# Time-varying Reeb Graphs: A Topological Framework Supporting the Analysis of Continuous Time-varying Data

by
Ajith Arthur Mascarenhas

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2006

## ABSTRACT

**AJITH ARTHUR MASCARENHAS: Time-varying Reeb Graphs: A Topological Framework Supporting the Analysis of Continuous Time-varying Data.**
**(Under the direction of Jack Snoeyink.)**

I present time-varying Reeb graphs as a topological framework to support the analysis of continuous time-varying data. Such data is captured in many studies, including computational fluid dynamics, oceanography, medical imaging, and climate modeling, by measuring physical processes over time, or by modeling and simulating them on a computer.

Analysis tools are applied to these data sets by scientists and engineers who seek to understand the underlying physical processes. A popular tool for analyzing scientific datasets is *level sets*, which are the points in space with a fixed data value $s$. Displaying level sets allows the user to study their geometry, their topological features such as connected components, handles, and voids, and to study the evolution of these features for varying $s$.

For static data, the *Reeb graph* encodes the evolution of topological features and compactly represents topological information of all level sets. The Reeb graph essentially contracts each level set component to a point. It can be computed efficiently, and it has several uses: as a succinct summary of the data, as an interface to select meaningful level sets, as a data structure to accelerate level set extraction, and as a guide to remove noise.

I extend these uses of Reeb graphs to time-varying data. I characterize the changes to Reeb graphs over time, and develop an algorithm that can maintain a Reeb graph data structure by tracking these changes over time. I store this sequence of Reeb graphs compactly, and call it a *time-varying Reeb graph.* I augment the time-varying Reeb graph with information that records the topology of level sets of all level values at all times, that maintains the correspondence of level set components over time, and that accelerates the extraction of level sets for a chosen level value and time.

Scientific data sampled in space-time must be extended everywhere in this domain using an interpolant. A poor choice of interpolant can create degeneracies that are difficult to resolve, making construction of time-varying Reeb graphs impractical. I investigate piecewise-linear, piecewise-trilinear, and piecewise-prismatic interpolants, and conclude that piecewise-prismatic is the best choice for computing time-varying Reeb graphs.

Large Reeb graphs must be simplified for an effective presentation in a visualization system. I extend an algorithm for simplifying static Reeb graphs to compute simplifications of time-varying Reeb graphs as a first step towards building a visualization system to support the analysis of time-varying data.

# ACKNOWLEDGMENTS

I have been fortunate to receive the unstinting support of many people during the course of my graduate studies. I would like to express my gratitude to them.

I've had the pleasure of working with my advisor Prof. Jack Snoeyink for many years at UNC. During this period I have learned a lot from Jack. His superb writing skills and immediate constructive feedback have improved and refined my own writing skills and thereby clarified my thinking. I believe that these skills are crucial for success in a research career and I'm forever indebted to Jack for imparting them to me. I will definitely miss playing foosball and ultimate frisbee with Jack.

I've been fortunate to interact and learn from Prof. Herbert Edelsbrunner. The meticulous care and mathematical rigor with which Herbert develops and describes concepts has been an inspiration to me. I'm grateful to Herbert for the many insightful discussions and for the support over the years.

Interaction and feedback from researchers working in the field of visualization is invaluable in applying my research. In this regard I've been lucky to work with such a wonderful person as Dr. Valerio Pascucci at Lawrence Livermore National Laboratory (LLNL). Valerio has provided constant encouragement during my graduate studies and continues to do so now in my research career at LLNL. I could not ask for a more invigorating and fun mentor than Valerio.

I would like to thank Prof. Dinesh Manocha and Prof. Leonard McMillan for their support, time, and constructive feedback during the course of my dissertation work.

I'm grateful to Prof. John Harer for the many enlightening discussions on topology when we worked together on developing time-varying Reeb graphs. I thank Prof. Ming Lin for her support and guidance during my first year at UNC. A big thank you to the wonderful, caring, friendly faculty and staff in the CS dept at UNC for creating such a great work environment in Sitterson hall. I'm indebted to Giorgio Scorzelli who put in a superlative effort, going well beyond the call of duty, in developing the visualization software incorporating time-varying Reeb graphs. I would like to acknowledge the funding agencies that have supported my research: NSF-ITR, LLNL.

During my years at UNC I've enjoyed the friendship of many outstanding people. I thank Andrew Nashel, Caroline Green, Sharif Razzaque, Yuanxin Liu, Greg Coombe, Kavita Dave-Coombe, David Marshburn, and Kelly Ward for the many years of friendship, good humor, interesting discussions, countless coffee breaks, and the many Iron Chef showdowns. My heartfelt thanks go out to my dear friends Avneesh Sud, Adrian Ilie, Udita Patel, John Chek, Heather Hanna, Bala Krishnamoorthy, Anand Srinivasan, Chandna Bhatnagar, Gopi Meenakshisundaram, Aditi Majumder, Anupama Krishnan, Iftekhar Kalsekar, and Vijay Natarajan for all the support, potlucks, airport rides in the wee hours, and being there when it mattered. My childhood friends deserve a special thank you for all the many years of support: Sushanth Tharappan, Srinivas Premdas, and Prashanth Sarpamale. I'm lucky to have a family away from home in Arghya, Shalini, Ira, Sujeet, Anand, and Sangita.

Finally, my deepest gratitude goes to my family for their unconditional love.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS

| | |
|---|---|
| $\mathbb{M}$ | manifold |
| $\mathbb{R}^3$ | Euclidean space |
| $\mathbb{S}^3$ | 3-sphere |
| $f, g : \mathbb{M} \to \mathbb{R}$ | Morse functions |
| $g$ | genus of a 2-manifold |
| $\chi$ | Euler characteristic of a 2-manifold |
| $\gamma, \gamma'$ | segments of Jacobi curve |
| $K$ | triangulation |
| $u, v, \sigma, \tau$ | vertices, simplices |
| $\mathrm{Lk}\,\sigma, \mathrm{Lk}_{-}\sigma$ | link, lower link |
| $\beta_k, \tilde{\beta}_k$ | Betti number, reduced Betti number |
| $s, t$ | level, time parameters |
| $R_t$ | Reeb graph at time $t$ |
| $L_s$ | level graph at level value $s$ |
| $x, y$ | critical points |
| $x, a$ | node, arc in Reeb graph |

# Chapter 1

# Introduction

With increasing computer capabilities to capture or generate data, we need to develop and implement new mathematical tools to visualize and analyze data. If a picture can be worth a thousand words, then an abstraction can be worth a thousand pictures. I develop such an abstraction, time-varying Reeb graphs, as a topological framework to support the analysis of continuous time-varying data. Here, I describe the nature of continuous time-varying data, motivate why such data is analyzed, outline key problems in analyzing the data, and describe how time-varying Reeb graphs can assist in their solution.

## 1.1 Continuous Space-time Data

Physical processes that are measured over time, or that are modeled and simulated on a computer, can produce large amounts of time-varying data that can be interpreted with the assistance of computational tools. Such data arises in a wide variety of scientific studies including computational fluid dynamics [CM97], oceanography [BSRF00], medical imaging [TO91], and climate modeling [MCW00].

