanced search tree [18]. The *InsertEdge* and *DeleteEdge*

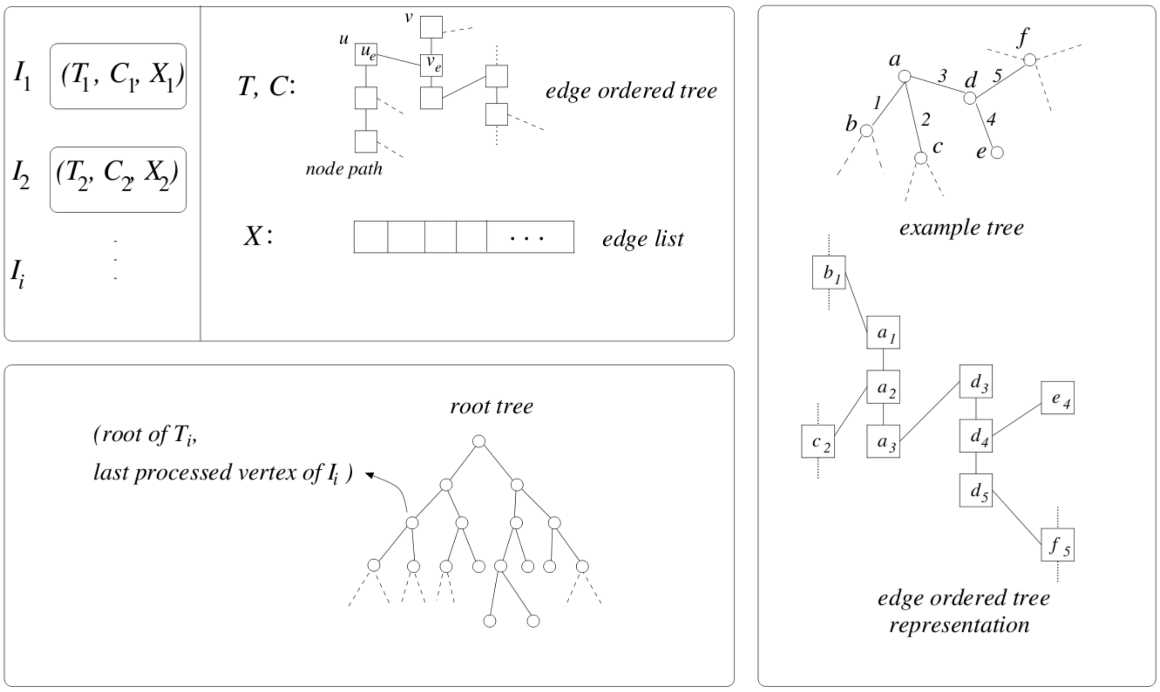


Figure 4.4: Data structures used by the algorithm to maintain isosurfaces.

operations of the edge-ordered tree are invoked by our algorithm while maintaining the isosurface during the sweep. The insert and delete operations on the tree-cotree data structure, which is required to maintain the isosurface, is performed as follows.

Insert an edge into the isosurface: To insert an edge *e*, first check if the endpoints of *e* are in the same isosurface component, and process them as follows:

- End points of *e* belong to different components: Connect the spanning trees of the two isosurface components using this edge, resulting in a spanning tree for the merged component. This also results in merging the two spanning cotrees since the insertion of this edge merges two faces. So, the corresponding nodes in the cotree are merged. This operation is illustrated in Figure 4.5(g), where an isolated node is connected with the spanning tree of an existing isosurface component.

- End points of *e* belong to the same component: Try inserting the edge into the cotree C. This is possible only if the pair of edges preceding *e* in the edge lists of the endpoint vertices of *e* share a common face. Else, the inserted edge will intersect with an edge

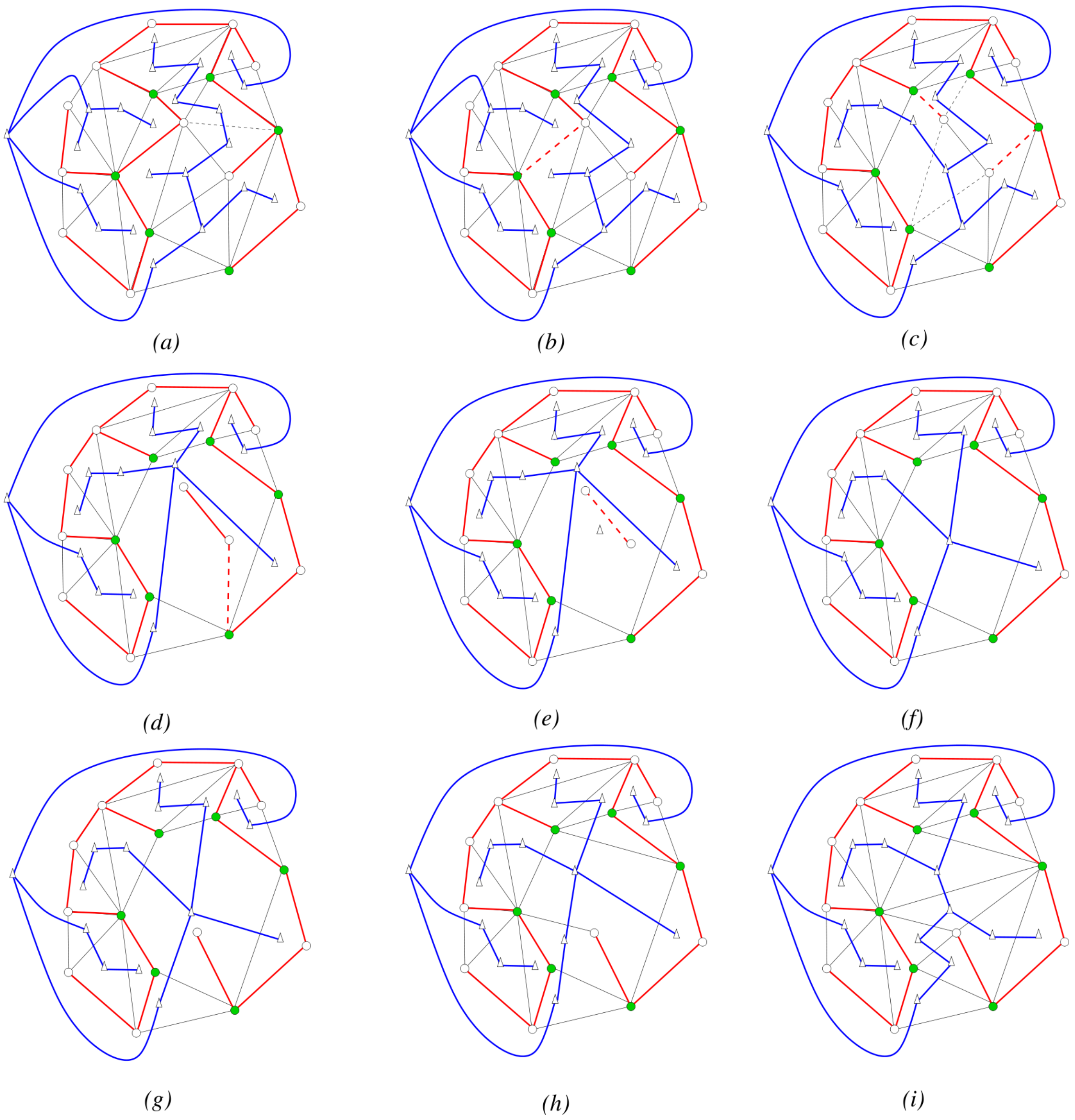


Figure 4.5: Illustration of the update operations on a tree-cotree partition when a regular vertex is processed. The 1-skeleton of the isosurface does not change outside the ring of green nodes. The dashed edges are scheduled to be removed in the next step. **(a)** The initial tree-cotree partition. **(b)** After deleting a non-tree edge. **(c)** After deleting a tree edge. A cycle is created in the cotree C when the dual nodes are merged. So, an edge from the path connecting the two dual nodes is transferred into the tree T. The dashed edges will be removed one after another, where each deletion will be of the type described in (a) or (b). **(d)** A tree edge will be removed next, resulting in a split in the component. **(e)** The dashed edge along with its dual node forms the tree-cotree partition for the newly created component. **(f)** Deleting the lone edge destroys the isosurface component. **(g)** Addition of an edge to the isosurface, thereby merging two components. The new node is considered as an individual component before addition of the edge. **(h)** Addition of a non-tree edge to the isosurface, which results in the modification of the cotree. **(i)** The tree-cotree partition after the regular vertex is processed.

in the tree T or the cotree C. This insertion is illustrated in Figures 4.5(h) and 4.5(i). Finally, if e cannot be added to the cotree C, then it is added to X.

Delete an edge from the isosurface: To delete an edge e from the isosurface, it is deleted from either the tree T, the cotree C, or the set X, as necessary. If the edge lies in the tree T, then the following two situations can occur:

- Deleting an edge merges two distinct faces of the isosurface into one: This causes a cycle in the cotree C. To handle this, remove any dual edge e^* in the cotree C, from the path connecting the dual nodes corresponding to the two faces, and add the primal edge e to the tree T. Figure 4.5(c) shows the result of this operation after the dashed tree edge in Figure 4.5(b) is removed.
- The edge is incident on a single face: In this case, removing the edge splits the corresponding isosurface component into two. This also results in the split of the dual node in the cotree corresponding to the split component. An example of such a removal is seen in Figure 4.5(d)-4.5(e).

If the edge lies in the cotree C, then removing this edge is equivalent to contracting the corresponding dual edge (Figure 4.5(a)-4.5(b)). If the edge was deleted from either the tree T or the cotree C, then the genus of the surface may have decreased and hence an edge from X can be inserted into the tree T or the cotree C without introducing a cycle. The algorithm exhaustively searches the set X to locate such an edge and moves it to the tree T or cotree C as appropriate.

4.2.2 Dynamic maintenance of the Reeb graph

Each connected component of the isosurface is represented by the root of its tree T. Two nodes lie within the same component if their roots are equal. For fast access to the individual components, the algorithm stores all roots in a balanced search tree, called the *root tree*. When processing a vertex v_i from the input mesh, the algorithm performs a set of edge insertions and/or deletions to the tree-cotree data structure. If a new component is created during this operation, then v_i is a minimum. If an existing component is destroyed, then v_i is a maximum.

If either two components merge into one or a single component splits into two, then v_i is a saddle. The criticality of a node v_i and the isosurface components that are modified is identified by comparing the connectivity of the end points of inserted / deleted edges before and after v_i is processed. A new node is added to the root tree if v_i is a minimum or a saddle that splits a component. If v_i is a maximum or a saddle that merges two components, an isosurface component is destroyed and corresponding node is deleted from the root tree. Each node in the root tree is also associated with the last processed vertex, v_{last}, that caused a modification of the corresponding isosurface component.

The Reeb graph is constructed incrementally by inserting a node after processing v_i. Assuming that edges in the Reeb graph are directed from a node with lower function value to a node with higher function value, each node can have at most two predecessors. So, the Reeb graph can be stored using an adjacency list representation where each node has at most two adjacent nodes, namely the predecessors. Similarly, each node in the Reeb graph can have at most two successors. A Reeb graph node whose successors have been inserted is called a *stationary node*, else it is called a *growing node*. A node when inserted into the Reeb graph attaches to a growing node after which it becomes a growing node, unless it is a local maximum. The predecessor growing node becomes stationary if all of its successors have been inserted into the graph.

To insert a node into the Reeb graph, its predecessor is first identified from the list of growing nodes as the one representing the updated isosurface component. This is accomplished by querying the root tree for the updated component and obtaining the associated vertex. We then associate v_i with this component in the root tree. If v_i is a saddle that merges two components, then the corresponding node will have two predecessors, each of which can be identified by looking up the two modified components in the root tree. The vertex v_i is then associated with the merged component. If the vertex v_i is a component splitting saddle, it is associated with both components that are created. The graph obtained when all vertices are processed corresponds to the augmented Reeb graph of the input. Each node in the augmented Reeb graph corresponds to a vertex, regular or critical, from the input mesh. All regular nodes and genus-modifying saddle nodes are identified as degree-2 nodes and removed by merging their incident edges to obtain the Reeb graph.

4.2.3 Analysis

Let n denote the number of triangles in the input mesh. The number of vertices and edges in the input is less than $3n$. Let g denote the maximum genus over all isosurfaces of the function. The number of saddles is a loose upper bound for g, since the genus of the isosurface can change only at a saddle. The maximum genus is typically a much smaller number.

The initial sorting of the mesh vertices takes $O(n \log n)$ time. To process each vertex, the algorithm performs a set of *InsertEdge* and *DeleteEdge* operations on the edge-ordered trees that store the tree-cotree partition. Each *InsertEdge* and *DeleteEdge* operation takes $O(\log n)$ time. Whenever an edge is deleted from the tree T or cotree C, a replacement edge is identified from X. Since $|X| \leq 2g$, finding the replacement edge and updating the tree-cotree data structure takes $O(g \log n)$ time. In order to bound the number of insertions and deletions, consider the number of insertions into and deletions from each triangle. As shown in Figure 4.2, there are exactly two insertions and two deletions per triangle to give a total of $2n$ insert/deletes. Nodes in the data structure correspond to edges in the input. So, maintaining the tree-cotree partition requires $O(ng \log n)$ time using the edge-ordered tree.

Finding the replacement edge from X is a costly operation. Selected edges from the tree T and cotree C can be contracted to derive a new tree T' and cotree C', each of which has $|X|$ edges, such that a replacement edge for (T', C') is also the replacement edge for (T, C). When (T, C) changes, (T', C') can be updated in $O(\log n)$ time [30]. The general dynamic graph connectivity algorithm of Holm et. al. [44] applied on smaller graphs $T' \bigcup X$ and $C' \bigcup X$ can find the replacement edges in $O(\log^2 g)$ time. Using the randomized dynamic graph connectivity algorithm by Thorup [69] improves the time complexity of this step to $O(\log g (\log \log g)^3)$ expected. The dynamic connectivity algorithms [44, 69] are outlined in Section 4.3.

To identify the various isosurface components and to maintain the Reeb graph, the algorithm performs a constant number of insert, delete, or update operations on the root tree when a mesh vertex is processed. Inserting a node into the Reeb graph requires at most two $O(\log n)$ time queries on the root tree to identify the predecessor(s), and a constant time update of the adjacency list representation. Thus, the Reeb graph can be maintained in $O(n \log n)$ time. Putting the various steps together, we obtain the following theorem.

THEOREM 4.1. *The sweep algorithm constructs the Reeb graph of a PL function defined on a 3-manifold in* $O(n \log n + n \log g (\log \log g)^3)$ *expected time.*

4.3 Computing Reeb graph of a higher-dimensional input

The tree-cotree partition described previously works only when the level sets are 2-manifolds because the 1-skeleton of the level set corresponds to a map M. This is not true in higher dimensions. So, we require a different data structure to store connected components of a level set. The data structure used to maintain the Reeb graph remains unchanged.

4.3.1 Maintaining level sets

Working with a graph representation of the 1-skeleton of the level set, our algorithm uses the fully-dynamic connectivity algorithm described in [44] to track the evolution of level sets and answer connectivity queries. The dynamic connectivity algorithm stores the spanning forest F of a graph G for fast insertion of edges and quick response to connectivity queries. When an edge in F is deleted, it causes a split in a tree in F, and if the corresponding component in G is not split, then a replacement edge must be inserted into F. In order to find this replacement edge efficiently, each edge $e = (v, w)$ is associated with a level, $l(e) \leq l_{max} = \lfloor \log n_v \rfloor$, for a graph with n_v nodes. For each i, F_i, a sub-forest of F induced by the edges of level at least i, is maintained. The replacement edge for a tree edge is now searched systematically in the set of sub-forests. The above replacement is carried out by a recursive *Replace*$((v, w)$,$i)$ operation, which, assuming that there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leq i$, if any, such that v and w belong to the same component after adding the replacement edge.

4.3.2 Analysis

The fully-dynamic graph connectivity algorithm supports maintaining the spanning forest in $O(\log^2 n_v)$ amortized time per update and answering connectivity queries in $O(\log n_v / \log\log n_v)$ time for a graph with n_v nodes. Since the rest of the implementation of the Reeb graph algorithm remains unchanged and the number of vertices in the level set is $O(n)$, we have an $O(n\log^2 n)$ time algorithm for constructing the Reeb graph. Here, n is the number of triangles in the input.

In [69], the connectivity algorithm was modified to store an alternative rooted forest S, called the *structural forest* for a given graph G, instead of the spanning forest F. The leaves of S correspond to vertices in G, and all of them have a depth equal to l_{max}. A level of a node in S is its depth. For each i, G_i denotes the subgraph induced by edges of level at least i. Nodes in S at a level i represents the components in G_i. This alternative representation was shown to support update operations in $O(\log n(\log\log n)^3)$ expected amortized time and connectivity queries in $O(\log n/\log\log\log n)$ time [69]. Using this algorithm to store level sets will improve the time complexity of the Reeb graph construction as stated in the following theorem.

THEOREM 4.2. *The sweep algorithm constructs the Reeb graph of a PL function defined on an arbitrary simplicial complex in $O(n\log n(\log\log n)^3)$ expected time.*

Chapter 5

Cylinder Map Algorithm

Most of the algorithms proposed to compute Reeb graphs focus on efficiently keeping track of the level set components. While these techniques produced fast algorithms for a 2-dimensional input, such optimizations either did not extend to a generic input, or was too complex to be of any practical use. This chapter presents 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, is easy to implement, and works well in practice. In Section 5.1, we first describe an alternate definition of the Reeb graph that follows from a simple observation. The cylinder map algorithm described in Section 5.2 follows directly from this definition.

5.1 Cylinder Map

The description of the Reeb graph presented in Chapter 2 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 5.1.

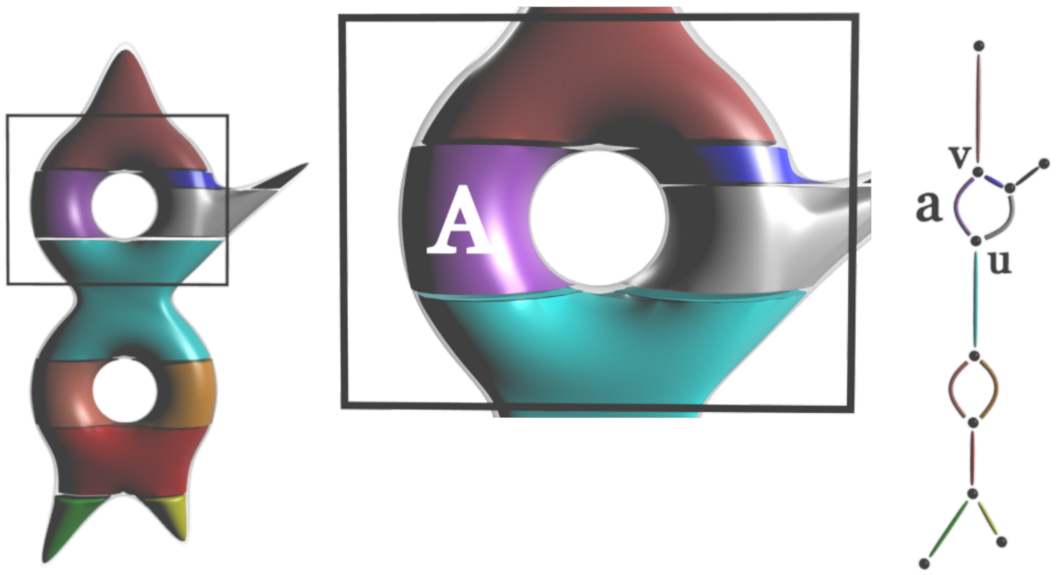


Figure 5.1: The 2-torus is partitioned by grouping together regular level set components that are topologically equivalent to each other. Each cylinder in this partition corresponds to the region in the input corresponding to an arc in the Reeb graph. For example, arc *a* in the Reeb graph maps to cylinder *A* in the input.

Given a critical point c_i, call the level set $f^{-1}(f(c_i))$ as a *critical level set*. 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 boundary component. Different colors in the figure depict the cylinders corresponding to individual arcs of the Reeb graph. In Section 5.1.1 we discuss how various cylinders in the input can be represented. We then introduce the *LS*-graph in Section 5.1.2, which is used by our algorithm to efficiently track individual cylinders.

5.1.1 Cylinder representation

As mentioned in the previous chapter, the 1-skeleton representation (vertices and edges) of the level sets is sufficient to track its connectivity. The edges 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.