**Data representation and interpolants.** Many scientific datasets consist of measured or computed values at a finite set of sample points in a space-time domain. Often the values are scalar, with perhaps several scalar values for each sample point. Examples include pressure, temperature, density. Values can also be vectors, usually in a study of motion, forces, or velocity of some kind. In this dissertation, I focus on scalar-valued data, often called *scalar fields.*

An interpolation scheme extends these values to an *interpolant*: a continuous function over the whole domain that agrees with the values at the sample points. Mesh-based interpolation schemes connect the sample points into a mesh, and define the interpolant over each mesh element.

## 1.2 Analysis and Visualization of Scientific Data

To understand the physical process that a scientific dataset captures, it is analyzed by exploring it for important features. The features depend on the field of study; a medical researcher might be interested in tumors, while a climatologist might be interested in regions of high pressure. Because humans possess a highly developed visual system, transforming the data into images and movies that can be displayed, and providing the scientist with tools to control them, can be a powerful *visualization* technique [MDB87].

Popular techniques employed to create such visualizations for static data are *direct volume rendering* [KH84, Lev90, Wes89], where we render the entire data volume with transparent colors chosen to highlight features, and *level sets* [LC87, WG92, LSJ96], where we compute the points in space with a fixed scalar value $s$ and display the results [LC87]. A level set is not necessarily connected. Topological features of the level sets, such as connected components, handles, and voids, and the interaction between these features as we vary $s$ can aid in interpreting the data. In Chapter 3, I discuss related research in level-set-based visualization.

In this dissertation, I develop concepts, algorithms, and data-structures to aid in the analysis of time-varying data using level sets. In particular, I will extend the Reeb graph, which is used to analyze static data, to analyze time-varying data.

**The Reeb graph.** The Reeb graph encodes the evolution of topological features and compactly represents topological information of all level sets. The Reeb graph essentially contracts each level set component to a point. It can be computed efficiently [CSA03], and it can be used in the analysis of static data as a succinct summary of the data, as an interface to select meaningful level sets [BPS97], to extract level sets fast [CS03], and to remove noise [CSvdP04].

The Reeb graph provides information that is not obvious from looking at level sets by themselves. Figure 1.1 and Figure 1.2 illustrate two cases.

Figure 1.1 shows a level set of the electron density distribution of a silicium grid simulation with its Reeb graph. Visual inspection of the level set does not indicate if the level set has only one or several components, but the Reeb graph tells us that the level set has only one connected component because the level value (arrow) is contained on only one arc of the Reeb graph. The Reeb graph also tells us that this level set

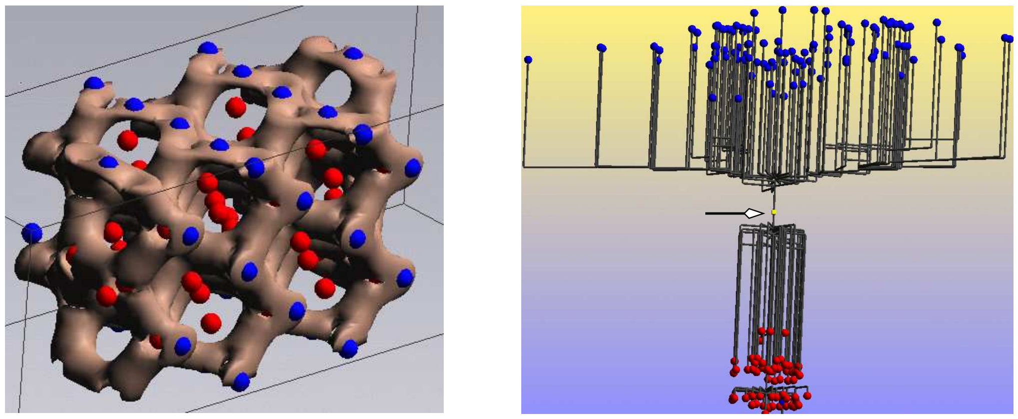


Figure 1.1: At left, a level set of the electron density distribution of a silicium grid simulation, and at right its Reeb graph. Because the level value (arrow) is contained on only one arc of the Reeb graph we know that the level set is one connected component. Red spheres are minima, blue spheres are maxima. [PCMS04]

component will split into several components at a level value that is smaller or larger than the value of the end points of the arc on which it lies.

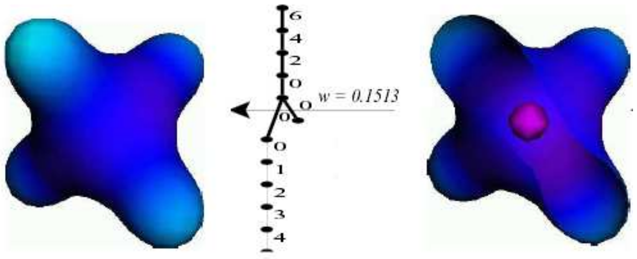


Figure 1.2: At left, a level set of the electron density distribution of a methane molecule. In the middle a portion of its Reeb graph. The Reeb graph indicates that there are two level set components, but there is only one level set component visible. The second component is inside the first component, as shown in the sliced surface on the right. [PCM03]

Figure 1.2 shows a level set (left) of the electron density distribution of a methane molecule with a portion of its Reeb graph (middle). The Reeb graph tells us that there are two level set components because two of its arcs contain the level value, but there is only one visible component. Cutting this level set component (right) and looking inside reveals the hidden second component.

The Reeb graph provides a useful framework to analyze static data. I extend its use to the analysis of time-varying data.

Historically algorithms for the analysis and visualization of static data preceded those for time-varying data. A natural step towards the analysis and visualization of time-varying data is to compute time-slices to study temporal behavior, and use existing algorithms to compute level sets within each time-slice to study variation over space.

To perform level-set-based analysis of time-varying data, we must answer several questions: How do different components of a level set interact as the time and level value are continuously varied? At what time, and level values do components appear, disappear, merge, split or change genus? How can we track the evolution of the topology of a fixed level set over time? What tools can provide this topological information without actually computing each possible level set? How can we extract level sets for a chosen level value and time quickly for interactive display in a visualization?

I present time-varying Reeb graphs as a framework to answer some of these questions.

## 1.3 Time-varying Reeb Graphs as a Framework for Analyzing Continuous Space-time Data

I define the *time-varying Reeb graph* as the family of Reeb graphs parameterized by time $t$, stored compactly in a partially persistent data-structure supporting access to Reeb graph $R_t$ at any time $t$.

My thesis is:

*Time-varying Reeb graphs provide a topological framework supporting the analysis of continuous time-varying data by recording the topology of level sets of all level values at all times, by maintaining the correspondence between level set components over time, and by accelerating the extraction of level sets for a chosen level value and time.*

In support of this thesis, I first define time-varying Reeb graphs rigorously in the language of Morse theory and develop an algorithm to compute them in Chapter 4. Morse theory provides the concepts, theorems, and structure to define Reeb graphs and to characterize how they change over time. In Chapter 2, I collect definitions for the required mathematical background.

I develop algorithms in Chapter 5 to augment the time-varying Reeb graph with

- Betti numbers, which describe the topology of level sets of all level values at all times, and

- Path seeds, which can accelerate the extraction of the individual components of level sets of all level values at all times.

Also in Chapter 5, I show how the time-varying Reeb graph maintains the correspondence between level set components over time. I encode this correspondence as a *level graph* which captures the topological evolution of the level set of a fixed level value over time, and show how to compute the level graph from the time-varying Reeb graph.

To develop a practical implementation of these algorithms I address some technical obstacles. Because I define and study time-varying Reeb graphs using Morse theory, and because Morse theory demands generic smooth functions, I take special care to ensure that my algorithms work on the interpolated, continuous but not always smooth, real-world scientific data.

I use *Jacobi sets*, which are the paths traced by the critical points over time, to connect the family of Reeb graphs and compute time-varying Reeb graphs. In the smooth world, the Jacobi set of Morse functions is a collection of smooth, and separated curves. In the continuous world, the Jacobi set of continuous functions is a collection of continuous curves that may intersect. Because scientific datasets are extended to all points in space-time by an interpolant, and because different interpolants produce Jacobi sets which approximate the structure of smooth Jacobi sets to different degrees, I evaluate which interpolant produces Jacobi sets with the best approximation to the smooth case. The choice of interpolant often depends on the characteristics of the sampling of the data. In Chapter 6, I compute and compare the Jacobi sets produced by three different interpolants that can be defined on three classes of samplings.

My experiences in implementing the time-varying Reeb graphs convinced me that the devil was indeed in the details. Because I believe that key details are important to readers who wish to implement the algorithms developed in this dissertation, I collect them in Chapter 7.

Large Reeb graphs must be simplified for a clear and uncluttered visualization. To use the Reeb graphs as an interface to select level sets it is also important that the user be able control the amount of simplification, and that the presentation be coherent between simplification, and over time. In Chapter 8, I present some preliminary results on presenting time-varying Reeb graphs over time with simplification and coherence between simplifications.

Finally, I summarize, discuss future work, and pose open problems in Chapter 9.

### 1.3.1 Results

The main results developed in this dissertation are

- A rigorous Morse theoretic definition of time-varying Reeb graphs, and an enumeration of the type of combinatorial changes experienced by the Reeb graph over time. [Chapter 4]
- An algorithm to compute time-varying Reeb graphs. [Chapter 4]
- An algorithm to record the Betti numbers of level set components of all level values at all times. [Chapter 5]
- An algorithm to maintain path seeds for accelerated extraction of level sets for a chosen level and time. [Chapter 5]
- A definition of the level graph, which captures the evolution of the level set components of a fixed level value over time, and an algorithm to compute the level graph. [Chapter 5]
- An evaluation of piecewise-linear, piecewise-trilinear, and piecewise-prismatic interpolants for computing time-varying Reeb graphs. I present algorithms to compute Jacobi sets of the latter two interpolants, compare the properties of the Jacobi sets produced by all three interpolants acting on regularly sampled data, and conclude that piecewise-prismatic is the best choice for computing time-varying Reeb graphs. [Chapter 6]
- An algorithm to maintain a hierarchical branch decomposition of the Reeb graph over time. The hierarchical branch decomposition supports simplification, and a coherent presentation of the Reeb graph in a visualization system. [Chapter 8]

# Chapter 2

# Mathematical Background

The mathematical background for this thesis comes from Morse theory [Mat02, Mil63], and from combinatorial and algebraic topology [Ale98, Mun84]. I attempt to understand functions and their properties in a smooth setting and translate the concepts developed to the discrete setting required for computation. The structure of this chapter reflects this approach. I begin with ideas from general topology in Section 2.1, followed by definitions of smooth maps on manifolds and their properties in Section 2.2. I define simplicial complexes, which are the discrete analogue of manifolds, and piecewise linear (PL) functions that are defined on them in Section 2.4. I define concepts from algebraic topology that enable us to develop combinatorial algorithms to detect critical points of a PL function in Section 2.4. Such algorithms are preferred over numerical algorithms because they are amenable to a robust implementation.

I do not attempt to collect all necessary mathematics into this chapter; when the supporting mathematics is better explained together with an algorithm or concept I place them together later in the thesis.

## 2.1 General Topology

General topology formalizes several fundamental concepts.

**Definition 2.1.1 (Topology)** A *topology* on a set $X$ is a collection $\mathcal{T}$ of subsets of $X$ having the following properties:

1. The sets $\emptyset$ and $X$ are in $\mathcal{T}$;
2. The union of any subcollection of $\mathcal{T}$ is in $\mathcal{T}$;

3. The intersection of a finite subcollection of $\mathcal{T}$ is in $\mathcal{T}$.

**Definition 2.1.2 (Topological Space)** A set $X$ for which a topology $\mathcal{T}$ is defined is called a *topological space*. Reference to the topology $\mathcal{T}$ is often omitted when it is clear from context.

**Definition 2.1.3 (Open Set)** A subset $U \subset X$, of the topological space $X$ is an *open* set if it belongs to $\mathcal{T}$.

**Definition 2.1.4 (Closed Set)** A subset $A \subset X$, of the topological space $X$ is a *closed* set if its compliment $X - A$ is open.

**Definition 2.1.5 (Function)** A *function* $f\colon X \to Y$ associates each element of space $X$ with a unique element of space $Y$.

A function is also called a *map*.

**Definition 2.1.6 (Continuous Function)** Let $X, Y$ be topological spaces. A function $f\colon X \to Y$ is *continuous* if for each open subset $U \subset Y$, the set $f^{-1}(U)$ is an open subset of $X$.

**Definition 2.1.7 (Homeomorphic Spaces)** Two spaces $X, Y$ are homeomorphic if and only if there exists a continuous injection $f\colon X \to Y$ with a continuous inverse $f^{-1} : Y \to X$.

**Definition 2.1.8 (Covering)** A collection $\mathcal{A}$ of subsets of a space $X$ is a *covering* of $X$, if the union of elements of $\mathcal{A}$ is equal to $X$.

**Definition 2.1.9 (Compact Space)** A space $X$ is *compact* if every open covering $\mathcal{A}$ of $X$ contains a finite subcollection that also covers $X$.

## 2.2 Smooth Maps on Manifolds

We need concepts from the theory of smooth manifolds to understand time-varying functions.

**Definition 2.2.1 (Smooth Map)** A map $f\colon X \to Y$ is called a *smooth map* if it has continuous partial derivatives of all orders.

**Definition 2.2.2 (Manifold)** A space $\mathbb{M}$ is a $d$-manifold if every point $x \in \mathbb{M}$ has an open neighborhood homeomorphic to $\mathbb{R}^d$.

Let $\mathbb{M}$ be a smooth, compact $d$-manifold without boundary and $f\colon \mathbb{M} \to \mathbb{R}$ a smooth map.

**Definition 2.2.3 (Critical Point and Critical Value)** Assuming a local coordinate system in the neighborhood of $p \in \mathbb{M}$, the point $p$ is a *critical point* of $f$ if all partial derivatives vanish at $p$. If $p$ is a critical point, $f(p)$ is a *critical value.*

**Definition 2.2.4 (Regular Point and Regular Value)** Non-critical points and non-critical values are called *regular points* and *regular values*, respectively.

**Definition 2.2.5 (Hessian)** The *Hessian* at $x$ is the matrix of second-order partial derivatives, expressed here in terms of local coordinates

$$H_f(x) \quad = \quad \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \cdots & \frac{\partial^2 f}{\partial x_d \partial x_1}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_d}(x) & \cdots & \frac{\partial^2 f}{\partial x_d^2}(x) \end{bmatrix},$$

**Definition 2.2.6 (Non-degenerate Critical Point)** A critical point $p$ is *non-degenerate* if the Hessian at $p$ is non-singular.