5.1.2 *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 $V = \{t_1, t_2, \ldots, t_n\}$ correspond 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 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 5.2. 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 5.2(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 5.2(f) shows a 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.

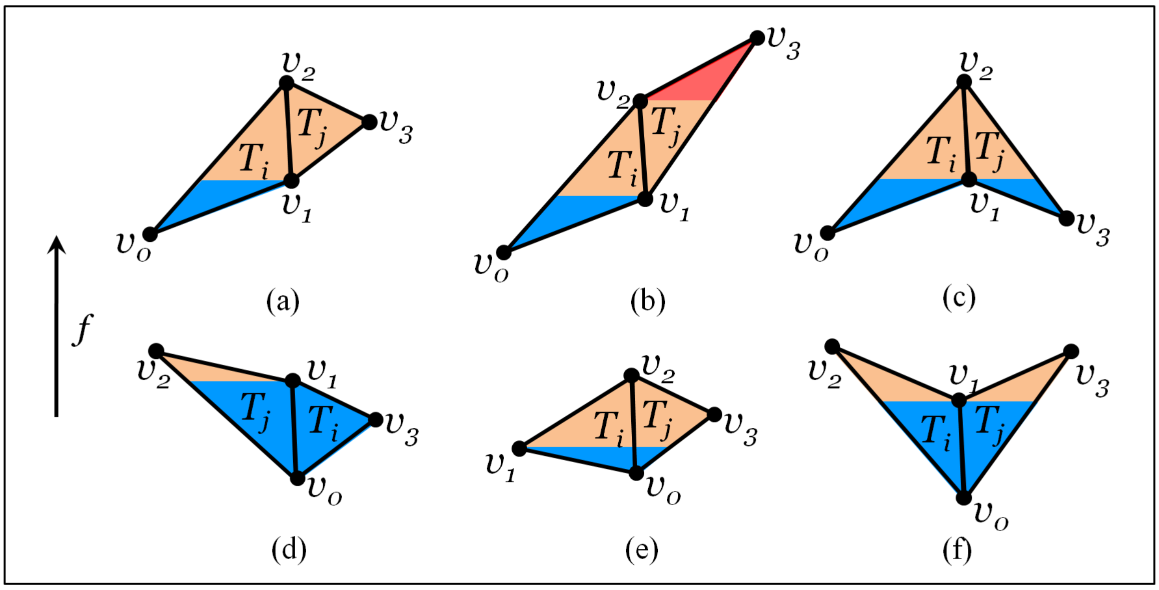


Figure 5.2: Adjacent triangles in the input can have one of six possible configurations. The *LS*-graph contains an edge from t_i to t_j in all cases except the forbidden configuration in (f). Edges in the graph are directed towards the node with higher cost. The boundary between the blue and orange 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.

5.2 The Cylinder Map Algorithm

We now describe an algorithm that computes the Reeb graph of a PL function f defined on a triangular mesh. In order to simplify the description, we will illustrate this algorithm for a 3-manifold input. The algorithm directly extends to d-manifolds ($d \geq 2$) and non-manifolds, and this generality is discussed later in Section 5.2.3. The cylinder map algorithm computes the Reeb graph in two stages:

1. Locate critical points in the domain and sort them based on function value.

2. Identify pairs of critical points that define cylinders and insert the corresponding arcs in the Reeb graph.

This algorithm[1] is summarized in Procedure CYLINDERMAP.

[1]The video at *http://vgl.serc.iisc.ernet.in/pub/paper.php?pid=009* [24] illustrates the working of the algorithm.

Procedure CYLINDERMAP

```
Input: Triangulated mesh $K$, PL function $f$.
Output: Reeb graph $R$
 1: Initialize the set of critical points $C = \emptyset$
 2: Initialize the set of level sets $L = \emptyset$
 3: Initialize Reeb graph $R = \emptyset$
 4: for each vertex $v \in K$ do
 5:     if $v$ is critical then
 6:         Add $v$ to $C$
 7:     end if
 8: end for
 9: Sort $C$ in increasing order of function value
    Let $C = \{c_1, c_2, \ldots, c_t\}$ be the sorted set of critical points
10: for each critical point $c_i \in C$ do
11:     Let $L_i = \{$components of $f^{-1}(c_i - \varepsilon) \mid$ the component passes through the lower star of $c_i\}$
12:     Add $L_i$ to $L$
13: end for
14: for each critical point $c_i \in C$ do
15:     if $c_i$ is not a maximum then
16:         for each component in the upper link of $c_i$ do
17:             Travese the $LS$-graph starting from that component until a level set $L_p$ is reached
18:             Add arc $(c_i, c_p)$ to $R$
19:         end for
20:     end if
21: end for
22: return $R$
```

5.2.1 Identifying critical points

The algorithm uses the characterization described in Section 2.5.2 to identify the set of critical points of the PL function f. It counts the number of components in the upper and lower links of each vertex by performing a breadth first search in the graph formed by vertices and edges in the upper and lower links respectively. A vertex is classified as regular if the number of components in its upper and lower links is one. Else, it is classified as critical.

5.2.2 Connecting the critical points

The arcs in the Reeb graph are computed 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 $x < y$. Let L_i denote the set of triangles containing the components of the level set $f^{-1}(f_i - \varepsilon)$

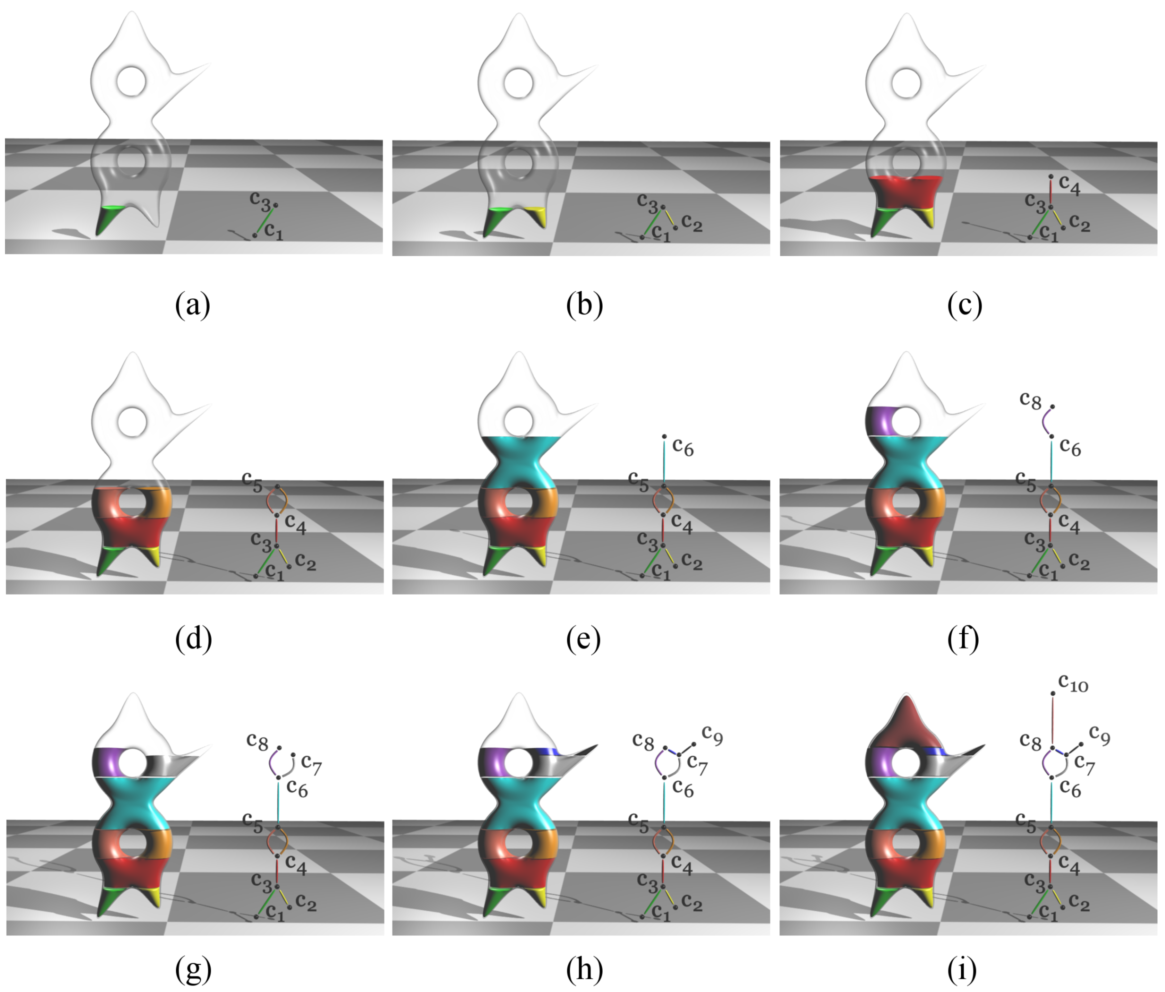


Figure 5.3: Illustration of the two-step algorithm computing the Reeb graph of the height function defined on a solid 2-torus. This model has ten critical points, including two minima, two maxima, and six saddle points. The critical points are first sorted in increasing order of function value. Let $c_1, c_2, ..., c_{10}$ 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.

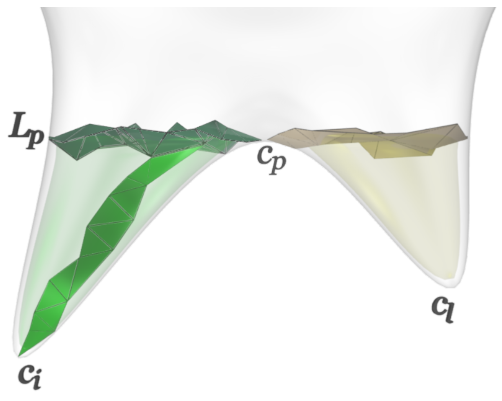


Figure 5.4: Connecting critical points. The set of green triangles shows the path traced in the *LS*-graph by the search procedure initiated at c_i. The search terminates when it reaches a triangle in L_p. Similarly the search initiated at c_l also terminates at L_p.

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 5.3 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 5.3(d) illustrates the case when the upper star of c_4 has two components. The cylinders bounded below by a level set component of c_i is traced during the i^{th} iteration of the algorithm. The algorithm initiates 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, the algorithm traverses to a higher cost node in G_{LS} and terminates the search when it reaches a node t' whose cost is greater than or equal to f_{i+1}. An arc is inserted into the Reeb graph between nodes corresponding to c_i and c_{i+1} if and only if the triangle T' dual to t' belongs to L_{i+1}.

If T' does not belong to L_{i+1}, the algorithm continues this traversal until it reaches a node whose cost is greater than or equal to f_{i+2}. It then tests if the dual triangle belongs to L_{i+2}. This search is repeated until the traversal finds the set L_p that bounds the cylinder. This operation is shown in Figure 5.3(f). Figure 5.4 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.

The *LS*-graph is implicitly stored in the triangle-edge data structure since each triangle-edge pair stores a reference to neighboring triangle-edge pairs. Traversal from a dual triangle is performed 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.

5.2.3 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 5.1.1, 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.

In the case of non-manifolds, whenever a regular vertex has two link components, an edge is added in the *LS*-graph between the 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 a point. Figure 2.6 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. After this preprocessing step, the algorithm works without any modification on the 2-skeleton representation of both d-manifolds and non-manifolds.

5.2.4 Analysis

We first prove that the Cylinder Map algorithm indeed computes the Reeb graph of the input PL function f and next 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 the algorithm tracks 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 the function value increases 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 it reaches 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.

Consider the *LS*-graph traversal done by the algorithm to identify the arcs of the Reeb graph. For each component in the upper link of a critical point, this traversal traces a path in the *LS*-graph. Let the last node of such a path be t'. Then either triangle T' belongs to L_i for some i, or the node t' was already labeled. Each node is labeled exactly once during the search procedure. So the total number of nodes that are labeled is bounded by n. The number of labeled nodes that are visited is bounded by the number of paths traced by the algorithm, which is also bounded by n. 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 the following theorem.

THEOREM 5.1. *The cylinder map algorithm constructs the Reeb graph of a PL function defined on an arbitrary simplicial complex in* $O(n + l + t \log t)$ *time.*

5.3 Implementation

Our implementation of the cylinder map algorithm requires data to be in memory. It may therefore not be possible to store the sets L_i of all critical points of a relatively large input. We develop two optimization strategies that allow us to overcome this shortcoming.

Filtering degree-2 saddles

Critical points in the input are identified based on the local neighborhood information obtained through the upper and lower links of each vertex. Our algorithm therefore also identifies degree-2 saddles, even though the connectivity of level sets do not change at these vertices. The same is true even for certain regular vertices that are present on the boundary of the input. We discard these vertices and compute the sets L_i only for the remaining critical points and hence reduce the memory footprint considerably. This is accomplished by marching 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 a triangle that is part of a different upper or lower star component, we recognize that the level set consists of a single component and stop processing c_i. This filtering step, in the worst case, requires the additional computation of the level set $f^{-1}(f_i + \varepsilon)$ for all critical points c_i whose upper link has two components. We observe in our experiments that this overhead is small in practice. In addition to reducing the memory used by the algorithm, removal of these false positives also significantly reduces the time taken to compute the sets L_i.

Process critical points in batches

As a second memory optimization, instead of processing all critical vertices in a single step, critical points are processed in batches of a predetermined granularity. Let b denote this granularity. After sorting the critical points in increasing order of function value, the sets L_i are computed for the first b critical points. Next, arcs originating from this set are identified. The second end point of an arc may not lie within the current set of b critical points. In this case, the partially traced paths are stored in a queue when the *LS*-graph traversal passes a function value greather than that of the b^{th} critical point. Paths in the queue are traced first when the next set of b critical points are processed. By splitting the second stage of the algorithm into batches, it is sufficient to store the sets L_i for b critical points at any instant of time. The memory required can thus be limited by varying the value of b.

Chapter 6

Recon Algorithm

The cylinder map algorithm presented in the previous chapter is both efficient and practical. However, its performance does not match that of algorithms customized for specific subclasses of the input as shown in the following chapter. The cylinder map approach essentially splits the input at every critical point to obtain a set of cylinders, computes the single arc Reeb graph (or contour tree) of each cylinder, and stitches together these arcs to obtain the Reeb graph of the input. The bottleneck of this approach, from the perspective of both the running time and memory used, is in its split process, where the algorithm computes the set of level sets L_i. In this chapter, we propose the Recon algorithm that addresses this issue by performing an optimal number of splits to the input domain, and constructs the Reeb graph by merging together the Reeb graphs of the split subdomains.

The algorithm, in a reconnaissance step, identifies the set of loops in the input where the input domain should be split. The Reeb graphs of the resulting split subdomains are "loop-free", and can therefore be constructed using the efficient contour tree algorithm. The set of contour trees are then merged to obtain the Reeb graph. We first describe the characterization used for identifying the set of loops in Section 6.2, followed by a detailed description of the algorithm in Section 6.3. Some of the optimizations performed in our implementation is then discussed in Section 6.4. We will illustrate the algorithm using the solid vase model shown in Figure 2.3. We note that the algorithm works without any modifications for d-manifolds, $d \geq 2$, and non-manifolds. We also assume that all saddles are simple in the following description.

6.1 Contour tree algorithm

The key idea in our proposed approach to compute Reeb graphs is to optimize the use of the efficient contour tree algorithm. For completeness, we now briefly describe the contour tree algorithm. For a more detailed description, we refer the reader to the paper by Carr et al. [12].

The contour tree algorithm makes two passes over the data to compute the *join tree* and *split tree* of the input. The join tree tracks the connectivity of super-level sets of the input scalar function and identifies all the maxima and join saddles of the contour tree. It is computed by first sorting the vertices of the input in decreasing order of function value. Next, for each vertex v in this sorted list, the algorithm performs the following operations:

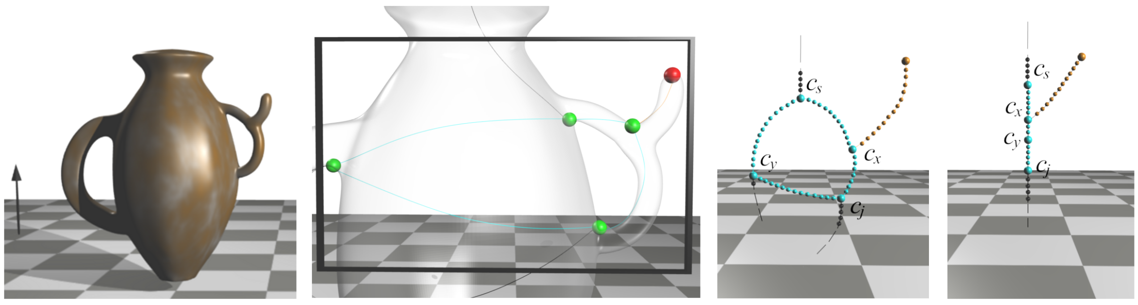
- If v is a maximum (its upper link is empty), create a new component containing v and set v as its *head*.
- If the upper link is not empty, find the components that contain the vertices in the upper link of v. Add an arc between v and the head of each of the components. Merge these components and set v as the head of the merged component. If the number of components is greater than one then v is a join saddle.

Similarly, the split tree tracks the connectivity of the sub-level sets and identifies the set of minima and split saddles. It is computed by traversing the vertices in increasing order of function values. The join and split trees are merged to obtain the contour tree.

6.2 Loop Identification

A *chord* in an undirected graph is an edge that connects two nodes of a cycle in the graph, but is not part of the cycle. A cycle is an *induced cycle* if the subgraph induced by the nodes of the cycle does not contain a chord [19]. *Loops* in a Reeb graph correspond to the set of all induced cycles in the graph. Figure 6.1(a) shows the height function defined on a solid vase model having two loops.

Consider any loop L in the augmented Reeb graph of the given input. Define the set V_L as the set of all vertices of the input that belong to loop L. Figure 6.1(c) shows the Reeb



(a) (b) (c) (d)

Figure 6.1: Join saddles that end a loop in the Reeb graph form degree-2 nodes in the join tree. **(a)** Height function defined on a solid vase model having two loops. **(b)** A loop in the vase model is highlighted. **(c)** The Reeb graph within the highlighted region augmented with regular vertices. **(d)** The augmented join tree corresponding to the loop. Note that c_j, the join saddle that ends the loop, forms a degree two node in the join tree.

graph corresponding to a loop in the vase model shown in Figure 6.1(b), augmented with degree-2 nodes. The set V_L for this loop is highlighted in cyan. Define $c_j = \inf(V_L)$ and $c_s = \sup(V_L)$ where the vertices are ordered based on the function values. If we sweep the input with decreasing function value, then the split saddle c_s *begins* the loop L, while the join saddle c_j *ends* it. We are interested in finding all such *loop saddles* – a set of saddles that begin or end a loop in the Reeb graph. The function values at these saddles are in turn used to obtain a set of loop-free interval volumes. This is accomplished by splitting the input at these saddles.

The set of all join or split saddles is a superset of the set of loop saddles. Using it as a conservative estimate while splitting the input domain may lead to an unnecessarily large number of interval volumes. In order to reduce this overhead, we utilize the contour tree algorithm to find a better estimate of the set of loop saddles. The following lemma provides us with the necessary condition to compute this set.

LEMMA 6.1. *Let G_R be the Reeb graph of a scalar function f. Consider the join tree T_J of f. Any join saddle that ends a loop in G_R appears as a degree-2 node in T_J.*

Proof. Consider a split saddle c_s that begins a loop L in G_R. Let this loop end at the join saddle c_j. Let V_L be the set of vertices that belong to the loop L. Consider the algorithm that computes the join tree T_J of f and a vertex $v_j \in V_L$, with $f(v_j) < f(c_s)$ and $v_j \neq c_j$, that is processed by the algorithm. Let $v_k \in Lk^+(v_j)$ be a vertex in the upper link of v_j such that

$v_k \in V_L$. Due to the way the join tree algorithm works, v_k will be in the same component as c_s. This is true for all vertices in V_L.

Now, consider the step when c_j is processed by the join tree algorithm. Its upper link $Lk^+(c_j)$ will consist of two components. Vertices in both components belong to L, and will therefore belong to the same component as c_s. Hence, the join tree algorithm will add only a single arc between c_j and the head of this component to T_J. For every node in the join tree T_J, the number of neighbors with function value less than the node is always one. Hence, the degree of c_j in the join tree T_J is two. $\square$

Figure 6.1(c) shows the join tree corresponding to the loop in Figure 6.1(a) where c_j is a degree-3 node in the Reeb graph and a degree-2 node in the join tree. A similar proof can be used to show that any split saddle that begins a loop appears as a degree-2 node in the split tree. The above lemma directly leads to an algorithm for identifying loop saddles.