**Definition 2.2.7 (Index of a Critical Point)** The *index* of a critical point $p$ is the number of negative eigenvalues of the Hessian.

Intuitively, the index is the number of mutually orthogonal directions at $p$ along which $f$ decreases. For $d = 3$ there are four types of non-degenerate critical points: the *minima* with index 0, the 1-*saddles* with index 1, the 2-*saddles* with index 2, and the *maxima* with index 3.

**Definition 2.2.8 (Morse Function)** A function $f$ is *Morse* if

I. all critical points are non-degenerate;

II. $f(x) \neq f(y)$ whenever $x \neq y$ are critical.

I will refer to I and II as Genericity Conditions as they prevent certain non-generic configurations of the critical points. This choice of name is justified because Morse functions are dense in $C^\infty(\mathbb{M})$, the class of smooth functions on the manifold [GG73, Mat02].

In other words, for every smooth function there is an arbitrarily small perturbation that makes it a Morse function.

**Definition 2.2.9 (Morse Lemma)** If $p$ is a non-degenerate critical point then there exist local coordinates such that

$$f(x_1, x_2, \cdots, x_d) = f(p) \pm {x_1}^2 \pm {x_2}^2 \pm \cdots \pm {x_d}^2$$

in a local neighborhood of $p$.

The Morse Lemma indicates that $f$ is quadratic in the local neighborhood of a non-degenerate critical point $p$, and that such critical points are isolated. The number of minuses is the index of $p$.

The critical points of a Morse function and their indices capture information about the manifold on which the function is defined.

**Definition 2.2.10 (Euler Characteristic I)** Let $f \colon \mathbb{M} \to \mathbb{R}$ be a Morse function. The Euler characteristic of the manifold $\mathbb{M}$ equals the alternating sum of critical points of $f$, $\chi(\mathbb{M}) = \sum_x (-1)^{\text{index}\, x}$.

There are several equivalent formulations of the Euler characteristic; we will see another in Section 2.4.

## 2.3 Level Sets and The Reeb graph

**Definition 2.3.1 (Level Set)** A *level set* of a function $f$ consists of all points in the domain whose function values are equal to a chosen real number $s$.

A level set of $f$ is not necessarily connected. If we call two points $x, y \in \mathbb{M}$ *equivalent* when $f(x) = f(y)$ and both points belong to the same component of the level set, then we obtain the *Reeb graph* as the quotient space in which every equivalence class is represented by a point and connectivity is defined in terms of the quotient topology [Ree46]. Figure 2.1 illustrates the Reeb graph for a function defined on a 2-manifold of genus two, namely the function that maps each point to its distance above a horizontal plane below the surface. I call a point on the Reeb graph a *node* if the corresponding level set component passes through a critical point of $f$. The rest of the Reeb graph consists of arcs connecting the nodes. The *degree* of a node is the number of arcs incident to the node. A minimum creates and a maximum destroys a level set component and

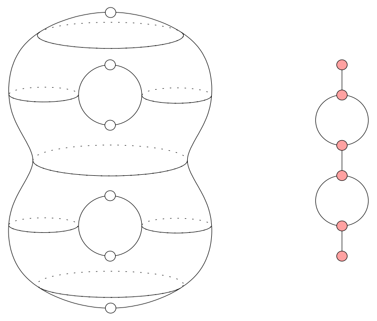


Figure 2.1: The Reeb graph of the function $f$ on a 2-manifold that maps every point of the double torus to its distance above a horizontal plane below the surface.

both correspond to degree-1 nodes. A saddle that splits one level set component in two or merges two to one corresponds to a degree-3 node. There are also saddles that alter the genus but do not affect the number of components, and they correspond to degree-2 nodes in the Reeb graph. Nodes of degree higher than three occur only for non-Morse functions.

## 2.4 Piecewise Linear Functions on Simplicial Complexes

Spaces and functions represented and manipulated on a computer are often derived from a discrete sampling of their smooth counterparts. I provide definitions of such discrete spaces and continuous PL functions defined on them.

**Definition 2.4.1 (Simplex)** A *$k$-simplex* is the convex hull of $k+1$ affinely independent points.

**Definition 2.4.2 (Face)** A *face* $\tau$of a simplex $\sigma$is the simplex defined by a non-empty subset of the $k+1$ points of $\sigma$, and is denoted $\tau \leq \sigma$.

**Definition 2.4.3 (Simplicial Complex)** A *s*implicial complex $K$ is a finite collection of non-empty simplices, such that every face of a simplex is also in $K$, and any two simplices intersect in a common face or not at all.

**Definition 2.4.4 (Underlying Space)** The *underlying space* of a simplicial complex $K$ is the union of simplices $|K| = \bigcup_{\sigma \in K} \sigma$.

**Definition 2.4.5 (Triangulation)** A *triangulation* of a manifold $\mathbb{M}$ is a simplicial complex, $K$, whose underlying space is homeomorphic to $\mathbb{M}$ [Ale98].

**Definition 2.4.6 (Barycentric coordinates)** The *barycentric coordinates* of a point $x$ inside a $k$-simplex $\sigma$ are the unique set of $k+1$ non-negative real numbers $w_i$ satisfying $\sum_i w_i = 1$ such that point $x$ is the weighted combination of the vertices of $\sigma$: $x = \sum_i w_i u_i$,

**Definition 2.4.7 (Piecewise Linear Function)** The value of a *piecewise linear function* at $x$ is the sum of the function values at the vertices of $\sigma$ weighted by their barycentric coordinates: $f(x) = \sum_i w_i f(u_i)$.

**Definition 2.4.8 (Euler Characteristic II)** Let $K$ be a triangulation of a manifold $\mathbb{M}$. The Euler characteristic of $\mathbb{M}$ is the alternating sum of simplices of $K$, $\chi(\mathbb{M}) = \sum_\sigma (-1)^{\dim \sigma}$.

I need some definitions to talk about the local structure of the triangulation and the function.

**Definition 2.4.9 (Star)** The *star* of a vertex $u$, denoted $\operatorname{St} u$, consists of all simplices that share $u$, including $u$ itself.

$$\operatorname{St} u = \{\sigma \in K \mid u \subseteq \sigma\}$$

**Definition 2.4.10 (Link)** The *link* of a vertex $u$, denoted $\operatorname{Lk} u$, consists of all faces of simplices in the star that are disjoint from $u$.

$$\operatorname{Lk} u = \{\tau \in K \mid \tau \subseteq \sigma \in \operatorname{St} u,\, u \notin \tau\}$$

**Definition 2.4.11 (Lower Link)** The *lower link*, denoted $\operatorname{Lk}_- u$, is the subset of the link induced by vertices with function value less than $u$.

$$\operatorname{Lk}_- u = \{\tau \in \operatorname{Lk} u \mid v \in \tau \Rightarrow f(v) \leq f(u)\}$$

Banchoff [Ban70] introduces the critical points of piecewise linear functions as the vertices whose lower links have Euler characteristic different from unity. A classification based on the reduced Betti numbers of the lower link is finer than that defined by Banchoff. I need some definitions from Algebraic topology before we describe the classification.

**Definition 2.4.12 ($k$-chain)** A $k$-chain is a subset of $k$-simplices.

We can impose a group structure on $k$-chains by defining an addition operation. The *sum* of two $k$-chains $c, d$ is the symmetric difference between the two sets: $c + d = (c \cup d) - (c \cap d)$. This operation is addition modulo 2 because a simplex is in the sum if it belongs to exactly one of $c$ or $d$. We denote a set of $k$-chains as $\mathsf{C}_k$ and the group of $k$-chains as $(\mathsf{C}_k, +)$.

**Definition 2.4.13 (Boundary of a Simplex)** The boundary of a $k$-simplex is the set of its $(k-1)$-simplex faces: $\partial_k(\sigma) = \{\tau \leq \sigma \mid \dim \tau = \dim \sigma - 1\}$.

**Definition 2.4.14 (Boundary of a $k$-chain)** The boundary of a $k$-chain is the sum of the boundaries of its simplices: $\partial_k(c) = \sum_{\sigma \in c} \partial_k(\sigma)$.

We can connect chain groups of successive dimensions by homomorphisms $\partial_k$ that map chains in $\mathsf{C}_k$ to their boundary in $\mathsf{C}_{k-1}$.

**Definition 2.4.15 (Chain Complex)** The *chain complex* of a simplicial complex $K$ is the sequence of its chain groups connected by boundary homomorphisms,

$$\ldots \xrightarrow{\partial_{k+2}} \mathsf{C}_{k+1} \xrightarrow{\partial_{k+1}} \mathsf{C}_k \xrightarrow{\partial_k} \mathsf{C}_{k-1} \xrightarrow{\partial_{k-1}} \ldots$$

**Definition 2.4.16 ($k$-cycle)** A *$k$-cycle* is a $k$-chain $c$ with empty boundary: $\partial_k(c) = \emptyset$.

**Definition 2.4.17 ($k$-boundary)** A *$k$-boundary* is the boundary of a $k+1$-chain.

The $k$-boundary and $k$-cycles are subgroups of $k$-chains, and are denoted $\mathsf{B}_k$ and $\mathsf{Z}_k$, respectively. The boundary of every boundary is empty: $\partial_{k-1}(\partial_k(c)) = \emptyset$. This implies that every $k$-boundary is a $k$-cycle, and results in the nested group structure shown in Figure 2.2.

**Definition 2.4.18 (The $k$-th Homology Group)** The $k$-th homology group is the $k$-th cycle group factored by the $k$-th boundary group: $\mathsf{H}_k = \mathsf{Z}_k/\mathsf{B}_k$.

**Definition 2.4.19 (The $k$-th Betti Number)** The $k$-th Betti number is the rank of the $k$-th homology group: $\beta_k = \operatorname{rank} \mathsf{H}_k$.

The Betti numbers of a space can be used to compute its Euler characteristic.

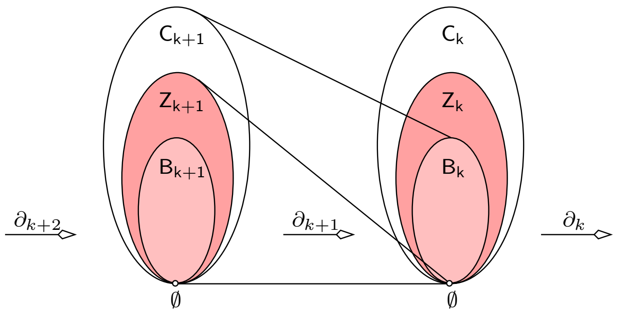


Figure 2.2:The Chain complex. Boundary homomorphism $\partial_k$ connects $\mathsf{C}_k$ to $\mathsf{C}_{k-1}$.

**Definition 2.4.20 (Euler Characteristic III)** The Euler characteristic of the manifold $\mathbb{M}$ equals the alternating sum of Betti numbers, $\chi(\mathbb{M}) = \sum_i (-1)^i \beta_i$.

We find that the reduced Betti numbers produce a more elegant classification for critical points than Betti numbers. This classification is shown in Table 2.1.

The *k-th reduced Betti number*, denoted as $\tilde{\beta}_k$, is the rank of the $k$-th reduced homology group of the lower link: $\tilde{\beta}_k = \operatorname{rank} \tilde{\mathsf{H}}_k$. The reduced Betti numbers are the same as the Betti numbers, except that $\tilde{\beta}_0 = \beta_0 - 1$ for non-empty lower links, and $\tilde{\beta}_{-1} = 1$ for empty lower links [Mun84].

When the link is a 2-sphere only $\tilde{\beta}_{-1}$ through $\tilde{\beta}_2$ can be non-zero. Simple critical points have exactly one non-zero reduced Betti number, which is equal to 1; see Table 2.1 and Figure 2.3. The first case in which this definition differs from Banchoff's is

| | $\tilde{\beta}_{-1}$ | $\tilde{\beta}_0$ | $\tilde{\beta}_1$ | $\tilde{\beta}_2$ |
|---|---|---|---|---|
| regular | 0 | 0 | 0 | 0 |
| minimum | 1 | 0 | 0 | 0 |
| 1-saddle | 0 | 1 | 0 | 0 |
| 2-saddle | 0 | 0 | 1 | 0 |
| maximum | 0 | 0 | 0 | 1 |

Table 2.1: Classification of vertices into regular and simple critical points using the reduced Betti numbers of the lower link.

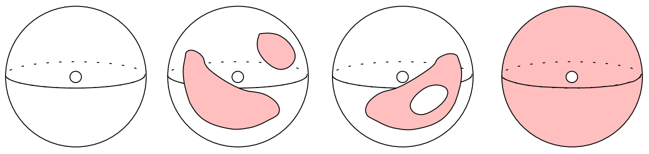


Figure 2.3: Lower links, shown shaded, for $d = 3$. From left to right, a minimum, a 1-saddle, a 2-saddle, and a maximum.

a double saddle obtained by combining a 1- and a 2-saddle into a single vertex. The Euler characteristic of the lower link is unity, which implies that Banchoff's definition does not recognize it as critical.

A *multiple saddle* is a critical point that falls outside the classification of Table 2.1 and therefore satisfies $\tilde{\beta}_{-1} = \tilde{\beta}_2 = 0$ and $\tilde{\beta}_0 + \tilde{\beta}_1 \geq 2$. By modifying the simplicial complex, it can be unfolded into simple 1-saddles and 2-saddles as explained in [CSA03, EHNP03]. This allows us to develop algorithms assuming that all critical points are simple.

The Betti numbers of a space count its topological features. The first three Betti numbers are the number of connected components, the number of tunnels, and the number of voids respectively. For example, a 2-torus has Betti numbers: $\beta_0 = \beta_2 = 1$, and $\beta_1 = 2$.

For $d = 3$ the link is a 2-sphere and the Betti numbers can be computed as follows: Compute the Euler characteristic $\chi$ of the lower link as the alternating sum of vertices, edges and faces in the lower link. Compute $\beta_0$, the number of connected components in the lower link, by using the union-find data structure [CLR94]. If all the link vertices are also in the lower link then $\beta_2 = 1$ else $\beta_2 = 0$. Compute $\beta_1$ using the relation $\beta_1 = \beta_0 + \beta_2 - \chi$. The reduced Betti numbers can be computed from the definitions. For an algorithm to compute Betti numbers of simplicial complexes on the 3-sphere see [DE95].