6.3 The Recon Algorithm

This section describes the Recon algorithm. As mentioned earlier, for ease of exposition, we illustrate the algorithm on the height function defined on a solid vase model. The algorithm, summarized in Procedure RECON, computes the Reeb graph in four stages:

1. Identify the loop saddles of the input.
2. Split the input at a function value infinitesimally above that of the loop saddles to obtain a set of interval volumes.
3. Compute the contour tree for each interval volume.
4. Construct the Reeb graph by computing the union of these contour trees.

Figure 6.2 illustrates the different steps of the algorithm. We explain the first two steps of the Reeb graph computation algorithm in detail, while the third and fourth steps of the algorithm are straightforward. We analyze the time and space complexity of the algorithm in Section 6.3.4. Extension of the algorithm to handle saddles with degree greater than three is

Procedure RECON

```
Input: Triangulated mesh $K$, PL function $f$.
Output: Reeb graph $R$
 1: Initialize the set of join saddles $S = \emptyset$
 2: Initialize the set of loop saddles $S_L = \emptyset$
 3: Initialize Reeb graph $R = \emptyset$
 4: for each vertex $v \in K$ do
 5:     if $v$ is critical and the upper link of $v$ has 2 components then
 6:         Add $v$ to $S$
 7:     end if
 8: end for
 9: Compute the Join Tree $T_J$ of $K$
10: for each join saddle point $c_j \in S$ do
11:     if $degree(c_j) = 2$ in $T_J$ then
12:         Add $c_j$ to $S_L$
13:     end if
14: end for
15: for each loop saddle point $c_j \in S_L$ do
16:     Split the input at a function value $f(c_j) + \varepsilon$
17:     Let $(c_{max}, c_{min})$ denote the maximum-minimum pair created when a level set component splits
        the input
18: end for
19: for each component in the split domain do
20:     Compute the contour tree $T_C$ of that component
21:     Add arcs of $T_C$ to $R$
22: end for
23: for each arc pair of the type $(c_i, c_{max}), (c_{min}, c_j)$ do
24:     Remove the pair of arcs from $R$
25:     Add arc $(c_i, c_j)$ to $R$
26: end for
27: return $R$
```

described in Section 6.3.5. Finally, in Section 6.3.6 we discuss the generality of the algorithm in handling d-manifold ($d \geq 2$) and non-manifold input.

6.3.1 Identifying loop saddles

The algorithm first identifies all potential loops of the Reeb graph. Each loop is represented by the join saddle that ends the loop. The algorithm computes the set of loop saddles as follows:

i. Compute all critical points of the input. This is accomplished by the method outlined in Section 5.2.1. Figure 6.2(a) highlights the critical points of the vase input. The blue,

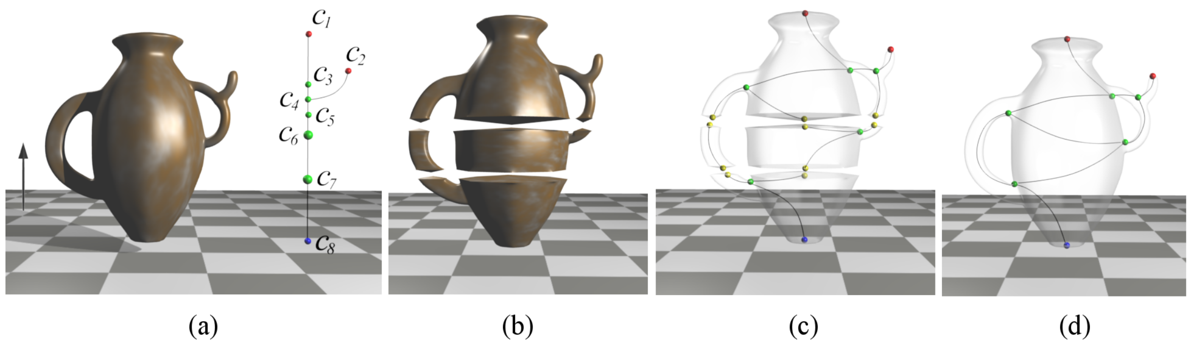


Figure 6.2: The Reeb graph computation algorithm. **(a)** The join tree for the input is computed and the loop saddles are identified. Here, c_6 and c_7 are identified as loop saddles. **(b)** The input is split at a function value infinitesimally above that of the loop saddles to obtain a set of interval volumes. **(c)** The contour tree for each interval volume is computed. **(d)** The contour trees are merged to obtain the Reeb graph of the input.

green, and red nodes denote the set of minima, saddles, and maxima respectively.

ii. Compute a superset S of the set of join saddles. This set S is equal to the set of all critical points whose upper link consists of two components. For the vase input, $S = \{c_4, c_6, c_7\}$.

iii. Compute the join tree T_J of f. The join tree of the vase input is shown on the right in Figure 6.2(a).

iv. Identify the set of loop saddles. A superset of the set of loop saddles S_L is obtained using the characterization from Lemma 6.1. It is defined as $S_L = \{c_i \in S \mid deg(c_i) \ in \ T_J = 2\}$. The join saddles c_6 and c_7 correspond to loop saddles in Figure 6.2(a), since they form degree-2 nodes in the join tree.

In case of a 2-manifold input, the set of split saddles of the input are also identified as loop saddles. This is because the upper and lower links of all saddles in such input consists of two components. Similarly, degree-2 saddles present in three and higher dimensional input are included in the set of loop saddles. However, such false positives can be identified and eliminated when splitting the input.

6.3.2 Splitting the input

For each potential loop saddle c_j in S_L, the algorithm splits the input at a function value $f(c_j) + \varepsilon$, for an appropriately small value of ε. The set of interval volumes thus obtained have the property that their Reeb graphs do not contain loops, and can therefore be computed using the contour tree algorithm of Carr et al. This operation is illustrated in Figure 6.2(b). Splitting the input requires splitting the set of triangles T that contains the level set at $f(c_j) + \varepsilon$. This level set could potentially consist of multiple components. Therefore, identifying this set T requires searching through the entire mesh. This naïve operation is not efficient especially when the number of loop saddles is large. Therefore, the algorithm incrementally maintains this set T during a sweep of the input vertices in increasing order of function value. After processing each vertex v in this sweep, the set T contains the level set at $f(v) + \varepsilon$. The set T is initially empty. Each vertex v is processed as follows:

i. Remove the triangles in the lower star of v from T.

ii. Add the triangles in the upper star of v to T.

iii. If $v \in S_L$, perform a breadth-first traversal along adjacent triangles in T to obtain the set of connected components of the level set at $f(v) + \varepsilon$.

iv. Split the input along this level set, creating a new maximum-minimum pair for each component.

In step (iii) above, if the traversal starting from one component of a loop saddle c_j's upper link reaches the other component, then it implies that the two components of the upper link belong to a single component of the level set at $f(c_j) + \varepsilon$. The saddle c_j is therefore not a join saddle. When the algorithm encounters such a situation, it classifies c_j as a false positive, and does not split the input at that vertex.

The split performed in step (iv) is realized by "cutting" each triangle in the set T and connecting the edges of the level set to two new extrema, a maximum and a minimum. The connections are established by creating a set of triangles that belong to either the lower star of the maximum or the upper star of the minimum. Figure 6.3 shows the set of created triangles

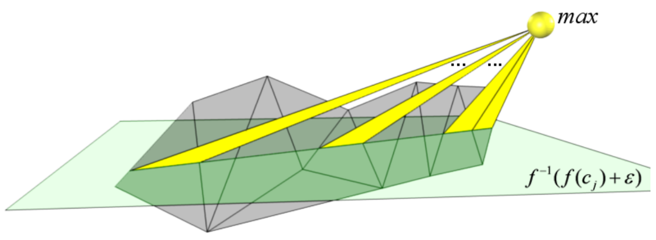


Figure 6.3: The split operation performed on triangles (gray) that contain the level set (green) at a function value infinitesimally above that of a loop saddle. The yellow triangles denote the lower star for the new maximum that is inserted. Each triangle in this star contains an edge that lies within a gray triangle. The lower star of the new maximum is represented using the gray triangles.

that are part of the lower star of a new maximum. Vertices colored yellow in Figure 6.2(c) denote all the additional extrema created for the vase input.

Each new triangle corresponds to an edge in the level set, which in turn can be uniquely identified by a triangle in the input mesh. This property is utilized by our algorithm to represent each new triangle using an existing mesh triangle. Also, the lower star of a new maximum (respectively, the upper star of a new minimum) is a connected component of a level set. This fact enables the algorithm to represent all triangles in the lower star of the maximum (respectively, upper star of the minimum) using only a single triangle that contains this level set component. When required, the other triangles in the star are obtained by marching along adjacent triangles starting from the representative. If a triangle is already split then it implies that this triangle spans a function range that encompasses more than one loop, and is therefore not split a second time.

6.3.3 Constructing the Reeb graph

The next step of the algorithm computes the contour tree of the individual interval volumes. Figure 6.2(c) shows the contour trees corresponding to the interval volumes of the vase input. In the final step, the algorithm constructs the Reeb graph of the input by merging the nodes of the contour trees corresponding to the maximum-minimum pairs created in Step 2 (Figure 6.2(d)).

6.3.4 Analysis

We now analyze the running time of and space required by our algorithm in the worst case scenario.

Time complexity

Let N be the number of vertices, n be the number of triangles and s be the number of saddles in the input. Identifying the loop saddles requires the computation of critical points which in turn requires the computation of the number of connected components of the lower and upper links of each vertex. This takes $O(n)$ time if the mesh is represented using the triangle-edge data structure. Computing the join tree requires sorting the input which takes $O(N \log N)$ time, and a set of $O(n)$ union-find operations, which takes $O(n\alpha(n))$ time. Here, α is the inverse Ackermann function.

In order to obtain the required set of interval volumes, the algorithm performs at most $|S_L|$ cuts to the input. The size of the set S_L is bounded by the number of saddles s. Note that in practice $|S_L|$ is usually much smaller than s. Each cut requires the traversal of a level set, whose size is at most n. Hence, the time required to perform these cuts is bounded by $O(sn)$. The new extrema are then inserted into the sorted list of vertices at the appropriate positions.

Computing the set of contour trees requires computing the join and split trees of each interval volume. Since the set of critical points are already identified and sorted, the join and split trees can be computed in $(n+sn)$ time using monotone paths [15]. Reconstructing the Reeb graph from the set of contour trees is done by merging the various maximum-minimum pairs, which can be done in $O(n)$ time. Combining the above steps, we obtain the following theorem.

THEOREM 6.2. *The Recon algorithm constructs the Reeb graph of a PL function defined on an arbitrary simplicial complex in $O(N \log N + sn)$ time.*

Space complexity

The triangle-edge data structure stores the triangles adjacent to each edge in the form of a triangle fan. This requires storing pointers to previous and next triangle for each edge of a triangle, a total of $6n$ pointers. The algorithm requires the star of each vertex in order to compute the critical points. For manifold input, this can be accomplished by traversing adjacent triangles using the triangle-edge data structure. But this method does not work for non-manifold input where the star of a vertex can consist of multiple components. Hence, all triangles in the star of each vertex need to be stored, a total of $3n$ triangles.

While splitting the domain into interval volumes, the algorithm stores a seed triangle for each component of the $|S_L|$ level sets. The number of such components is bounded by n. The auxiliary data structures required during the computation, such as the union-find data structure, vertex list, the edge lists representing the join and split trees, and the reconstructed Reeb graph together requires $O(n)$ space. Thus the over all space required for computing the Reeb graph is bounded by $O(n)$.

6.3.5 Handling higher degree saddles

A PL function may contain saddles that appear as Reeb graph nodes with degree greater than three. The sum of the number of components in the upper and lower link of such a saddle is greater than three. Loop saddle identification, based on Lemma 6.1, extends to these higher degree saddles. Consider a higher degree saddle which merges m level set components and ends k loops. The degree of the corresponding node in the join tree becomes $m-k+1$.

We extend our algorithm to handle higher degree saddles by modifying the loop saddle identification step as follows – given a join saddle c_j having m components in its upper link, if the degree of c_j in the join tree T_J is less than $m+1$, then c_j is added to the set S_L. The rest of the algorithm remains unchanged.

6.3.6 d-manifolds and non-manifolds

As mentioned earlier, tracking the connected components of the level set requires only the edges of the level set, which can be extracted from the triangles of the input mesh. So the algorithm works without any modifications for both d-manifolds and non-manifolds that are represented by triangles.

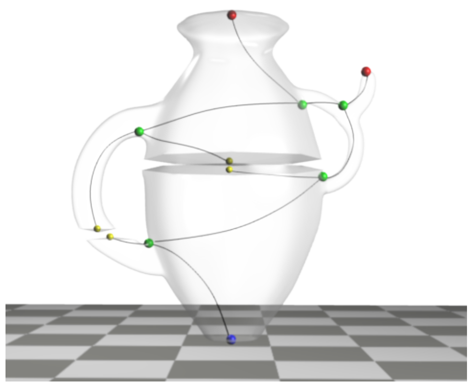


Figure 6.4: Our in-memory implementation performs splits on one of the arcs corresponding to a loop saddle. The maximum-minimum pairs shown in yellow denotes the two cuts performed by this modified procedure for the vase input instead of the five cuts shown in Figure 6.2(c).

6.4 Implementation

In this section, we describe some of the design choices made during the implementation to handle generic input and to improve the performance of our algorithm. Our implementation accepts a function sampled at vertices of a simplicial mesh as input, computes the Reeb graph, and stores it as a set of arcs. We describe the in-memory implementation of the algorithm in Section 6.4.1, followed by an extension to handle large data in Section 6.4.2.

6.4.1 In-memory implementation

We modify the Reeb graph computation algorithm to take advantage of the fact that the entire data set together with all the auxiliary data structures is available in memory. Instead of splitting the entire level set at a loop saddle, it suffices to split the level set component corresponding to one of the arcs of the loop. We compute the contour tree of the resulting connected subvolume, merge the newly inserted maximum-minimum pairs to obtain the loops and hence construct the Reeb graph. Figure 6.4 illustrates this modified procedure on the vase input, where we perform cuts only on two components of the level set instead of the five components as required by the unmodified algorithm. The use of this modified procedure not only reduces the overhead of additional splits, but also enables us to perform further optimizations to our implementation.

The in-memory implementation first sorts the vertices of the input in increasing order of function value. This sorted set is stored as a list. Finding all critical points followed by computing the join tree of the input to identify loop saddles would require two passes over the data, while computing the join tree of the split domain would require an additional pass. Our implementation combines all three operations into a single pass, thus reducing processing time. To achieve this, we modify the join tree / split tree computation algorithm as follows.

Computing the Join tree

A key difference between the join tree algorithm described by Carr et al. and our implementation is that we keep track of the components of the super-level sets using triangles instead of vertices. While processing each vertex, it is first classified as regular or critical and processed accordingly:

- **Regular:** The component containing the vertex is obtained from the triangles incident on its upper link. All triangles in its lower star are assigned to this component.
- **Maximum:** A new component is created and all triangles in the lower star are assigned to this component. The maximum is assigned as the head of this component.
- **Join Saddle:** The triangles incident on the upper link of a join saddle c_j belong to m ($m \geq 2$) level set components. The m arcs are inserted into the join tree, between c_j and the heads of the m components. The components are then merged, all triangles in c_j's lower star assigned to the resulting component, and c_j is set to be the head of this component.
- **Loop Saddle:** The vertex is essentially a join saddle c_j, where all triangles incident on at least two components of the upper link belong to a single component of the super-level set. Let the number of upper link components sharing a super-level set component be k. This results in a join saddle that closes $k-1$ loops with respect to the super-level set component. The loop is split along $(k-1)$ components of the level set passing through the upper star of c_j, to create the required maximum-minimum pairs. The function value

at each maximum is set to $f(c_j)+\varepsilon$, while that at each minimum is set to $f(c_j)+2\varepsilon$ where ε is an infinitesimally small positive value. These extrema are inserted into the sorted list after c_j. The new minima and maxima are then processed before processing c_j again. Note that when c_j is processed the second time, it will be processed as a normal join saddle. Each time a level set traversal starting from one component of the upper link reaches another component, the value of k is decreased by one. If k becomes one, then this loop saddle is classified as a false positive, and processed as a regular vertex.

- **Split Saddle:** The triangles incident on the upper link of a split saddle c_s belong to a single component. An arc is inserted into the join tree between c_s and the head of that component. The triangles in the lower star of c_s are then added to this component and c_s is set as its head.

- **Minimum:** The triangles in the upper star of the minimum belong to a single component. An arc is inserted into the join tree between the minimum and the head of this component. The minimum is set to be the head of the component.

The split tree is computed in a similar manner. Note that since the domain is already split while computing the join tree, it is not necessary to check for loop saddles and perform any additional processing while computing the split tree. The contour tree is computed using the merge procedure as described by Carr et al., and finally the newly introduced maximum-minimum pairs are merged to obtain the Reeb graph.

The advantages of using triangles to track the super- and sub-level set components instead of vertices is two-fold – (1) the first, second and third steps of the algorithm are executed during a single pass instead of three passes through the input, and (2) additional vertices need not be generated for each triangle that is cut during the split operation resulting in memory savings. This may have been necessary for computing the join and split trees using the algorithm described by Carr et al.

6.4.2 Handling large input

We now discuss a direct extension of our algorithm to compute Reeb graphs for large data that do not fit in memory. The main idea is to split the input into interval volumes that fit in memory, and compute the Reeb graph of the input scalar function by combining the Reeb graphs of the individual interval volumes.

Dividing the input

In our implementation of the out-of-core algorithm, the input is split into multiple interval volumes based on the input function. The required interval volumes are obtained by dividing the entire function range based on the number of vertices in the input. This requires sorting the vertices of the mesh based on the associated scalar values and partitioning them into intervals. Our implementation currently assumes that the vertices fit in memory and performs an in-memory sort. For example, given a memory limit of 4 GB, an input having 500 million vertices can be easily stored in memory. This is because it is enough to store the function values of each vertex together with the pointer reference to its sorted position, a total of 8 bytes per vertex.

The triangles are assigned intervals depending on the lowest function value of its vertices. One persistent file is created for each interval and triangles belonging to that interval are written into the file. This process requires one pass over the input data. Note that a triangle can belong to multiple intervals. Such a triangle has to be maintained in memory until all intervals containing it are processed.

Computing the Reeb graph

Our implementation first computes the Reeb graph of each interval volume, which now fits in memory. The intervals are processed in increasing order of function value. Each interval creates a maximum-minimum pair for every arc of the Reeb graph that spans that interval volume's upper boundary. Each extrema pair have an associated set of triangles that belong to their star. These triangles span more than one interval volume, and are retained in memory until all interval volumes that overlap their span are processed. A component consisting of an extrema pair together with its associated triangles is called a *boundary component*.

The in-memory implementation is used to compute the Reeb graph of each interval volume. In practice, many boundary components may not be accessed while processing a particular interval. Such situations occur when triangles of the boundary component are not incident on any vertex within the interval volume. Retaining all such components in memory can potentially result in exceeding the available memory. To avoid such situations, these unused boundary components are stored in persistent temporary files until they are necessary. For each boundary component, we compute the first interval volume where the vertex is incident on a triangle of the boundary component and store it.

Once the Reeb graph of all interval volumes are computed, the Reeb graph of the input is constructed by merging the additional maximum-minimum pairs that were created.

Handling large interval volumes

A potential problem with the method described above to handle large data is that, storing the entire interval volume in memory is not feasible when it is large. Such a situation may occur, for example, when a single level set is large. We have noticed that in practice, such interval volumes typically consist of multiple components. Our implementation stores these components in persistent storage and loads them into memory only when required. While this strategy handles almost all memory issues, this method will still run into trouble if a single level set component cannot fit into memory. Also, if large level sets are common then the I/O overhead increases, potentially slowing down the Reeb graph computation. A possible solution we are currently exploring in order to avoid such issues is to use the collapse operation proposed by Harvey and Wang [38], but applied to a single level set. Since this operation essentially collapses triangles, it reduces the size of the level sets, thus allowing the algorithm to handle large level set components.