# Chapter 3

# Related Work

Many scientific studies that model and simulate physical processes on a computer produce large amounts of data that is interpreted using computational tools. Such data arises in a wide variety of studies including computational fluid dynamics [CM97], oceanography [BSRF00], medical imaging [TO91], and climate modeling [MCW00]. Popular tools for analyzing scientific datasets are *direct volume rendering*, and *isocontouring.*

Direct volume rendering displays all the data simultaneously and requires recomputing the image for each new viewing direction. Direct volume rendering employs two classes of algorithms to display all the data: image-space projection and volume-space projection. Image-space projection algorithms cast rays into the 3-dimensional volumetric data, map the data at each volume element to a user determined color and opacity value, accumulate these values in front-to-back order and display them on screen [KH84, Lev90, KS86]. Volume-space projection algorithms, traverse the volume, compute the color and opacity contribution for each volume element and project it onto the image screen [Han90, Wes89].

Isocontours (also called *level sets*) reduce 3-dimensional volumetric data to a 2-dimensional form suitable for interactive display. Isocontour-based visualization fixes a scalar value $s$, computes the points in space with that value, connects them into a triangulated mesh, and displays the results [LC87]. By varying $s$ we can explore the variation in the data. I will focus on level-set-based techniques in this Chapter.

Visualization is an interactive process so algorithms for isocontour extraction focus on efficiency. Moreover time-varying scalar fields are large; often too large to fit into physical memory. Therefore reducing storage overhead and improving I/O efficiency are important.

Isocontour extraction algorithms for time-varying scalar fields use three techniques to increase efficiency: spatial techniques, discussed in Section 3.1.1, span space techniques, discussed in Section 3.1.2, and topological techniques, discussed in Section 3.1.3. I compare these techniques based on type of data representations, I/O efficiency, and the type of output.

Efficient extraction of isocontours is important for interactive visualization, but so is selecting meaningful isovalues, particularly when the parameter space is large. Topological structures such as the Reeb graph are useful aids in visualization. The Reeb graph encodes isocontours' topological features, such as number of components, component merge, split and genus change, and can support the analysis of scientific data by providing the user with a succinct summary of the data. I will describe topological structures for supporting visualization in Section 3.2.

## 3.1 Isocontour Extraction from Time-varying Scalar Fields

Three techniques have been used to increase efficiency of isocontour extraction: spatial techniques, span space techniques, and topological techniques. I describe each technique for isocontour extraction from static data, and then describe how they generalize to time-varying data.

I begin with some notation. Static volumetric data consists of finite point samples of a scalar function $f\colon \mathbb{R}^3 \to \mathbb{R}$. The sample points are connected into a mesh, and the sampled scalar values are extended to the entire domain by interpolation. The characteristics of the sampling often suggest the type of mesh, and a natural interpolant. Irregular samples suggest using a simplicial mesh and a piecewise linear interpolant. Regular gridded samples suggest using a cubical mesh and piecewise trilinear interpolant. We refer to the sample points as *vertices* and each mesh element as a *cell.* With linear or multilinear interpolation, the *min-max interval* of interpolated values for each cell $c$, is the interval $[\mathit{minval}(c), \mathit{maxval}(c)]$, where $\mathit{minval}(c)$, and $\mathit{maxval}(c)$ are the minimum and maximum scalar values at the vertices of $c$.

**History of isocontour-based visualization.** Research in isocontour extraction began in the medical imaging field, with algorithms to extract boundaries of organs from sampled medical images [HL79, Art79, FKU77]. The graphics and animation community used isocontours to model and render implicit surfaces [Bli82], and to model and animate objects [WMW86a, WMW86b].

Lorensen & Cline [LC87] introduce the *marching cubes* algorithm for isocontour extraction from data sampled regularly and connected into a cubical mesh. The algorithm iterates through all cubes in the volume, and extracts a piece of the isocontour from each cube. Each cube vertex can be classified as "above" or "below" the isocontour. For each cube edge that has vertices of opposite polarity, an isocontour vertex can be computed by linear interpolation and these isocontour vertices can be connected into one or more pieces of a triangular mesh. The connectivity is dictated by a reduced set of 15 cases derived from the possible $2^8 = 256$ cases of classifying the cube vertices. The marching cubes algorithm has a flaw [Dür88]; it sometimes generates isocontours with holes. Subsequent research provide algorithms that correct this problem [GW94, Mat94, MSS94, Nat94, NH91]. Although simple, the marching cubes algorithm can be inefficient because it examines the entire volume, while an isocontour may intersect only a small fraction of this volume. Techniques that speed-up isocontour extraction build search structures to efficiently find cells intersecting the isocontour, and can be classified into spatial techniques, span space techniques, and topological techniques.

### 3.1.1 Spatial Techniques for Efficient Isocontour Extraction

Spatial techniques subdivide the volume using a octree based hierarchy [WG92]. Each octant is equipped with the min-max interval of function values contained in it, enabling a search through the octree hierarchy to terminate at the octant if the query isovalue does not belong to the interval. This technique works because most data sets have large regions with function values that are distributed close together and can be quickly pruned in a search. Next we look at an extension of the octree technique to time-varying data.

#### Temporal Branch-on-need Tree

The Temporal Branch-on-Need Tree (T-BON) [SH99] extends the three dimensional branch-on-need octree for time-varying isocontour extraction. It aims at efficient I/O while retaining the accelerated search enabled by a hierarchical subdivision. Unlike the algorithms in Sections 3.1.2 and 3.1.3 this algorithm does not exploit temporal coherence and considers each time-step as a static volume.

**Construction.** The input to the T-BON construction is regularly sampled points in space-time. The output is a branch-on-need octree for each time step, stored on disk in two sections. Store information common to all trees, such as branching factor, pointers to children and siblings, once for all trees. Store the min-max intervals associated with nodes of an octree separately as a linear array, one per time-step, each packed in depth-first order.

**Search and extraction.** The problem is to extract the isocontour for isovalue $s$ at time $t$ as a triangulated mesh. As a first step, read the octree infrastructure from disk and re-create it in main memory. Resolve the query using the octree for $t$ and demand-driven paging [CE97]. Read the min-max interval of the root from disk, and if $s$ belongs to this interval read the child nodes. Proceed recursively, stopping at the leaf nodes. If the min-max interval of a leaf contains $s$ add the disk block containing the corresponding cells values onto a list. After completing the octree traversal read all disk blocks from the list into memory. Repeat the tree traversal, this time extracting the isocontour using the cell data read from disk.

### 3.1.2 Span-space Techniques for Efficient Isocontour Extraction

Livnat et al. [LSJ96] define the *span space* as the two dimensional space spanned by the minimum and maximum values of the cells of the volume. A cell $c$ with minimum value $min_c$ and maximum value $max_c$ maps to a point $(min_c, max_c)$ in the span space. See Figure 3.1. A variety of search structures on the span space have been used to speed-up finding cells that intersect an isocontour. Gallagher [Gal91] uses bucketing and linked lists, Livnat et al. [LSJ96] use k-d trees [Ben75], van Kreveld [vK94] and Cignoni et al. [CMM$^+$97] use the interval tree for two- and three-dimensional data respectively. Chiang et al. [CSS98] use a variant of the interval tree that enables out-of-core isocontour extraction, and use the algorithm of Section 3.1.2 to extend this work to time-varying data [Chi03]. Unlike the spatial techniques that use the octree, which requires regularly gridded data, span space techniques is also applicable to irregularly sampled data. Next we look at an algorithm that uses span space techniques for isocontour extraction from time-varying data.

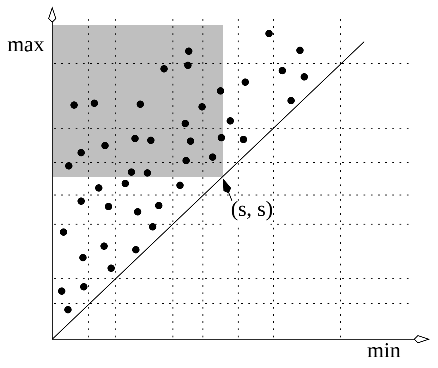


Figure 3.1: Points in shaded area of span space correspond to cells that intersect isocontour for value $s$.

### Temporal Hierarchical Index (THI) Tree

Shen's algorithm [She98] for the Temporal Hierarchical Index (THI) tree analyzes the span space of the time-varying data, and classifies and stores each cell in one or more nodes of a binary tree based on the temporal variation of its values. By placing a cell possessing a pre-defined small temporal variation in a single node of the THI tree, along with a conservative min-max interval of its variation over a large time span, this algorithm achieves savings in space. Cells with greater temporal variation are stored at multiple nodes of the tree multiple times, each for a short time span.

**Temporal variation.** Shen uses the span space to define the temporal variation of a cell's values. The area over which the points corresponding to a cell's min-max values over time are spread out give a good measure of its temporal variation; the larger the area of spread the greater the variation. In particular, subdivide the span space into $\ell \times \ell$ non-uniformly spaced rectangles called *lattice elements* using the *lattice subdivision* scheme [SHLJ96]. To perform the subdivision, lexicographically sort all extreme values of the cells in ascending order, find $\ell + 1$ values to partition the sorted list into $\ell$ sublists, each with the same number of cells. Use these $\ell + 1$ values to draw vertical and horizontal lines to get the required subdivision. Note that this subdivision does not guarantee that each lattice element has the same number of cells. Given a time interval $[i, j]$, a cell is defined to have low temporal variation in that interval if its $j - i + 1$ min-max interval points are located over an area of $2 \times 2$ lattice elements.

**Construction.** The input of the THI algorithm is a fixed mesh whose vertices are points in space. Each point has a data value for each time step in $[0, T - 1]$. Each cell has $T$ corresponding min-max intervals. The output of the THI algorithm is a binary tree constructed as follows. In the root node $N_0^{T-1}$, store cells whose min-max intervals have low temporal variation in the time interval $[0, T-1]$. The root has two children, $N_0^{T/2}$ and $N_{T/2+1}^{T-1}$ defined recursively on cells that are not stored in the root. Recursion stops at leaf nodes $N_t^t$, with $t \in [0, T-1]$. Cells that fall into leaf nodes have the highest temporal variation. See Figure 3.2.

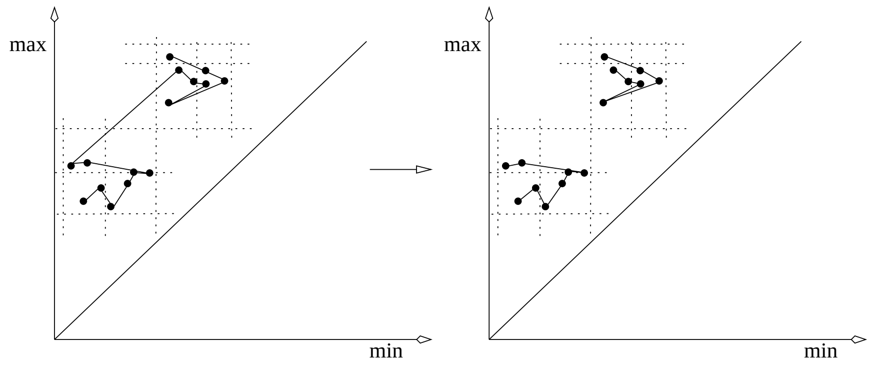


Figure 3.2: The min-max intervals of a cell over a time interval are shown in the span-space as points with a path connecting them in order of time. The points on the left are spread outside a $2 \times 2$ lattice area. On breaking the time interval into two halves, on the right, the respective points fall inside a $2 \times 2$ area.

Represent each cell that falls into an internal node $N_i^j$ by a conservative min-max interval, called the *temporal extreme values*, which contains all the cells min-max intervals for the time span $j - i + 1$. Because the temporal extreme values are used to refer to a cell for more than one time step, we get a reduction in the overall index size.

Within each tree node, organize the cells using one of the span-space based techniques; Shen uses a modified ISSUE algorithm [SHLJ96]. Use the lattice subdivision scheme described above, and sort cells that belong to each lattice row, excluding the cells in the diagonal lattice element, in ascending order, based on their minimum temporal extreme value. Similarly, sort the cells into another list, in descending order, based on their maximum temporal extreme value. Build an interval tree, as in [CMM+97], for cells in the lattice elements along the diagonal.

**Search and extraction.** As in the T-BON algorithm, the problem is to extract the isocontour for isovalue $s$ at time $t$ as a triangulated mesh. First collect all nodes in the THI-tree whose time span contains $t$. Traverse the tree, starting at the root node. From the current node, visit the child $N_i^j$, with $i \leq t \leq j$, stopping at leaf node $N_t^t$.

At each node in the traversal path, use the lattice index structure to locate candidate isocontour cells. Locate the lattice element that contains the point $(s, s)$. Because of the symmetry of the lattice subdivision this element is the $k^{th}$ row in the $k^{th}$ column, for some integer $k$. The isocontour cells are contained in the upper-left corner bounded by the lines $x = s$ and $y = s$, as shown in Figure 3.1. Collect these cells as follows:

- From each row $r = k+1$ to $\ell - 1$, collect cells from the beginning of the list sorted on the minimum temporal extreme value until the cell whose minimum is greater than $s$.
- From row $k$, collect cells from the beginning of the list sorted on the maximum temporal extreme value until a cell whose maximum is lesser than $s$.
- From the lattice element containing $(s, s)$, collect cells by querying the interval tree.

Recall that because we store a conservative min-max interval with cells, some collected cells may not actually intersect the isocontour. For all candidate cells, read the actual data at time $t$ to extract the isocontour.

### 3.1.3 Topological Techniques for Efficient Isocontour Extraction

Topological techniques for efficient isocontour extraction typically analyze the data in a preprocess step, and compute a subset of cells called the *seed set*. A seed set contains at least one cell, called a *seed*, intersecting each component of each isocontour. To extract an isocontour first search the seed set, which is stored in an appropriate search structure, to find a seed for each connected isocontour component. To extract each component, start at the seed and visit all intersecting cells by a breadth first search of the mesh. This method of extraction, by performing a breadth first search of the mesh, is called *continuation* by Wyvill et al. [WMW86a], *mesh propagation* by Howie & Blake [HB94], and *contour-following* by [CS03]. Next we look at an algorithm that uses topological techniques for isocontour extraction from time-varying scalar fields.