Chapter 7

Computational Experiments

In this chapter, we report the performance of our algorithms and compare them to existing algorithms. The in-memory variant of the RECON algorithm[1] is implemented in C++, while we use Java to implement the out-of-core version. The cylinder map algorithm (CMAP) is implemented in Java[2] and requires the data to be in memory. The sophisticated data structures used in the sweep algorithm did not result in an efficient implementation. We therefore exclude the sweep algorithm from this discussion.

We evaluated the performance of our implementations on an Intel Xeon workstation with a 2.0 GHz processor and 16 GB main memory. For all comparisons with existing algorithms, we used the implementations provided by the respective authors. In cases where the implementation was not available, we use the timings from the corresponding paper.

7.1 In memory experiments

In this section, we discuss the performance of our algorithms when working with data in memory. First, we report the performance results for two- and three-dimensional data. We then show the scalability results in higher dimensions, and discuss robustness in the presence of noise.

[1]Implementation available at *http://vgl.serc.iisc.ernet.in/software/software.php?pid=003*
[2]Implementation available at *http://vgl.serc.iisc.ernet.in/software/software.php?pid=001*

2D Model	# Triangles	# Critical Points	# Potential Loop Saddles	# Loops	Time taken (sec) RECON	ONLINE	CMAP	RAND
Youthful	3.4M	27,627	5,588	506	2.4	6.0	47.2	54.0
Neptune	4.0M	1,752	563	3	2.6	8.7	42.0	37.3
Awakening	4.0M	17,031	2,360	1,643	2.4	6.7	41.4	51.8
Day	6.0M	104,898	12,546	2,161	4.3	10.3	91.3	67.5
Dawn	6.6M	91,640	8,592	757	4.5	11.5	73.2	71.8
Lucy	28.0M	9,521	2,462	15	34.2	60.1	*Mem*	*Mem*

Table 7.1: Reeb graph computation time for various 2D input. The Reeb graph was computed for the height function defined by the y-axis. *Mem* denotes that the algorithm ran out of memory when trying to compute the Reeb graph. RECON is at least 50% faster than ONLINE and at least 10 times faster than CMAP and RAND.

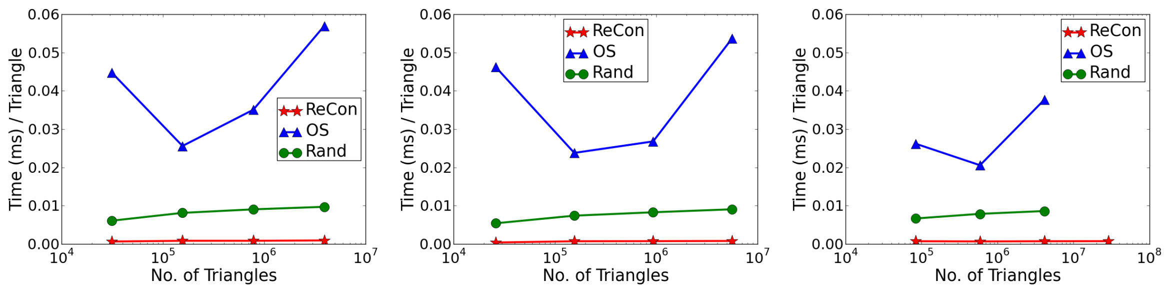
7.1.1 2D and 3D data

Table 7.1 shows the time taken to compute the Reeb graph for surface meshes. These models were obtained from the Stanford data archive [2]. We compare RECON and CMAP with the online algorithm (ONLINE) [55] and the randomized algorithm (RAND) [38]. ONLINE exhibits the best performance for 2D meshes among existing algorithms. RAND is the fastest known algorithm for generic input. While the performance of CMAP is comparable with RAND, it does not match the performance of ONLINE. However, RECON is an order of magnitude faster than both CMAP and RAND, and approximately twice as fast as ONLINE.

We use the data sets from the Aim@Shape repository [1] for our experiments with three-dimensional data. Table 7.2 compares RECON and CMAP with RAND and the loop surgery algorithm (LS) [70] for these data sets. LS is the fastest known algorithm for 3D data. We do not compare with ONLINE since it was previously shown to perform poorly for such input [70]. For 3D input, RECON performs at least an order of magnitude faster than both CMAP and RAND, while it performs at least as fast as LS. LS first computes the Euler characteristics of the 2D boundary of the input. If it detects that the boundary has no loops, then the contour tree is directly computed. The identification of loop saddles is performed only when a loop is detected in the input's boundary. Note that RECON performs at least twice as fast as LS in the latter case, for example in the Skull, Post and CubeHoles datasets.

3D Model	# Triangles	# Critical Points	# Potential Loop Saddles	# Loops	Time taken (sec) RECON	LS	CMAP	RAND
Fighter (3D)	143,881	6,787	1,717	0	0.2	0.1	123.8	6.5
Blunt fin	451,601	1,921	560	0	0.2	0.3	23.3	12.5
Bucky Ball	2,524,284	6,664	1,230	0	1.2	1.6	197.9	63.6
Plasma	2,646,016	4,719	935	0	1.5	1.9	396.3	132.7
SF Earthquake	4,198,057	20,655	529	0	2.4	2.8	598.1	166.9
Skull	336,296	26	15	2	0.1	0.3	3.4	2.2
Post	1,243,200	247	64	0	0.4	0.9	13.0	14.5
CubeHoles	2,355,234	2,402	1,300	1,200	3.2	*Seg*	97.7	23.0

Table 7.2: Reeb graph computation time for various 3D input. The scalar function was provided with the dataset. RECON is at least one order of magnitude faster and up to two orders of magnitude faster than CMAP and RAND. The time taken to compute the Reeb graph is comparable for RECON and LS. *Seg* indicates that the code exited with a segmentation fault.



(a) 4D data (b) 5D data (c) 6D data

Figure 7.1: Comparison of RECON, CMAP and RAND algorithms for different sized Sierpinski simplexes in four, five, and six dimensions. The time required to process each triangle remains consistent for RECON with increase in size as well as dimension of the input. Further, the time required is lower than that for CMAP and RAND

7.1.2 Higher dimensional data

Figure 7.1 compares the performance of RECON, CMAP, and RAND algorithms for 4-, 5- and 6-dimensional data. The input to these experiments were a set of Sierpinski simplexes in the corresponding dimension together with the height function. The number of loops in the Reeb graph also increases with increasing input size for these data sets. For the largest 6-dimensional model (s6d-7) with approximately 28.8 million triangles, CMAP algorithm threw a memory exception, while the RAND algorithm ran out of memory and started thrashing. These plots show that RECON not only performs much better than the other algorithms, but also exhibits a consistent performance with increasing dimensions.

Model	# Triangles	# Loops	Time taken (sec)
s4d-7	0.8M	1x10^5	0.7
s5d-6	0.9M	1x10^5	0.7
s4d-8	3.9M	5.8x10^5	3.6
s5d-7	5.6M	5.6x10^5	4.4
s6d-6	4.1M	2.9x10^5	2.9
Lucy	28.0M	15	34.2
s6d-7	28.8M	2x10^6	21.2

Table 7.3: RECON exhibits consistent performance for similar sized models with different number of loops. The model sxd-y denotes an x-dimensional Sierpinski simplex subdivided y times.

7.1.3 Robustness of RECON

For the remainder of this section, we consider only the RECON algorithm in our analysis. In Table 7.3, we group models of approximately same size, but with different number of loops in the Reeb graph. Notice that RECON's execution times do not change significantly when the number of loops in the input changes. This is true even when the number of loops changes from 15 in Lucy to approximately 2 million in the case of s6d-7.

Tables 7.1 and 7.2 also show the number of vertices classified as critical points along with the number of potential loop saddles, and the actual number of loops in the input. Note that for 2D input, the split saddles are also classified as potential loop saddles. In the case of 3D input, majority of the points classified as potential loop saddles correspond to regular points on the boundary. In both cases, these false positives are handled while performing the breadth-first traversal to split the input. We also observe that the additional processing does not significantly impact computation time. For example, in the case of the *Day* model, which contains the largest number of false positives (greater than 10,000), RECON requires less than 10 milliseconds to discard these vertices.

Our 2D and 3D experiments were performed on real world data sets. Also, most of these data sets are noisy, as can be seen by the number of critical points in them. In order to test the robustness of the algorithm under the stress of intense noise, we artificially introduced Gaussian noise to the input, and computed the Reeb graph for the resulting data set. Table 7.4 shows the results from this experiment. Note that in the Dawn model, which was already noisy to begin with, adding additional noise increased the number of critical points by a factor of

Dimension	Model	# Critical Points	Time taken (sec) RECON	ONLINE	LS
2D	Dawn*	1,924,221	10.8	26.8	NA
	Lucy*	7,520,211	71.2	585.9	NA
3D	Plasma*	6,765	1.5	NA	2.1
	SF Earthquake*	46,908	3.0	NA	3.1

Table 7.4: Reeb graph computation times for noisy versions of various models (denoted by *).

20 resulting in approximately 2 million critical points. This accounts for about 60% of the input vertices. Similarly, for the Lucy model, the increase in the number of critical points is almost three orders of magnitude and consists of more than half the input vertices. We notice that RECON performs efficiently even in such extreme scenarios. The running time doubles for RECON while the performance of ONLINE and LS reduces significantly. These observations hold for the remaining data sets also. For a given input size, the time required to sort the input vertices and identify critical points remains constant, and contributes to about 50% of the total running time for the above data sets. The increase in number of critical points, caused due to the introduction of noise, mainly affects the steps corresponding to the identification of false positive loop saddles, and the merge procedure that constructs the contour tree from the join and split tree. The time taken by these two operations, which is less than 1% of the total running time, increases proportionally with the number of critical points. The increase in running time of the algorithm is primarily due to these two steps.

7.1.4 Storing triangle adjacencies

We now discuss a space-time trade-off issue related to the in-memory implementation of RECON. The implementation stores the star of each vertex as well as the triangle adjacencies using the triangle-edge data structure. These triangle adjacencies can be obtained from the star, and thus it is not necessary to explicitly store the adjacencies. However, this results in increased effort for performing traversals, since the algorithm has to process more triangles to determine adjacencies, and hence the overall computation time increases. But an advantage of this approach is that the algorithm can process larger data sets in memory because of the $6n$ space that is saved by not storing triangle adjacencies. Table 7.5 compares the running

Model	# Triangles	Time taken (sec) RECON	Time taken (sec) RECON′	Memory saved
Dawn	6.6M	4.5	5.0	338 MB
Lucy	28.0M	34.2	34.5	1.3 GB
Plasma	2.6M	1.5	2.5	91 MB
SF Earthquake	4.1M	2.4	2.8	135 MB

Table 7.5: Comparison of running times between RECON′, which uses the star of a vertex to find triangle adjacencies, and RECON.

			RECON			ONLINE		
Model	# Triangles (millions)	Function	Creating interval volumes	Reeb graph computation	Total Time	Finalizing input	Reeb graph computation	Total Time
David	56M	x	37.0s	3.0m	3.6m	2.6m	2.1m	4.7m
		y	41.0s	3.1m	3.8m	2.6m	2.2m	4.8m
		z	27.0s	2.7m	3.2m	2.6m	14.0m	16.6m
St. Matthew	372M	x	5.5m	21.4m	26.9m	25.0m	15.0m	40.0m
		y	3.9m	22.8m	26.7m	25.0m	3.8h	4.2h
		z	2.8m	22.4m	25.2m	25.0m	16.0m	41.0m
Atlas	507M	x	7.2m	34.3m	41.5m	*	*	*
		y	5.8m	32.7m	38.5m	*	*	*
		z	7.3m	35.3m	42.6m	*	*	*

Table 7.6: Performance of the out-of-core implementation of RECON for large 2D data sets. RECON is faster than the online algorithm for all inputs, up to a factor of 8 for the St. Matthew data set (y-coordinate). * denotes that the running time for the data set is not available.

time when triangle adjacencies are stored (RECON) and when they are not stored explicitly (RECON′).

7.2 Experiments with large data

Table 7.6 shows experimental results of the out-of-core implementation. We compare it with the online algorithm for large data sets available from the Stanford data archive [2]. Note that for the Atlas model which has approximately 500 million triangles, the total time taken to compute the Reeb graph is approximately 40 minutes. The timings for the online algorithm are as reported in the paper [55]. The online algorithm requires finalization of the input to be performed only once irrespective of the input function used to compute the Reeb graph. Since our method constructs the interval volumes based on the input function, it has to perform this operation once for each function.

Chapter 8

Visualization of Reeb graphs

Effective presentation of Reeb graphs is crucial for its application to interactive exploration of scalar fields. An important task for effective presentation is the design of a useful layout to visualize the Reeb graph. A method for controlled and user-directed 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. We first describe a method to simply the Reeb graph in Section 8.1. Next, we provide two layout schemes for visualizing them in Section 8.2. Finally, in Section 8.3, we describe the Topoview software, which uses the tools described here to allow users to interactively explore scientific data using its Reeb graph.

8.1 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 [29] resulting in a multiresolution representation of the input scalar field. Features can also be ordered and removed based on geometric measures like *hyper-volume*, which is defined as the integral of the input scalar function over the enclosed volume [14]. For the remainder of this section, we will use persistence to denote the simplification measure. However, we note that our simplification procedure can also be used with other measures.

Our Reeb graph simplification approach extends the contour tree simplification algorithm [14] to include critical point pairs that create / destroy loops. In addition to the *leaf pruning* and *node reduction* simplification operations, the simplification algorithm performs an additional *loop pruning* operation on the Reeb graph. Leaf pruning removes a leaf and the incident arc from the Reeb graph. Node reduction removes a degree-2 node by merging the two adjacent arcs. The loop pruning operation removes an edge that is part of a loop, thus decreasing the number of loops in the Reeb graph by one. The algorithm does not prune a leaf or a loop if this operation results in a degree-2 node which is a maximum or a minimum. Such a situation is possible when both the neighbors of the node have a function value that is either less than or greater than that of the node, respectively. This condition ensures that the simplified graph conforms to the structural properties of Reeb graphs.

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.

8.2 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.

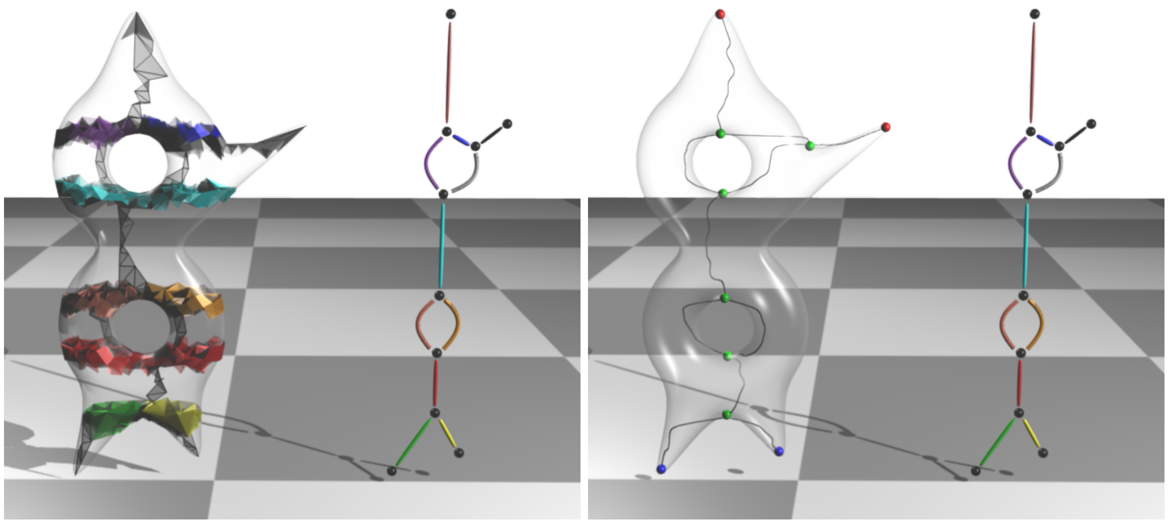

(a) (b)

Figure 8.1: Embedded Reeb graph layout. **(a)** The triangles traced by the cylinder map algorithm while computing the Reeb graph. **(b)** The embedded layout obtained by tracing the triangles in this path.

8.2.1 Embedded Reeb graph layout

In the cylinder map algorithm, 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.

Figure 8.1(a) shows the set of triangles traced by the algorithm while computing the Reeb graph of the height function defined on the solid 2-torus. The arcs of the Reeb graph correspond to the path between two critical level sets (represented by the different colored triangles). Figure 8.1(b) shows the embedded layout of the Reeb graph that is obtained from this path. The blue, green and red nodes in this figure correspond to minima, saddles and maxima respectively.

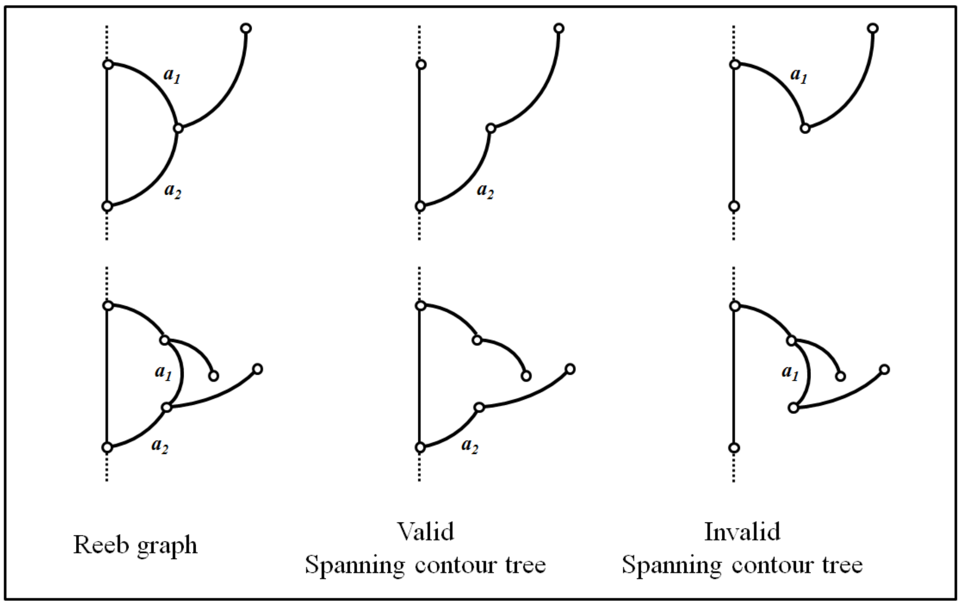


Figure 8.2: The spanning contour tree of a Reeb graph is structurally similar to a contour tree. Removal of a_1 results in a spanning contour tree. Removing a_2 results in a tree with an invalid degree-2 node.

8.2.2 Feature directed radial layout

We build upon the toporrery layout proposed for contour trees [54] to obtain a layout for Reeb graphs. The extension to Reeb graphs is non-trivial because of the presence of loops. This difficulty is overcome 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 8.2, removing arc a_1 results in a spanning contour tree. Removal of a_2 also results in a spanning

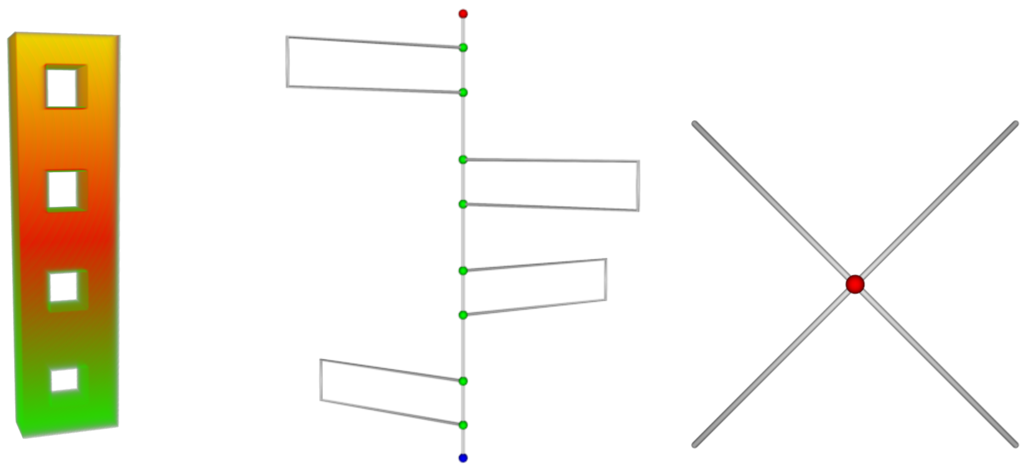


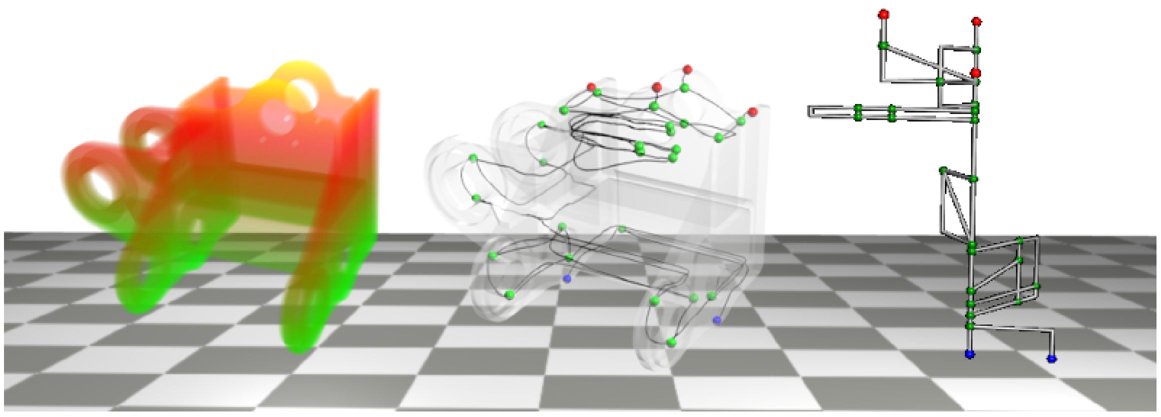
Figure 8.3: Reeb graph of the height function defined on a solid 4-torus. **(a)** Volume rendering of the input **(b)** Side view of the radial layout of the Reeb graph. **(c)** Top view of the radial layout of the Reeb graph.

tree, but one that does not correspond to a valid contour tree. Note that there exists multiple spanning contour trees for a single Reeb graph.

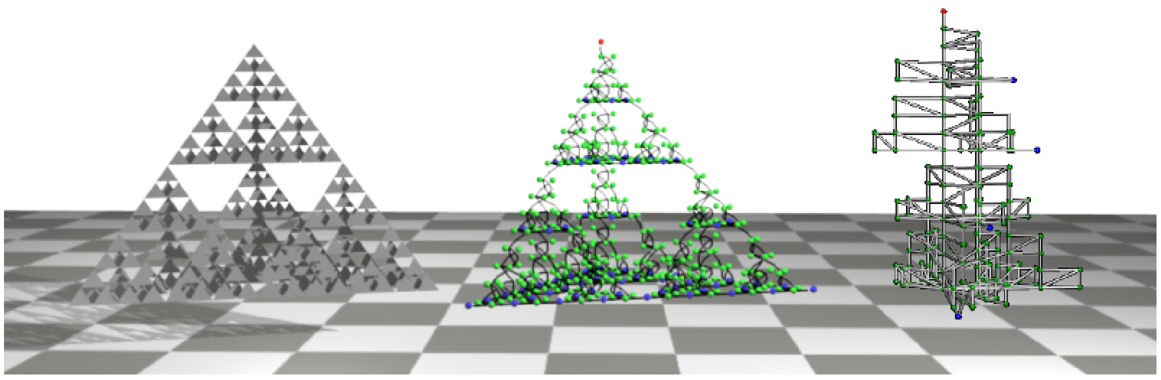
We use the simplification procedure from the previous section to obtain the spanning contour tree. This is accomplished by removing the loop edges that are pruned. This process not only guarantees that the resulting tree satisfies the structural properties of a contour tree, but also enables a multi-resolution representation of the Reeb graph. Such a representation enables interactive simplification and presentation of the Reeb graph.

A branch decomposition is an alternate representation of a contour tree that explicitly stores the topological features and their hierarchical relationship [54]. 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.

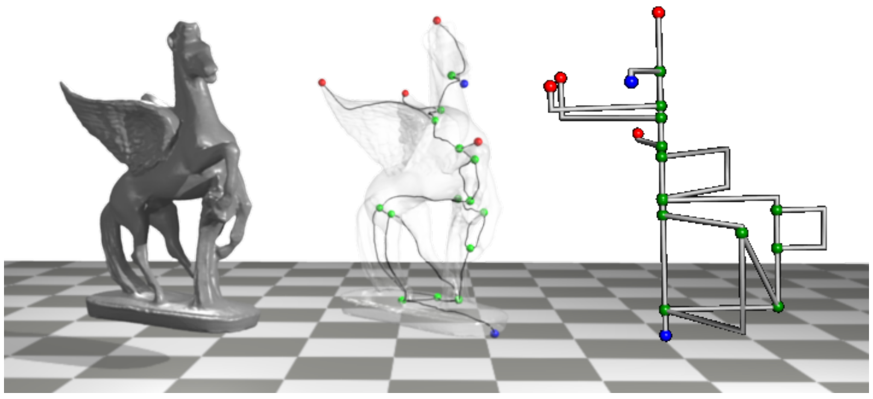
All branches of the spanning contour tree are drawn as L-shaped polylines and the y-coordinate corresponds to function value. The (x,z) coordinates are computed for each branch using a radial layout scheme. The root branch is located at the origin and other branches are placed on concentric circles centered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



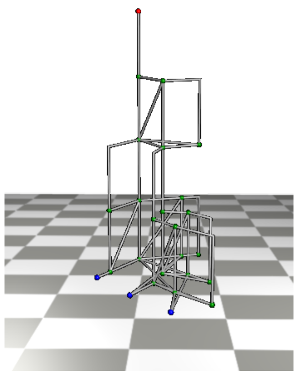
(a)



(b)



(c)



(d)

Figure 8.4: Embedded and radial layout schemes for the Reeb graph computed for the height function defined on various models. **(a)** Volume rendering of a 3D CAD model along with the simplified Reeb graph. **(b)** Reeb graph of a non-manifold mesh representing a three-dimensional Sierpinski simplex subdivided four times. **(c)** Reeb graph of a 2D Pegaso model. **(d)** Reeb graph of the height function defined on a 4D Sierpinski simplex subdivided twice.

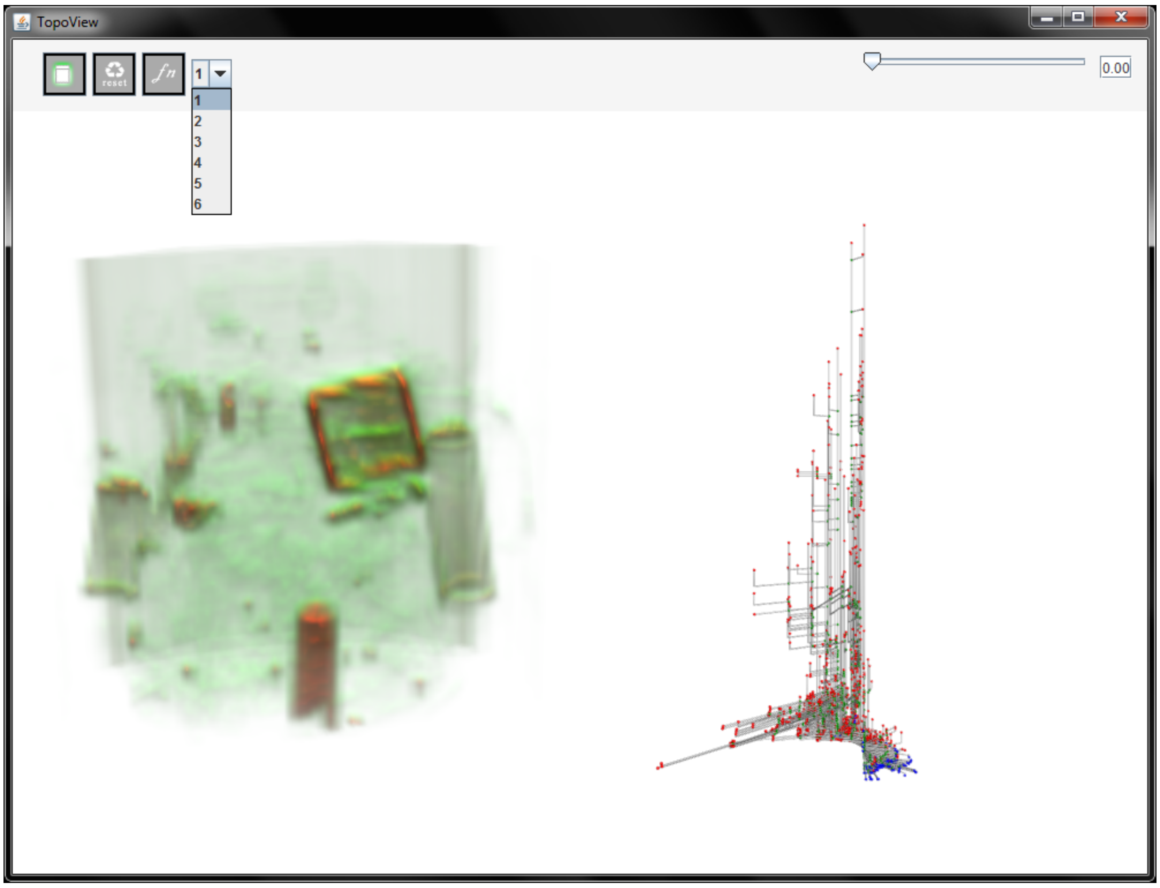
that connect to an interior node of a first-level branch are placed in the second concentric circle within a wedge centered at a level-one branch. The angle subtended is proportional to the number of descendant branches. In order to avoid intersections when the non-tree arcs are inserted, we include a dummy branch for each loop arc before calculating the angular wedge subtended at each branch. Figure 8.3 shows the layout of the Reeb graph computed for the height function defined on a 4-torus.

Figure 8.4 illustrates the embedded and radial layouts of the Reeb graph computed for height functions defined on 2D, 3D, and 4D models. Note that the Sierpinski simplexes in three and four dimensions are also non-manifold.

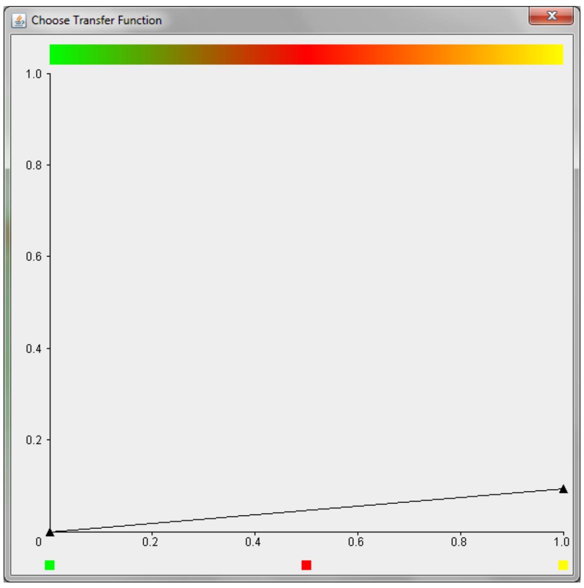
8.3 Topoview

We now describe our Topoview software that allows users to interactively explore 2D and 3D scalar fields using its Reeb graph. The software was developed using Java and OpenGL. We use the Backpack data [3] to illustrate the different operations supported by Topoview. This data was obtained after performing a CT scan of a backpack filled with items. The interface of the software is shown in Figure 8.5(a). The input scalar function is rendered on the left of the screen, while its Reeb graph is displayed on the right. A volumetric input is rendered using the ray casting technique. Topoview provides access to multiple transfer functions, which can be modified using a transfer function editor. Figures 8.5(b) and 8.5(c) show two of the default transfer functions in a transfer function editor.

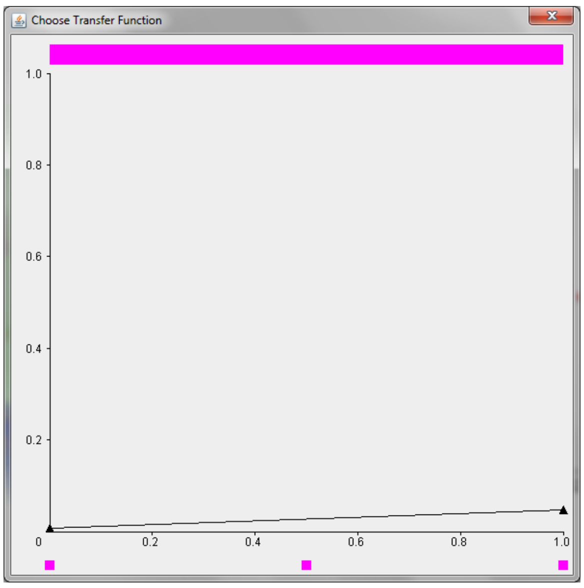
The slider on the top right of the tool allows users to simplify the Reeb graph. This operation is shown in Figure 8.6(a). The users can explore the input by selecting arcs of the Reeb graph. Figure 8.6(b) shows an instance where two arcs of the Reeb graph are selected. The items in the backpack corresponding to the selected arcs are highlighted in the volume. Topoview provides an interface to assign different transfer functions to the input subdomain corresponding to different groups of arcs of the Reeb graph. Figure 8.6(c) shows this operation of assigning another transfer function to the two arcs selected in Figure 8.6(b). The result of



(a)

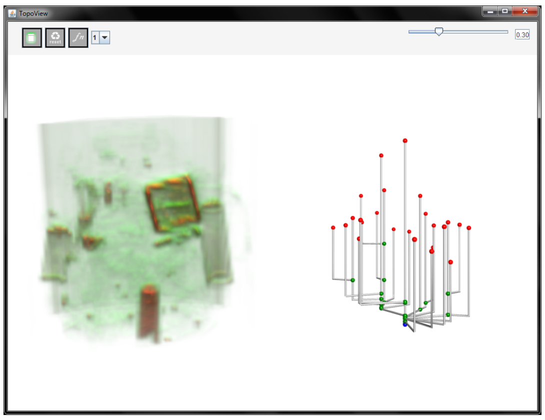


(b)

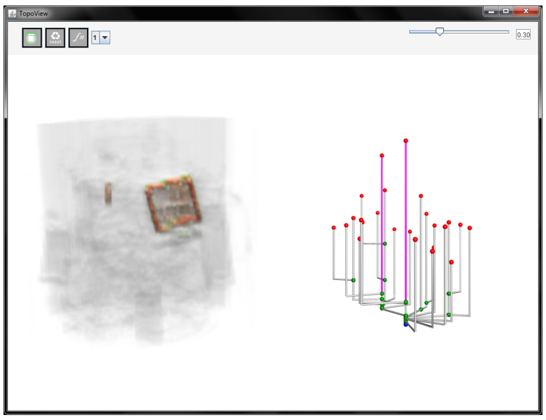


(c)

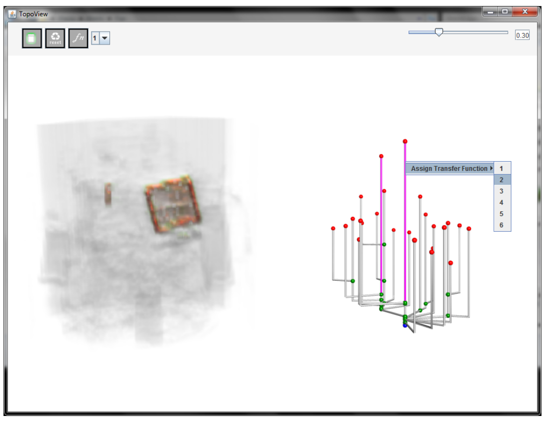
Figure 8.5: Topoview software. **(a)** The software's user interface. **(b)** and **(c)** Transfer function editor showing two transfer functions. A transfer function can be chosen to be edited by selecting the appropriate number from the drop down menu, and selecting the "*fn*" button.



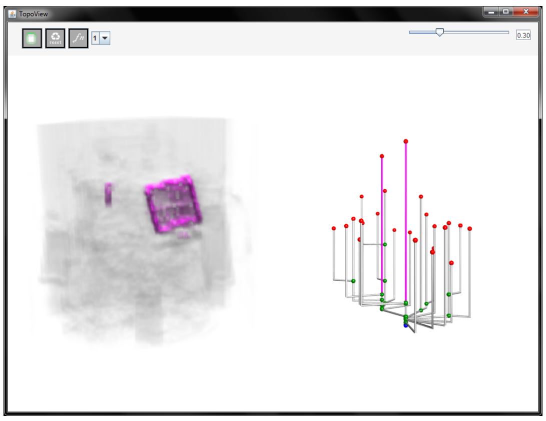
(a)



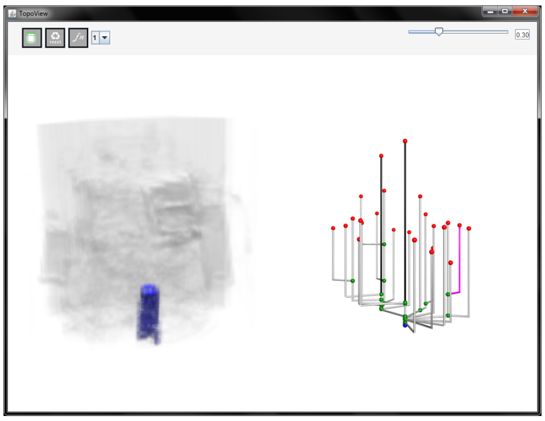
(b)



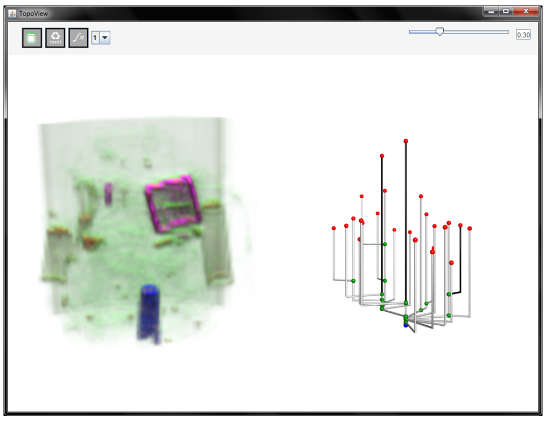
(c)



(d)



(e)



(f)

Figure 8.6: Operations supported by the Topoview software. **(a)** The slider on the top right is used to simply the Reeb graph. **(b)** Users can explore the input by selecting multiple arcs in the Reeb graph. Here, two arcs corresponding to two items in the backpack are selected. **(c)** Different transfer functions can be assigned to the selected arcs. Here, the transfer function No. 2 is assigned to the subdomain corresponding to the selected arcs. **(d)** Result of the operation performed in (c). **(e)** An arc corresponding to another item in the backpack is selected and assigned a different transfer function. **(f)** Rendering of the entire volume which highlights three items in the backpack.

this operation is depicted in Figure 8.6(d). Figure 8.6(e) shows the result of applying a different transfer function to another arc. Figure 8.6(f) shows a volume rendering where three items in the backpack are highlighted using different transfer functions. The rest of the volume is rendered using the default transfer function.

Chapter 9

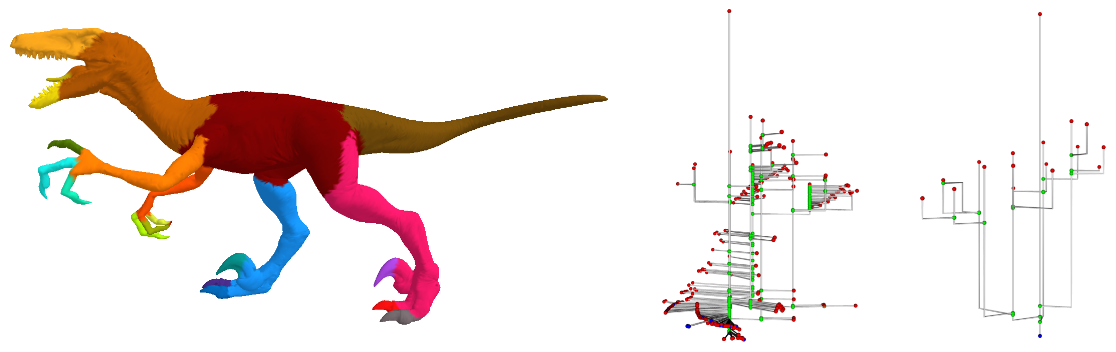
Application of Reeb graphs

In this chapter, we describe four applications of Reeb graphs to visualization and computer graphics.

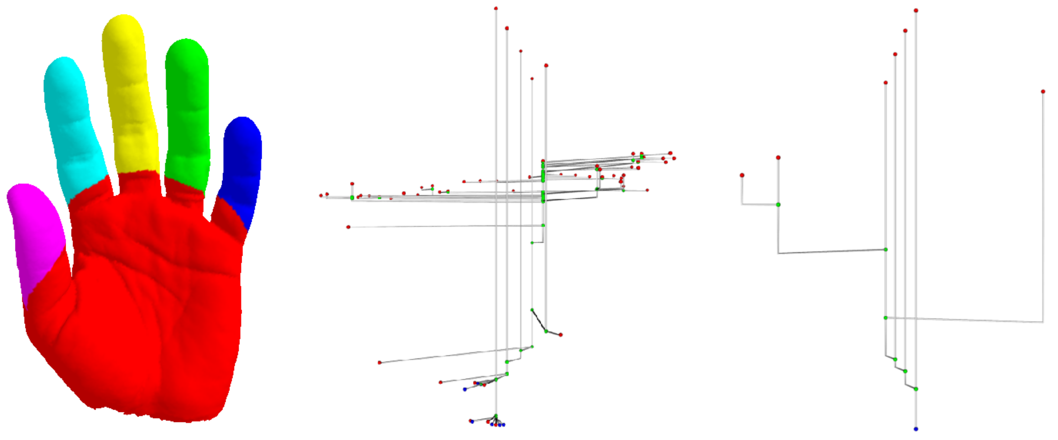
9.1 Segmentation of surface meshes

The cylinders partition the input mesh into potentially interesting features. A minor extension of the cylinder map 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 a 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).

A common scalar function defined on surface meshes in order to extract its features is the *geodesic function*, in which the function value at a vertex is equal to the geodesic distance between that vertex and a root vertex. Zhou et al. [82] use the geodesic function and decompose the surface mesh based on the critical points of this function. Hilaga et al. [42] use the average geodesic function, and use multi-resolution Reeb graph to identify similarity between meshes. When using the *average geodesic function*, the function value at a vertex is defined as the average geodesic distance from a given vertex to all other vertices on the surface. Zhang et



(a)



(b)

Figure 9.1: Using Reeb graphs to segment surfaces into features of interest. **(a)** Partition induced by the Reeb graph on the Raptor model. **(b)** Segmentation of the Olivier hand. The Reeb graph and the simplified Reeb graph used to segment the surface are also shown beside the models.

al. [80] present an automatic parameterization method that segments a surface mesh into a set of patches. They use the Reeb graph of the average geodesic function defined on a surface mesh to obtain the segments. The advantage of the average geodesic function is that it is invariant to deformations of the mesh. In this thesis, we use the average geodesic function to identify interesting features of a surface mesh.

We first compute the average geodesic function on the input mesh [79], and use the Reeb graph computed on this function to segment the surface. Figure 1.1(b) visualizes the average geodesic function computed on the raptor model using a color map. Figure 9.1(a) shows a segmentation of the raptor model into its key features such as the main body, tail, legs and talons, jaws, and tongue. Figure 9.1(b) shows the Olivier hand model partitioned using the Reeb graph. 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 using the interface provided by the Topoview software, but can be automated with further geometric processing.

9.2 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 [32, 35], which is the preimage of a given range of scalar values.

We first convert the rectilinear grid into an unstructured mesh by decomposing each cube into a set of tetrahedra following the method outlined by Sohn [63]. 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

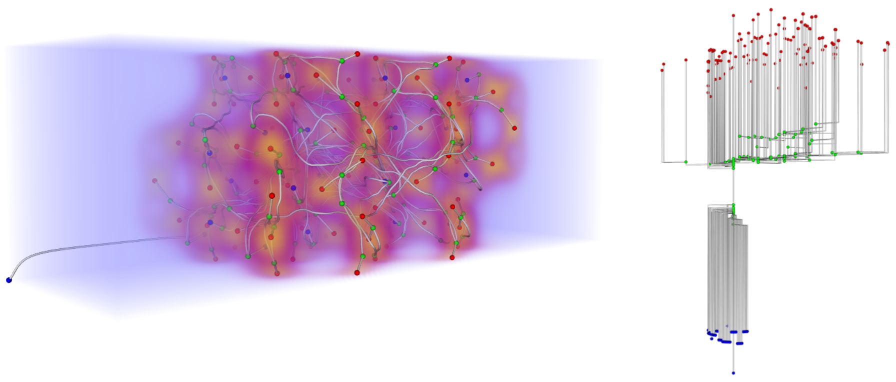
$$\begin{aligned} f(x,y,z) = & f_{000}(1-x)(1-y)(1-z) + f_{001}(1-x)(1-y)z \\ & + f_{010}(1-x)y(1-z) + f_{011}(1-x)yz \\ & + f_{100}x(1-y)(1-z) + f_{101}x(1-y)z \\ & + f_{110}xy(1-z) + f_{111}xyz, \end{aligned}$$

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 are 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 [63]. 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.

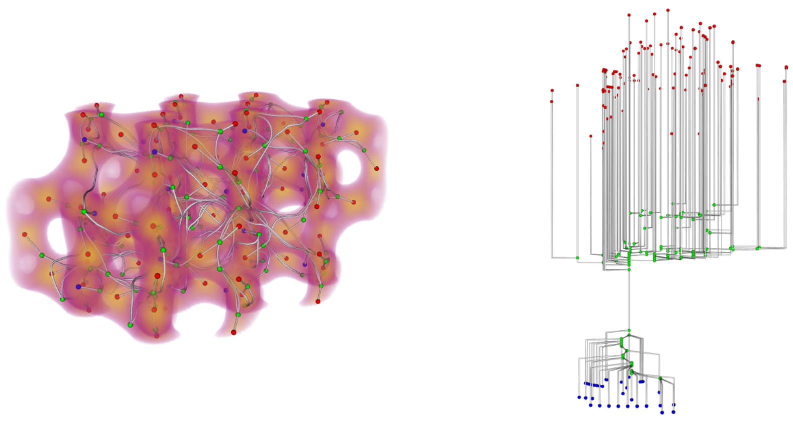
We use the tetrahedral mesh obtained from the above-described decomposition and compute its Reeb graph. Generating the mesh takes time linear in the size of the grid. Also, the number of tetrahedra in the generated mesh is linear in the number of grid nodes. Thus, the time complexity for computing the Reeb graph for a structured mesh remains unchanged. Figure 9.2(a) shows a volume rendered image of the silicium data set [3] along with its Reeb graph embedded within the volume. The Reeb graph was computed for the original dataset. Figures 9.2(b) and 9.2(c) 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 9.2(c).

9.3 Spatially-aware transfer function design

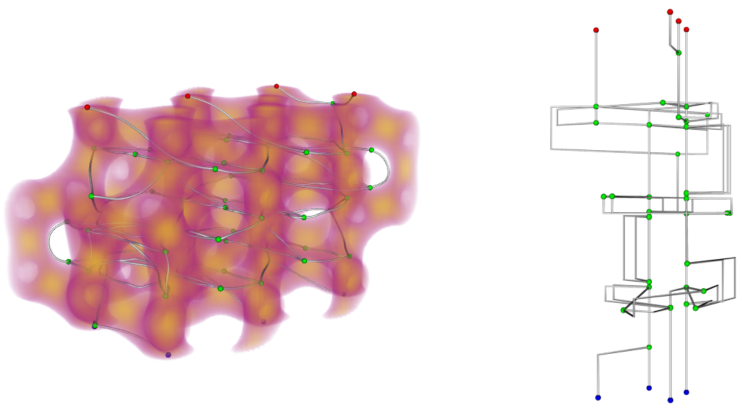
Transfer functions maps the scalar field of a volumetric mesh to optical properties, such as color and opacity. Designing good transfer functions for volume rendering is essential to obtain meaningful images of the volume. Topology based methods, in particular Reeb graphs, have been used to design effective transfer functions for volume rendering. Each cylinder can be accessed using arcs of the Reeb graphs and assigned individual colors and opacity based on different properties of the arc, thereby creating a volume rendered image that distinctly highlights the user-specified areas of the volume.



(a)



(b)



(c)

Figure 9.2: Visualization of the Silicium dataset: volume rendered images with the embedded Reeb graph and the radial layout. **(a)** The rectilinear volume. **(b)** Interval volume. **(c)** Height function (y-coordinate) defined on the interval volume.

Prior Work

Fujishiro et al. [33] propose two methods to automatically design transfer functions based on critical points. Assume the critical points are sorted in the increasing order of their function values. In their first method, the change in color is uniform between two consecutive critical points, and is increased by a constant value in the next interval. The opacity remains constant for the entire volume. In their second method, they design a step function, where in the color and opacity values increase at critical points. Takahashi et al. [65] follow a similar approach which essentially decreases the hue uniformly attenuated with a jump at critical values. The opacity is increased, with hat-like patterns replacing the steps.

Takeshima et al. [68] use additional properties of the Reeb graph, such as the level of nesting of an arc, the number of loops, distance between isosurfaces etc. to assign color and opacity at different function values. Weber et al. [75] extend this approach, to allow assigning different transfer functions to subdomains in the input corresponding to different arcs of the contour tree. Zhou et al. [81] use a residue flow model based on Darcys Law to control distributions of opacity between branches of the contour tree. Color is assigned depending on the topological properites of the branches of the contour tree.

Spatially-aware transfer function design

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 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

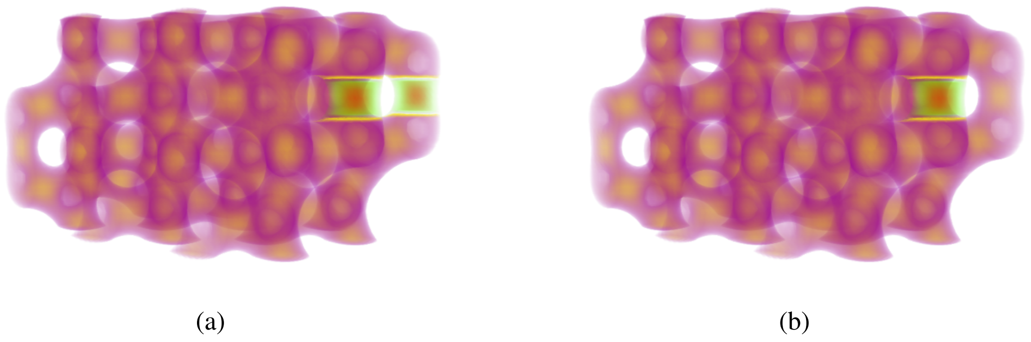


Figure 9.3: Reeb graph computed on y-coordinate function in the silicium data set is used to highlight interesting geometric features. **(a)** Highlight the volumetric region corresponding to a loop in the Reeb graph by designing a different transfer function. **(b)** Highlight an individual atom by selecting one arc in the Reeb graph and designing a different transfer function for the corresponding cylinder.

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 9.3(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 9.3(b) highlights a single atom in the silicium data set. In this case, we select one arc from the loop.

9.4 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.

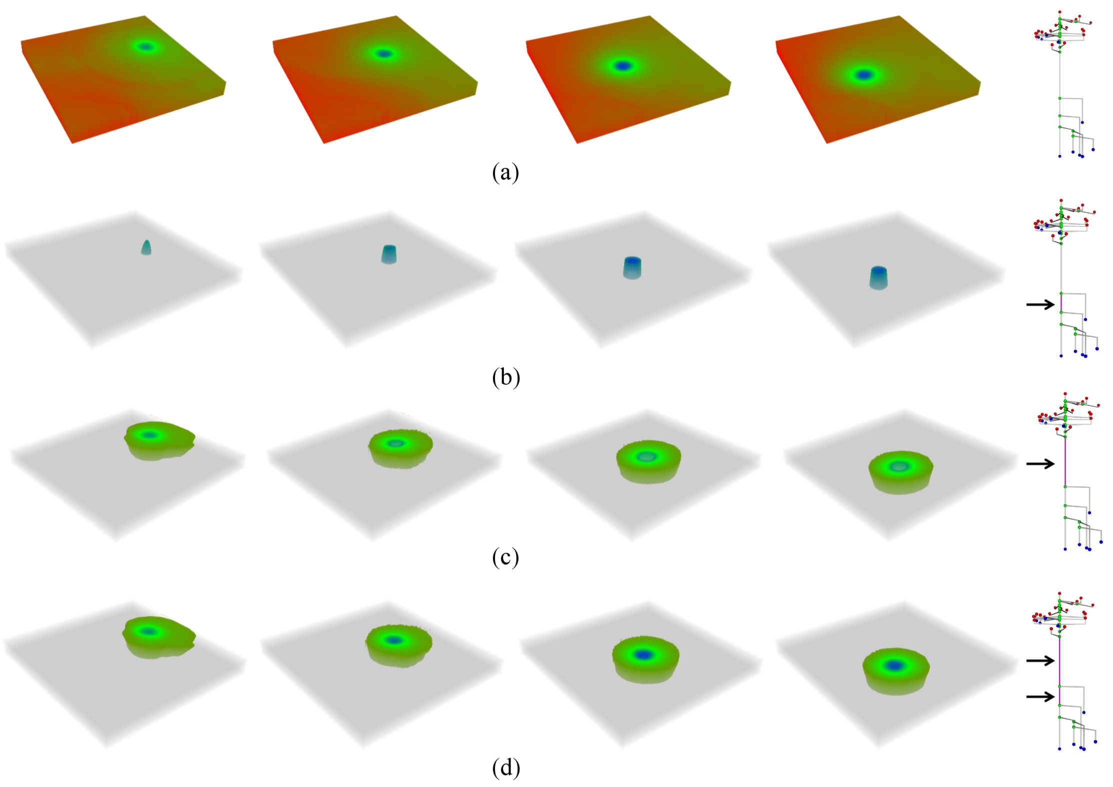


Figure 9.4: Exploring the hurricane Isabel data using Reeb graphs. **(a)** The input is shown as a set of volumes at time steps $1, 16, 32, 40$ and the Reeb graph is shown on the right. **(b)** A transfer function is designed specifically for the cylinder corresponding to the arc selected in the Reeb 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.

Figure 9.4 shows results of our experiment on the pressure field in the hurricane Isabel data set [74]. Figure 9.4(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 9.4(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 9.4(c) corresponds to region of the hurricane surrounding the eye. Figure 9.4(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.

Chapter 10

Topological Saliency

Topological methods used for analyzing, visualizing, and exploring scalar fields often involve simplification of the input as an integral part of its process. All the applications that were presented in the previous chapter perform an initial simplification of the input. Such simplification is necessary for effective application of other topological data structures such as Morse-Smale complex [28] also. The process of simplification, as described in Chapter 8.1, involves assigning a measure of importance to different features, followed by removing features of low importance. Here, features correspond to critical points of the function. While the notion of persistence, which we have used so far, can effectively describe the importance of a feature with respect to an input scalar function or filtration, it is somewhat oblivious to other geometric information not encoded in the input function. In particular, it does not reflect how important a feature is relative to other features in its neighborhood. In this chapter, we follow an approach similar to saliency models used in image and geometric mesh analysis to introduce a notion of topological saliency for features in the input that captures the relative importance of a feature within a spatial neighborhood.

The rest of this chapter is organized as follows. We discuss related work in Section 10.1 and define topological saliency in Section 10.2. Finally, in Section 10.3 we present two applications of topological saliency for analyzing and visualizing scalar functions.

10.1 Related work

Many models for obtaining salient locations in images have been proposed [43,45,49,59,71]. In particular, Itti et al. [43] propose a model that computes the saliency of a pixel in an image based on the properties of pixels in its neighborhood. Lee et al. [47] extend this model to geometric features and propose a notion of mesh saliency that captures the saliency of a point in a surface or volume mesh. It is computed as the curvature at a point weighted by the average curvature within a small neighborhood.

Our topological saliency framework can be viewed as a way to combine geometry information with topological methods. We remark that the theme of combining geometry and topology is not new. For example, Carr et al. [14] employed geometric measures computed on contour trees to find and simplify less significant features. Weber et al. [75] used Reeb graphs to identify significant features in volumetric data in order to design transfer functions for rendering the volume. The concept of topological landscapes [38,76] provides an intuitive view of the data by displaying its topological features, abstracted by the contour tree, as a terrain. In Agarwal et al. [4], the authors aim to use a descriptor function to encode certain geometric information of interest, and use topological persistence to identify geometric features from this function. The concept of "localized homology" was later proposed by computing the homology from local pieces to global pieces, so that the generators of homology classes are localized in local pieces [83]. Reininghaus et al. [58] proposed an importance measure for critical points in two dimensional scalar fields called the scale space persistence, which combines the notion of deep structure of the scale space with topological persistence. The scale space persistence is computed by accumulating the persistence values of a critical point through its evolution in the scale space.

10.2 Topological saliency

Consider the terrain shown in Figure 10.1 with seven peaks. Existing topology-based methods would ignore peak F even though it dominates a large area of the domain in the sense that it remains an important feature within a large neighborhood size. Similarly, based on persistence,

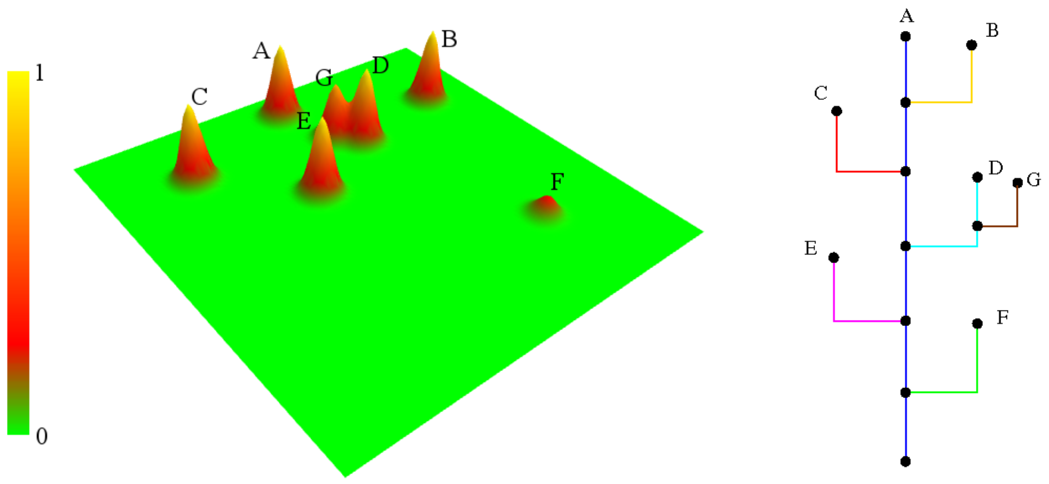


Figure 10.1: A sample terrain with seven peaks. Traditional topological methods identify peaks *A*, *B*, *C*, *D* and *E* as important. Even though peak *F* remains a lone peak for a significantly large part of the domain, it is not considered to be important. The Reeb graph for this input is shown on the right.

peaks *C* and *D* would have been declared as equally important even though *D* is surrounded by other peaks of similar height making it not as dominant as *C*. In general, spatial distribution of topological features has not been considered while measuring the size of a feature and its significance.

We address the problem raised in the above example by defining a notion of topological saliency that considers the presence or absence of other features within the neighborhood while measuring the importance of a topological feature. A feature in this chapter is always represented by an extremum (minimum or maximum) of the input scalar field. We formally define topological saliency next in Section 10.2.1. In Section 10.2.2 we introduce a saliency plot that is generated by computing the topological saliency of all features for varying neighborhood sizes. This plot can be considered as an augmentation or refinement of persistence, obtained by injecting certain spatial geometry information into it. We then describe the use of topological saliency for simplifying the input scalar field in Section 10.2.3.

10.2.1 Definition

Let the set $C = \{c_1, c_2, \ldots, c_n\}$ be the set of minima of the input function $f : \mathbb{M} \to \mathbb{R}$. Let $\mathrm{P}(i)$ denote the persistence of the topological feature created at c_i. Let $d_g(p,q)$ denote the geodesic distance between two points $p,q \in \mathbb{M}$. Consider a *r-neighborhood* $N_r(i) = \{x \in \mathbb{M} \mid d_g(x,c_i) \leq r\}$, which is the geodesic ball of radius r centered at critical point c_i. We define the *topological saliency* $\mathrm{T}_r(i)$ of the feature created at c_i as

$$\mathrm{T}_r(i) = \frac{\omega_i^i \mathrm{P}(i)}{\sum\limits_{c_j \in N_r(i)} \omega_j^i \mathrm{P}(j)},$$

where ω_j^i is a weighting function for the feature j with respect to i. The topological saliency at a maximum is defined in a symmetric manner. Two common choices of the weighting function are (a) the uniform weight $\omega_j^i = 1$; and (b) the Gaussian weight $\omega_j^i = e^{-\frac{d_g(c_i,c_j)^2}{r^2}}$, for $i, j \in [1,n]$. The topological saliency of a topological feature essentially normalizes the persistence of that feature based on the features that are present in its neighborhood. A Gaussian weighting function reduces the influence of farther-away features, while a uniform weighting function treats all features within the neighborhood equally. Unless otherwise mentioned, we use uniform weights for all experiments reported in this chapter.

Note that when computing the topological saliency of a minimum (resp. a maximum), we only consider features of the same type, *i.e.*, other minima (resp. maxima) in its neighborhood. Intuitively, critical points of different indices capture different types of features: a minimum captures a valley while a maximum captures a mountain peak. We also remark that one can extend the topological saliency to critical points of other indices (i.e, various saddle points). An index-k saddle point indicates the formation of a k-cycle. However, the meaning of a neighborhood of such features become less clear, and we leave the definition of salient index-k features for future work.

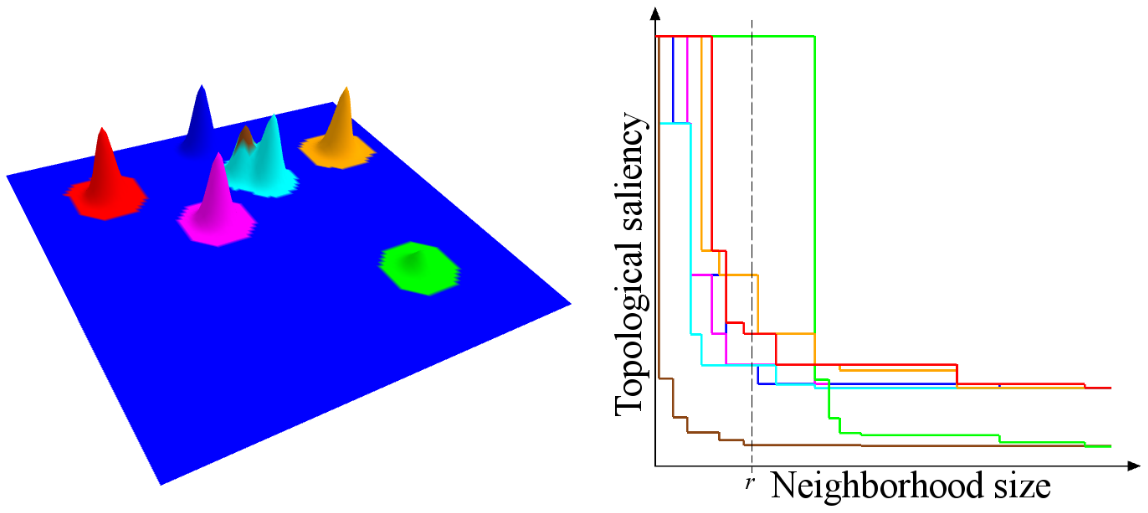


Figure 10.2: The topological saliency plot of the terrain data. The parititon of the input was obtained using its Reeb graph. Features in the input and the corresponding curve plots are highlighted using a common color. Note that the green colored peak F, which has low persistence, maintains a topological saliency of 1 up to a large value of neighborhood size.

10.2.2 Topological saliency plot

Consider the neighborhood of a feature c_i when $r = 0$. It consists of just the critical point c_i. Its topological saliency $\mathrm{T}_0(i)$ is 1. As we increase the neighborhood size r, $\mathrm{T}_r(i)$ remains at 1 until $N_r(i)$ includes another feature represented by, say c_j. At this point, the value of $\mathrm{T}_r(i)$ reduces depending on the value of $\mathrm{P}(j)$. Note that, at this value of r, $\mathrm{T}_r(j)$ also decreases simultaneously. We can continue increasing r until r equals the diameter D of the input domain $\mathbb{M}$, at which point $N_D(i)$ covers $\mathbb{M}$. Plotting the values of $\mathrm{T}_r(i)$ for all features from $r = 0$ to D, we obtain a *topological saliency plot*. Figure 10.2 shows the topological saliency plot for maxima in the terrain data from Figure 10.1.

Note that if we use the uniform weighting scheme, then when r equals the diameter D of the input domain $\mathbb{M}$, the topological saliency $\mathrm{T}_D(i)$ of c_i is simply the standard persistence $\mathrm{P}(i)$ scaled down by the total persistence $\sum_i \mathrm{P}(i)$. Hence by varying the parameter r from 0 to D, we move from a local perspective of the feature to its global perspective. One can recover the traditional persistence of a feature by looking at the corresponding value of T_D.

10.2.3 Saliency based simplification

Simplification based on persistence could possibly remove salient features. For example, if we were to simplify the terrain dataset using persistence, then the peak F may be simplified away at a small threshold.

To address this issue, we propose a saliency based simplification method, which uses the topological saliency at a fixed neighborhood size r in order to simplify features. Note that, when using topological saliency for simplification, removing a feature affects the saliency of features in its neighborhood, and hence the saliency of the affected features have to be recalculated. This can be efficiently done by obtaining the neighborhood of each feature during a preprocessing step, storing it, and updating the neighbors during the simplification process. Since we consider only the maxima or the minima to define this measure, extension of topological saliency to simplifying Reeb graphs is not straight forward. When simplifying the Reeb graph, it is possible for the simplification procedure to encounter an edge to be simplified that does not correspond to a feature. Currently, in such situation, we assign a weight of zero to that edge and simplify it immediately. This does not affect the analysis for data sets where the features are predominantly the set of maxima, or the set of minima.

As a side effect of this simplification process, we obtain a good segmentation from the resulting set of features. This is attributed to the fact that features that are close to each other get merged, as opposed to persistence based simplification, where no spatial information is used when merging an existing feature.

10.3 Applications

In this section we show two applications of topological saliency. The features in our experiments are identified by the set of maxima in the input. The partitions corresponding to them are obtained using the Reeb graph. Unless otherwise specified, in all the experiments, the input is first simplified using topological saliency before further processing.

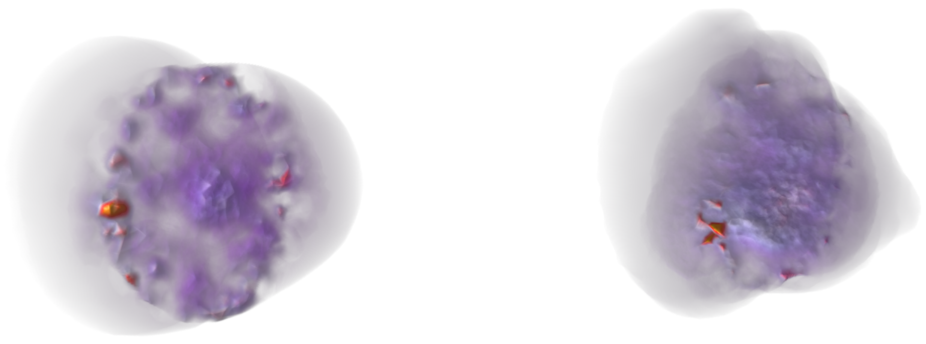
10.3.1 Significant features

The topological saliency plot can be used to identify significant features in multiple ways. In this thesis, we use topological saliency defined for a fixed neighborhood size r as a measure to order features. Applying this alternative notion of importance to the terrain dataset for the value of r shown in Figure 10.2, the features are ordered as follows: F, B, C, E, A, D and G. This notion helps resolve our problem of identifying the green peak F in this input as being most significant. Note that the brown peak G is not considered to be significant because of its neighborhood even though its persistence is similar to F. The labeling of the peaks is the same as in Figure 10.1. We now apply this method to identify breast tumors.

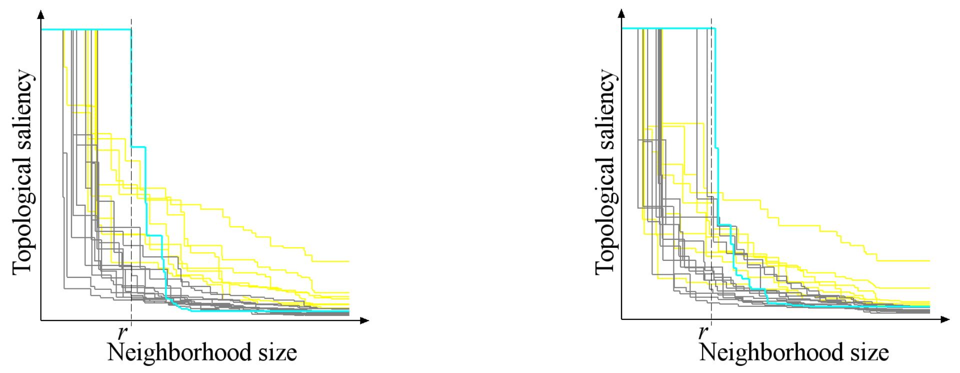
Diffuse optical tomography is used as an adjunct imaging modality for breast and brain imaging to provide functional images. Non-ionizing near infrared (NIR) light with wavelength in the range of 600-1000 nm is the interrogating medium of choice [11,34]. Typically, the NIR light is delivered and collected using fibre bundles at the boundary of tissue. These boundary measurements are used to reconstruct the internal distributions of optical absorption and scattering coefficients. The data is available as a tetrahedral mesh where the scattering coefficient at each vertex defines the input scalar function.

Figure 10.3(a) shows the volume rendering of two `breast` data sets that have a tumor. The topological saliency plots for the two data sets are shown in Figure 10.3(b). The persistence of the fibre bundles at the periphery of the volume, that are used to collect data, is higher than that of the tumor itself. Therefore, a persistence based ordering would identify one such fibre bundle as the most significant feature. Using an appropriate value for neighborhood size r, shown in the saliency plot, we are able to identify and isolate the region corresponding to the tumor as the most salient feature, see Figure 10.3(c).

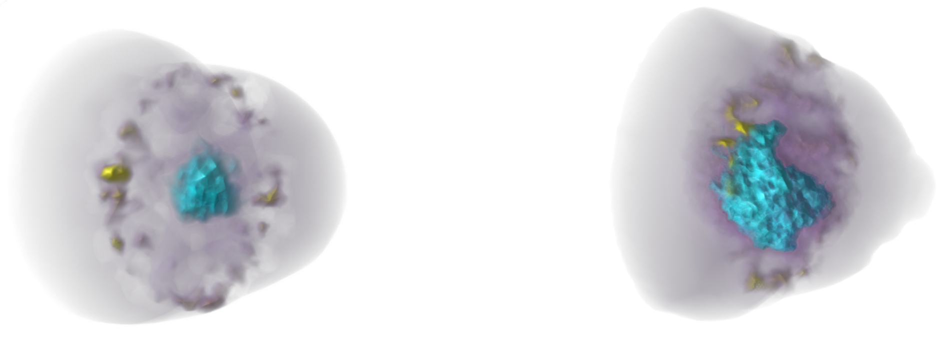
Ordering features based on its topological saliency requires choosing an appropriate value of r, which is application dependent. The user can compute the order at different perspectives, from local to global, by suitably specifying the neighborhood size r.



(a)



(b)



(c)

Figure 10.3: Identifying tumors using topological saliency. **(a)** Volume rendering of the two `breast` datasets. **(b)** The most salient feature of the dataset is highlighted in cyan in the topological saliency plot. The plots corresponding to the sensors are colored yellow. **(c)** The most salient feature corresponds to the region containing the tumor. The volume rendering highlights the most salient feature.

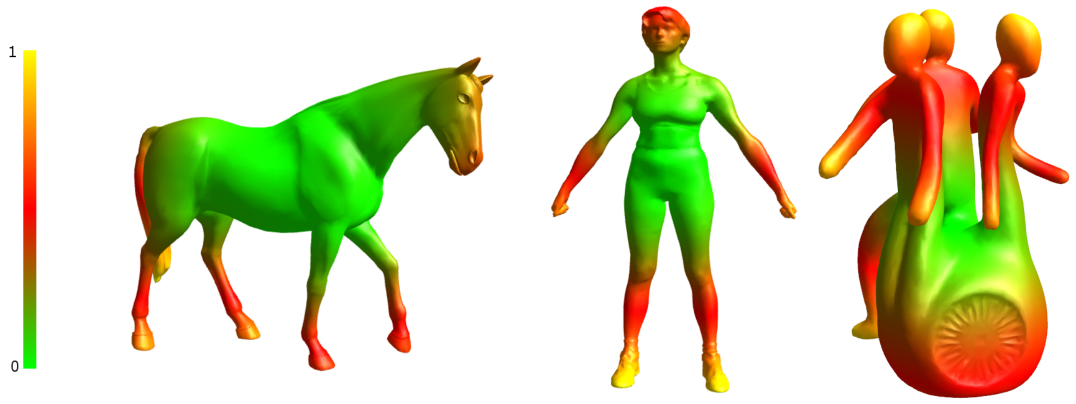
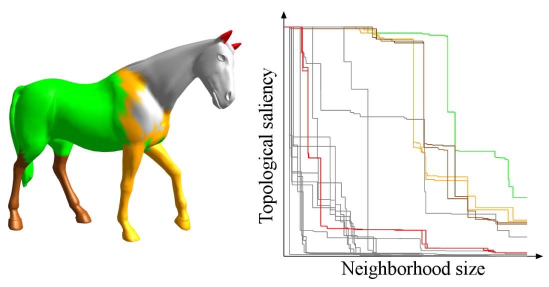


Figure 10.4: The `horse`, `human` and `memento` models used in the experiments with the shape descriptor function mapped to color.

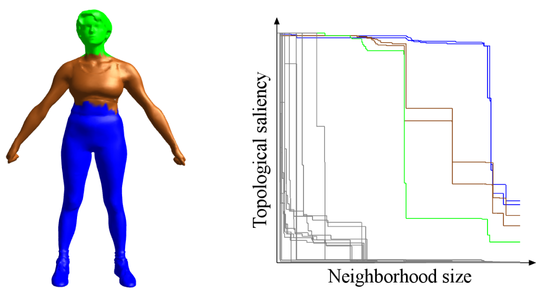
10.3.2 Extracting similar features

The saliency plot of a single feature can be considered as its descriptor and used to find similarity between features. The behavior of the plots of various features also aids in studying the relationship between features. Consider the plots corresponding to similar peaks B and C, colored orange and red respectively, in Figure 10.2. We observe that the two plots corresponding to them have a similar behavior. Another observation is that, though peaks A, D and E have persistence similar to peaks B and C, they differ in terms of the behavior of their corresponding plots. The neighborhood of B and C are similar, in the sense that they contain other peaks whose relative sizes are similar. In fact, the neighborhood is similar for different sizes of r. The same is not true for the peaks A, D and E, and hence their plots are different. Note that using persistence, it is not possible to distinguish between the features A, B, C, D, and E. In order to automatically capture this similarity between features, we define the distance between two features as the area between their corresponding plots. Two features are said to be similar if the area between the corresponding plots is close to zero. We now show that these observations indeed hold for many real world datasets. We group similar features of these datasets by analyzing the topological saliency plot.

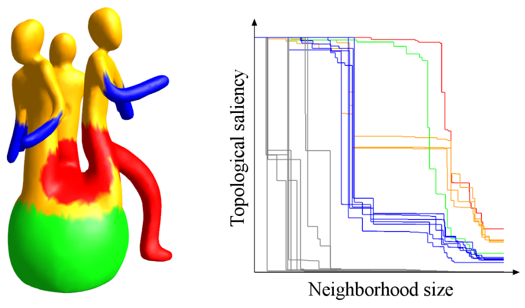
Figure 10.4 shows three surface meshes used as input in our experiments – `horse`, `human`, and `memento`. The average geodesic (AGD) function defined on the mesh is used as its shape



(a)



(b)



(c)

Figure 10.5: The topological saliency plots for the three surface meshes. Different features of the input surface like arms and legs are grouped together based on the similarity between their saliency plot.

descriptor. We compute and plot the topological saliency for varying r. Figure 10.5 shows the topological saliency plot for these models. Similar features in these models are highlighted in the figure. The similar plots and the corresponding regions of the mesh are represented by the same color. Segments in the model are computed using the branch decomposition of the join tree. Notice that for the `horse` model shown in Figure 10.5(a), the plot helps distinguish between its forelegs and hind legs. Ears of the horse are also grouped together. For the `human` model shown in Figure 10.5(b), the legs and hands form groups. An interesting point to note is that the plot corresponding to the head of the `human` does not cluster together with any of the other plots signifying that it is different from the other features. The torso and the hands of the three humanoid figures in the `memento` model form two groups, while the base of the model and the lone leg form separate groups.

As discussed earlier, the topological saliency plot can be used to distinguish between features that have similar persistence. Consider the plots corresponding to the forelegs and hind

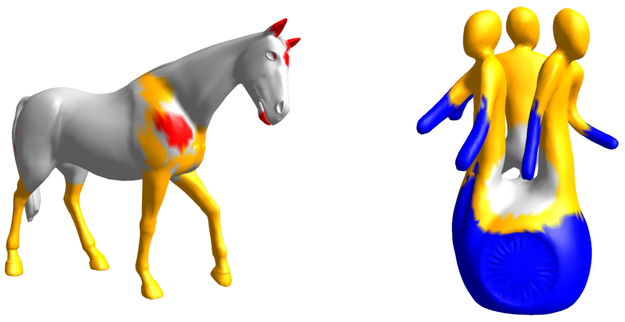


Figure 10.6: Using persistence to group features. Note that features having similar persistence need not correspond to similar features. Notice that the forelegs of the `horse` is grouped with its hind lings, while its ears are grouped with its jaw and a few small patches on its face and body. Similarly, the hands and the base of the `memento` model are grouped together.

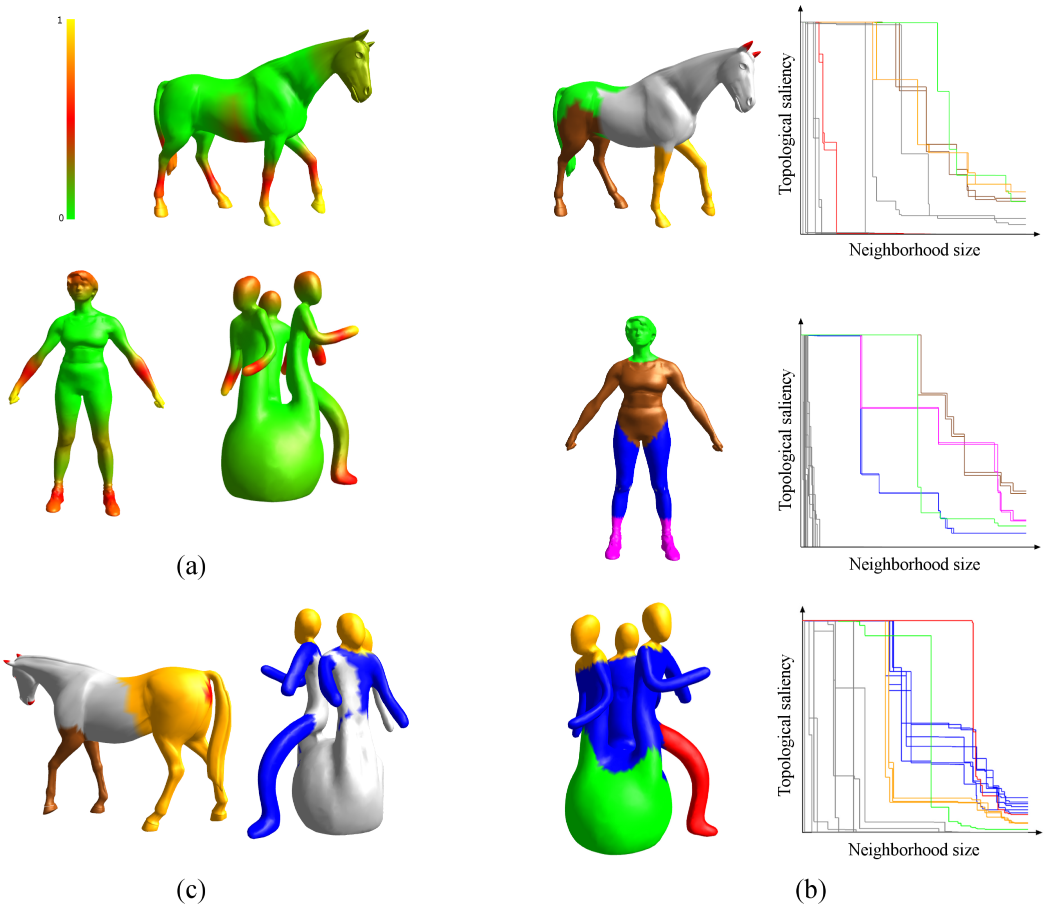


Figure 10.7: Experiments with the HKS functions defined on surface meshes. **(a)** The input scalar function. **(b)** Similar features are grouped together using the saliency plot. **(c)** Features grouped together using persistence. Note that persistence alone is not sufficient to distinguish between features.

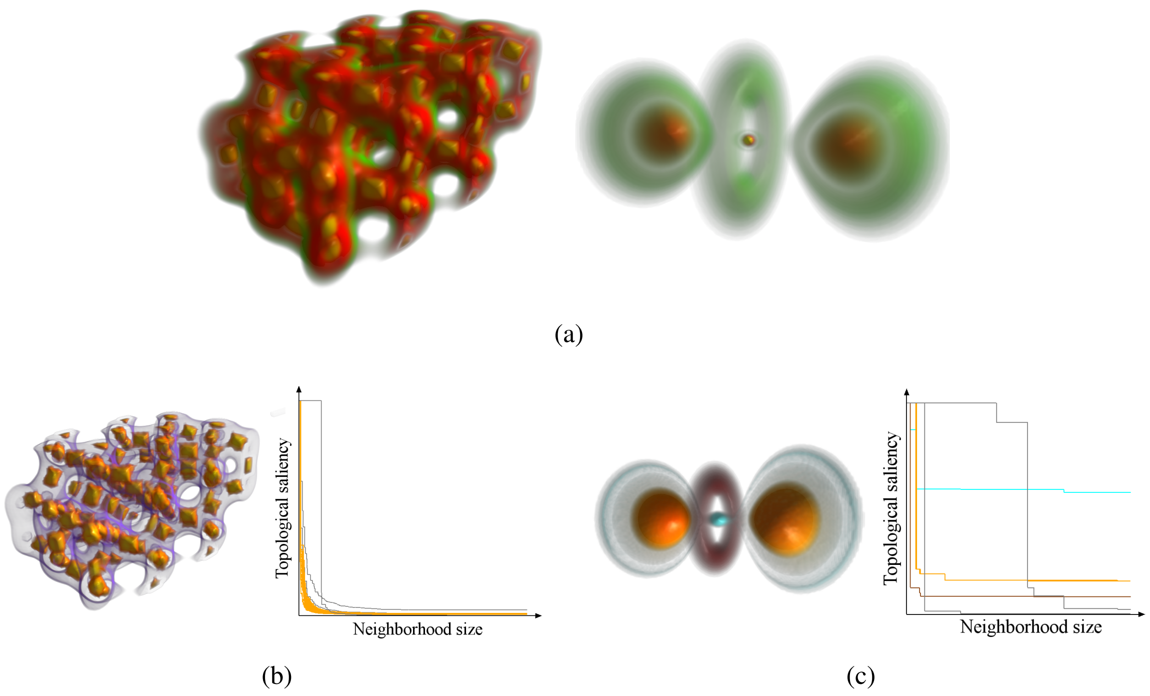


Figure 10.8: The topological saliency plots for the `silicium` and the `hydrogen atom` datasets. All the atoms of silicium, and the two spherical lobes of the hydrogen data are grouped together using the saliency plot.

legs of the `horse`. Notice that they have similar persistence; but the fact that their saliency plots differ helps us distinguish between them. Similarly, we are able to distinguish the hands in the `memento` model from its base by observing the behavior of the corresponding plots. Figure 10.6 highlights regions that are grouped together when using traditional persistence. The results indicate that persistence may not be sufficient to distinguish between features.

We repeat the above experiment using the heat kernel signature (HKS) function [64] as the scalar function instead of the AGD function. Figure 10.7 shows the results from this experiment. While we could again group similar features using the saliency plot, it was still not possible to distinguish between features using only persistence.

Figure 10.8(a) shows a volume rendering of the `silicium` and `hydrogen atom` datasets. The topological saliency plot and the features that are grouped in the `silicium` dataset is shown in Figure 10.8(b). Notice that all the curves corresponding to individual atoms have a similar behavior, and are thus clustered together. The plots corresponding to the two spherical lobes forms a group in the `hydrogen atom` dataset shown in Figure 10.8(c). Other dissimilar branches correspond to the toroidal region and the outer envelope.

Chapter 11

Conclusions

In this thesis, we explored the problem of efficiently computing Reeb graphs of scalar functions defined on manifolds and non-manifolds in any dimension, and its application to scientific data visualization. We accomplished this by first proposing three generic algorithms that compute the Reeb graph:

1. The sweep algorithm

   - Uses the conventional approach of sweeping the input and explicitly maintains connected components of level sets.
   - Has the best known theoretical bound on the running time.

2. The cylinder map algorithm

   - Uses an alternate definition of Reeb graphs using cylinder maps.
   - Simple to implement and efficient in practice.

3. The Recon algorithm

   - Computes the Reeb graph as a union of contour trees.
   - Simple to implement and efficient in practice.
   - Fastest algorithm among existing methods. It outperforms current generic algorithms by at least an order of magnitude.

- Handles input data that do not fit in memory.

Second, we outlined a method to simplify the Reeb graph based on an extended notion of persistence. We also described methods to compute an embedded layout and a feature-directed layout of the Reeb graph. These layouts serve as useful interfaces for exploring and understanding three-dimensional scalar fields.

Third, we discussed how Reeb graphs can be used to segment surfaces and design transfer functions for volume rendering. We also 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. Finally, we introduced topological saliency, which when used in conjunction with Reeb graphs, becomes a useful tool for various applications including key feature identification and feature clustering.

Experiments indicate that the algorithm, Recon, exhibits excellent practical performance. Therefore, we believe that the bound on the worst-case running time is loose. It will be interesting to either provide a tighter analysis of the algorithm or prove a lower bound that is different from that of the contour tree computation. Also, Recon is capable of handling large data, albeit with some limitations. With data sizes increasing rapidly, it becomes crucial to have robust techniques for computing Reeb graphs. Further, with the advent of affordable multi-core and many-core computing resources, it would be interesting to explore methods to compute Reeb graphs in parallel.

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. We expect Reeb graph based techniques used together with topological saliency will be useful in the analysis of higher dimensional data, where explicit visualization of the data becomes difficult.

Bibliography

[1] "Aim@shape shape repository." [Online]. Available: http://www.aimatshape.net/ 62

[2] "The Digital Michelangelo Project." [Online]. Available: http://graphics.stanford.edu/projects/mich/ 62, 66

[3] "Volvis repository." [Online]. Available: http://www.volvis.org/ 2, 73, 80

[4] P. K. Agarwal, H. Edelsbrunner, J. Harer, and Y. Wang, "Extreme elevation on a 2-manifold," *Discrete & Computational Geometry*, vol. 36, no. 4, pp. 553–572, 2006. 6, 14, 21, 86

[5] C. L. Bajaj, V. Pascucci, and D. R. Schikore, "The contour spectrum," in *Proc. IEEE Conf. Visualization*, 1997, pp. 167–173. 4

[6] T. F. Banchoff, "Critical points and curvature for embedded polyhedral surfaces," *American Mathematical Monthly*, vol. 77, pp. 475–485, 1970. 11, 13, 23

[7] S. Biasotti, L. De Floriani, B. Falcidieno, P. Frosini, D. Giorgi, C. Landi, L. Papaleo, and M. Spagnuolo, "Describing shapes by geometrical-topological properties of real functions," *ACM Computing Surveys*, vol. 40, pp. 12:1–12:87, October 2008. 20

[8] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno, "Reeb graphs for shape analysis and applications," *Theoretical Computer Science*, vol. 392, pp. 5–22, February 2008. 20

[9] S. Biasotti, D. Attali, J.-D. Boissonnat, H. Edelsbrunner, G. Elber, M. Mortara, G. S. Baja, M. Spagnuolo, M. Tanase, and R. Veltkamp, "Skeletal structures," in *Shape Analysis and Structuring*, ser. Mathematics and Visualization, L. Floriani, M. Spagnuolo,

G. Farin, H.-C. Hege, D. Hoffman, C. R. Johnson, and K. Polthier, Eds. Springer Berlin Heidelberg, 2008, pp. 145–183. 20

[10] S. Biasotti, L. Floriani, B. Falcidieno, and L. Papaleo, "Morphological representations of scalar fields," in *Shape Analysis and Structuring*, ser. Mathematics and Visualization, L. Floriani, M. Spagnuolo, G. Farin, H.-C. Hege, D. Hoffman, C. R. Johnson, and K. Polthier, Eds. Springer Berlin Heidelberg, 2008, pp. 185–213. 20

[11] D. A. Boas, D. H. Brooks, E. L. Miller, C. A. DiMarzio, M. Kilmer, R. J. Gaudette, and Q. Zhang, "Imaging the body with diffuse optical tomography," *IEEE Signal Processing Magazine*, vol. 18, no. 6, pp. 57–75, 2001. 91

[12] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Computational Geometry: Theory and Applications*, vol. 24, no. 2, pp. 75–94, 2003. 13, 17, 18, 47

[13] H. Carr and J. Snoeyink, "Path seeds and flexible isosurfaces using topology for exploratory visualization," in *Proc. Symposium on Data visualisation 2003*, ser. VISSYM '03. Eurographics Association, 2003, pp. 49–58. 20

[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. 4, 6, 20, 67, 68, 86

[15] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Computational Geometry: Theory and Applications*, vol. 30, no. 2, pp. 165–195, 2005. 17, 54

[16] Y.-J. Chiang and X. Lu, "Progressive simplification of tetrahedral meshes preserving all isosurface topologies," *Computer Graphics Forum*, vol. 22, no. 3, pp. 493–504, 2003. 4

[17] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Loops in Reeb graphs of 2-manifolds," *Discrete & Computational Geometry*, vol. 32, no. 2, pp. 231–244, 2004. 9, 18, 19

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 2001. 17, 27

[19] R. Diestel, *Graph Theory*, 3rd ed., ser. Graduate Texts in Mathematics. Springer-Verlag, Heidelberg, 2005, vol. 173. 47

[20] H. Doraiswamy and V. Natarajan, "Efficient algorithms for computing Reeb graphs," *Computational Geometry: Theory and Applications*, vol. 42, no. 6-7, pp. 606–616, 2009. 5

[21] ——, "Output-sensitive construction of Reeb graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 146–159, 2012. 5

[22] ——, "Computing Reeb graphs as a union of contour trees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 2, pp. 249–262, 2013. 5

[23] H. Doraiswamy, N. Shivashankar, V. Natarajan, and Y. Wang, "Topological saliency," *Computers & Graphics*, vol. to appear, 2013. 6

[24] H. Doraiswamy, A. Sood, and V. Natarajan, "Constructing reeb graphs using cylinder maps," in *Proc. Symposium on Computational geometry*, ser. SoCG '10, 2010, pp. 111–112. 38

[25] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction*. Amer. Math. Soc., Providence, Rhode Island, 2009. 8

[26] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Morse-Smale complexes for piecewise linear 3-manifolds," in *Proc. Symposium on Computational geometry*, 2003, pp. 361–370. 11, 13, 23

[27] H. Edelsbrunner, *Geometry and Topology for Mesh Generation*. England: Cambridge Univ. Press, 2001. 13

[28] H. Edelsbrunner, J. Harer, and A. Zomorodian., "Hierarchical Morse-Smale complexes for piecewise linear 2-manifolds," *Discrete & Computational Geometry*, vol. 30, no. 1, pp. 87–107, 2003. 85

[29] H. Edelsbrunner, D. Letscher, and A. Zomorodian., "Topological persistence and simplification," *Discrete & Computational Geometry*, vol. 28, no. 4, pp. 511–533, 2002. 14, 67

[30] D. Eppstein, "Dynamic generators of topologically embedded graphs," in *Proc. Symposium on Discrete Algorithms*, 2003, pp. 599–608. 26, 32

[31] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a minimum spanning forest in a dynamic plane graph," *Journal of Algorithms*, vol. 13, no. 1, pp. 33–54, 1992. 27

[32] I. Fujishiro, Y. Maeda, and H. Sato, "Interval volume: a solid fitting technique for volumetric data display and analysis," in *Proc. IEEE Conf. Visualization*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 151–158. 79

[33] 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. 4, 82

[34] A. Gibson, J. C. Hebden, and S. R. Arridge, "Recent advances in diffuse optical tomography," *Physics in Medicine and Biology*, vol. 50, pp. R1–R43, 2005. 91

[35] B. Guo, "Interval set: A volume rendering technique generalizing isosurface extraction," in *Proc. IEEE Conf. Visualization*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 3–10. 79

[36] I. Guskov and Z. Wood, "Topological noise removal," in *Proc. Graphics Interface*, 2001, pp. 19–26. 4

[37] W. Harvey and Y. Wang, "Topological landscape ensembles for visualization of scalar-valued functions," *Computer Graphics Forum*, vol. 29, pp. 993–1002, 2010. 4, 22

[38] W. Harvey, Y. Wang, and R. Wenger, "A randomized O(m log m) time algorithm for computing Reeb graphs of arbitrary simplicial complexes," in *Proc. Symposium on Computational geometry*, 2010, pp. 267–276. 18, 19, 60, 62, 86

[39] A. Hatcher, *Algebraic Topology*. New York: Cambridge U. Press, 2002. 8

[40] C. Heine, D. Schneider, H. Carr, and G. Scheuermann, "Drawing contour trees in the plane," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1599–1611, Nov. 2011. 22

[41] F. Hétroy and D. Attali, "Topological quadrangulations of closed triangulated surfaces using the Reeb graph," *Graphical Models*, vol. 65, no. 1-3, pp. 131–148, 2003. 4

[42] 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. 4, 20, 77

[43] L. Itti, C. Koch, and E. Niebur, "A model of saliency-based visual attention for rapid scene analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 1254–1259, November 1998. 86

[44] K. d. L. J. Holm and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *Journal of the ACM*, vol. 48, no. 4, pp. 723–760, 2001. 32, 33

[45] C. Koch and S. Ullman, "Shifts in selective visual attention: towards the underlying neural circuitry," *Human Neurobiology*, vol. 4, pp. 219–227, 1985. 86

[46] F. Lazarus and A. Verroust, "Level set diagrams of polyhedral objects," in *Proc. Symposium on Solid modeling and applications*, 1999, pp. 130–140. 4

[47] C. H. Lee, A. Varshney, and D. W. Jacobs, "Mesh saliency," *ACM Transactions on Graphics*, vol. 24, pp. 659–666, July 2005. 86

[48] Y. Matsumoto, *An Introduction to Morse Theory*. Amer. Math. Soc., 2002, translated from Japanese by K. Hudson and M. Saito. 8, 10

[49] R. Milanese, H. Wechsler, S. Gil, J. Bost, and T. Pun, “Integration of bottom-up and top-down cues for visual attention using non-linear relaxation,” in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 1994, pp. 781–785. 86

[50] J. Milnor, *Morse Theory*. New Jersey: Princeton Univ. Press, 1963. 8

[51] M. Mortara and G. Patané, “Affine-invariant skeleton of 3d shapes,” in *SMI ’02: Proceedings of the Shape Modeling International 2002 (SMI’02)*, 2002, pp. 245–252. 4

[52] E. P. Mücke, “Shapes and implementations in three-dimensional geometry,” Ph.D. dissertation, Dept. Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1993. 15

[53] P. Oesterling, C. Heine, H. Jänicke, G. Scheuermann, and G. Heyer, “Visualization of high-dimensional point clouds using their density distribution’s topology,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 11, pp. 1547–1559, Nov. 2011. 4

[54] 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. 20, 21, 70, 71

[55] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas, “Robust on-line computation of Reeb graphs: simplicity and speed,” *ACM Transactions on Graphics*, vol. 26, no. 3, 2007. 18, 19, 21, 62, 66

[56] 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. 18, 19

[57] 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. 10

[58] J. Reininghaus, N. Kotava, D. Guenther, J. Kasten, H. Hagen, and I. Hotz, “A scale space based persistence measure for critical points in 2d scalar fields,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 2045–2052, Dec. 2011. 86

[59] R. Rosenholtz, “A simple saliency model predicts a number of motion popout phenomena,” *Vision Research*, vol. 39, pp. 3157–3163, 1999. 86

[60] Y. Shinagawa and T. L. Kunii, “Constructing a Reeb graph automatically from cross sections,” *IEEE Computer Graphics and Applications*, vol. 11, no. 6, pp. 44–51, 1991. 19

[61] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien, “Surface coding based on Morse theory,” *IEEE Computer Graphics and Applications*, vol. 11, no. 5, pp. 66–78, 1991. 4

[62] D. D. Sleator and R. E. Tarjan, “A data structure for dynamic trees,” *Journal of Computer and System Sciences*, vol. 26, no. 3, pp. 362–391, 1983. 27

[63] B.-S. Sohn, “Topology preserving tetrahedral decomposition of trilinear cell,” in *Proc. Intl. Conf. on Computational Science, Part I*, 2007, pp. 350–357. 79, 80

[64] J. Sun, M. Ovsjanikov, and L. Guibas, “A concise and provably informative multi-scale signature based on heat diffusion,” in *Proc. Symposium on Geometry Processing*, ser. SGP ’09. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2009, pp. 1383–1392. 96

[65] 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. 4, 20, 82

[66] S. Takahashi, G. M. Nielson, Y. Takeshima, and I. Fujishiro, “Topological volume skeletonization using adaptive tetrahedralization,” in *Proc. Geometric Modeling and Processing*, 2004, pp. 227–236. 4, 20

[67] S. Takahashi, Y. Shinagawa, and T. L. Kunii, “A feature-based approach for smooth surfaces,” in *Proc. Symposium on Solid modeling and applications*, 1997, pp. 97–110. 4

[68] Y. Takeshima, S. Takahashi, I. Fujishiro, and G. Nielson, “Introducing topological attributes for objective-based visualization of simulated datasets,” *International Workshop on Volume Graphics*, vol. 0, pp. 137–236, 2005. 82

[69] M. Thorup, “Near-optimal fully-dynamic graph connectivity,” in *Proc. ACM Symposium on Theory of Computing*, 2000, pp. 343–350. 32, 34

[70] 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. 18, 19, 62

[71] J. K. Tsotsos, S. M. Culhane, W. Y. K. Wai, Y. Lai, N. Davis, and F. Nuflo, “Modeling visual attention via selective tuning,” *Artificial Intelligence*, vol. 78, pp. 507–545, October 1995. 86

[72] T. Tung and F. Schmitt, “Augmented Reeb graphs for content-based retrieval of 3d mesh models,” in *Proc. Shape Modeling Intl.*, 2004, pp. 157–166. 20

[73] 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. Symposium on Computational geometry*, 1997, pp. 212–220. 4

[74] W. Wang, C. Bruyere, and B. Kuo, “Competition data set and description in 2004 IEEE Visualization design contest,” 2004. [Online]. Available: http://vis.computer.org/vis2004contest/data.html 84

[75] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann, “Topology-controlled volume rendering,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 330–341, 2007. 4, 82, 86

[76] G. Weber, P.-T. Bremer, and V. Pascucci, “Topological landscapes: A terrain metaphor for scientific data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 1416–1423, November 2007. 21, 86

[77] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder, “Removing excess topology from isosurfaces,” *ACM Transactions on Graphics*, vol. 23, no. 2, pp. 190–208, 2004. 4

[78] 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. 4

[79] I. Yamazaki, V. Natarajan, Z. Bai, and B. Hamann, “Segmenting point sets,” in *Shape Modeling and Applications, 2006. SMI 2006. IEEE International Conference on*, 2006, pp. 6–6. 78

[80] E. Zhang, K. Mischaikow, and G. Turk, “Feature-based surface parameterization and texture mapping,” *ACM Transactions on Graphics*, vol. 24, no. 1, pp. 1–27, 2005. 4, 78

[81] 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. 4, 82

[82] Y. Zhou and Z. Huang, “Decomposing polygon meshes by means of critical points,” in *Proc. Intl. Multimedia Modelling Conf.*, ser. MMM ’04. IEEE Computer Society, 2004, pp. 187–195. 77

[83] A. Zomorodian and G. Carlsson, “Localized homology,” *Computational Geometry: Theory and Applications*, vol. 41, pp. 126–148, November 2008. 86