#### Progressive Tracking

Bajaj, Shamir & Sohn [BSBS02] extend seed-set-based techniques to time-varying data. They use temporal coherence to compute an isocontour at time step $t + 1$ by modifying the isocontour computed at the time step $t$. New components at $t+1$ are separately computed from the seed set for that time step. Unlike the T-BON and THI algorithms they accept a range of time-steps and a single isovalue as arguments, and track components over the range of time-steps.

**Construction.** The input can be a fixed regular or irregular mesh whose vertices are points in space. Each point has a data value for each time step in $[0, T-1]$. The output is a collection of $T$ seed sets, one for each time step. Treat each time-step as a static scalar field and compute a seed-set using one of the algorithms proposed in [CS03, vKvOB$^{+}$97]. Organize each seed-set in an interval tree [CMM$^{+}$97].

**Search and extraction.** The query for this algorithm is different from the T-BON and THI algorithms: Find isocontours for isovalue $s$ over the time range $[t_0, t_1]$ as a collection of triangulated meshes. To extract an isocontour, first extract all isocontour components at time $t_0$ using contour propagation from the seeds for that time. Extract all isocontour components for subsequent discrete time steps, $t \leq t_1$ by a combination of modifying current isocontour components over time, and by extracting new components from the seed sets at $t$.

To modify components, at each time-step $t$, with $t_0 \leq t \leq t_1$, maintain, in one list per isocontour component, the set of intersecting cells. These cell lists are used to track the evolution of the isocontour for $s$ over time. As in the marching cubes case, we can label each cell vertex as "above" if its value at time $t$ is greater than $s$, and "below" otherwise. A cell edge joining opposite labels contains an isocontour vertex. To track an isocontour component, consider the label change for each cell edge at $t+1$. If the labels of the end vertices of a cell edge do not change then the isocontour vertex that lies on that edge just changes position, which can be found by linear interpolation. A cell vertex label can change, in which case the isocontour experiences a local connectivity change. The changes include the contour changing its geometry but not topology, two or more components merging, or a component splitting into two or more components. All these changes can be applied to the contours by enqueueing the cell lists, extracting each cell and examining its neighborhood, and propagating the isocontour spatially. See [BSBS02] for details.

### 3.1.4 Comparison

The TBON algorithm discussed in Section 3.1.1 can be applied to regularly gridded data and is designed to be I/O-efficient by using demand driven paging; load data only when it is required. It does not exploit temporal coherence and treats each time-step as a static volumetric scalar field. The THI algorithm discussed in Section 3.1.2 can be applied to both regularly and irregularly sampled data. It reduces the storage overhead of the search structure, but it requires all data to be loaded into memory, a serious drawback for time-varying data which can be large. The progressive tracking algorithm discussed in Section 3.1.3 can also be applied to regularly and irregularly sampled data. Since this algorithm computes seed sets the storage overhead is not significant. Bajaj et al. [BSBS02] show seed set sizes of less than 2% of the total number of cells. Moreover, extracting isocontours by propagation produces coherent triangulations that are amenable to compression [IG03], simplification [ILGS03], and streaming [IL04]. The other extraction algorithms do not produce coherent triangulations.

## 3.2 Topological Structures for Supporting Visualization

The Reeb graph, defined in Chapter 2, encodes the number of isocontour components at each isovalue, and the topological changes experienced by components. The Reeb graph can be used to display this information succinctly to aid visualization. Section 3.2.2 reviews an algorithm that extends the Reeb graph for visualizing and analyzing time-varying scalar fields.

**Visualization interfaces.** Topological structures, such as the Reeb graph, provide a succinct summary of the underlying function, and are used in visualization interfaces. Figure 3.3 shows a screenshot of the *contour spectrum* [BPS97] interface; a window displays isocontour properties, such as surface area and volume, using graphs, and topological properties using the Reeb graph, to aid selection of interesting isocontours which are displayed separately. Typically the functions are defined on simply-connected domains and their Reeb graphs are loop-free, and are also known as contour trees.

Figure 3.4 shows the safari interface [KRS03], which extends the ideas of the contour spectrum to time-varying data, provides the user with a *(time, value)* control plane for isovalue selection, and extracts an isocontour from a time-slice for display. The control

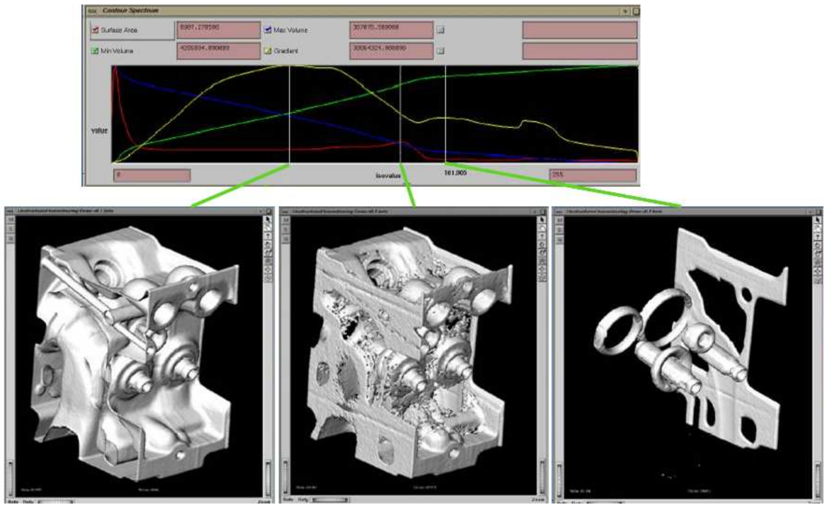


Figure 3.3: The contour spectrum interface [BPS97]. On the top, graphs of isocontour properties, such as surface area, and volume, versus isovalues. On the bottom, three isocontours chosen using the contour spectrum.

plane on the right displays the number of isocontour components for each time-step($x$-axis) and isovalue($y$-axis), which can be computed from the contour tree for each time step.

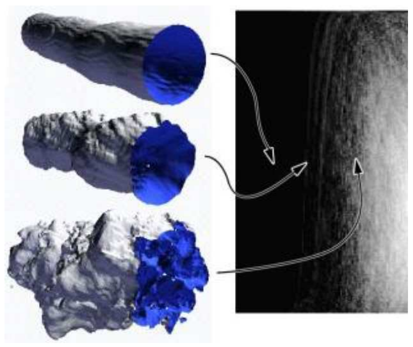


Figure 3.4: A portion of the safari interface [KRS03]. The control plane on the right displays the number of isocontour components for each time-step($x$-axis) and isovalue($y$-axis). The user selects isocontours for display by clicking on the contour plane.

Recently, Carr et al. [CS03] have used the contour tree to compute *path seeds*, a set of edges in the triangulation that can be used to quickly find seeds for any isocontour component, and use these path seeds to create a *flexible isocontour* interface. They provide the user with an interface to select individual arcs in the contour tree and can extract chosen isocontour components for display. Building on this work, they also present an algorithm to simplify the contour tree using local geometric measures to capture the essential features of the underlying data in the presence of noise [CSvdP04].

### 3.2.1 Reeb Graph Algorithms

In mathematics, the Reeb graph is often used to study the manifold $\mathbb{M}$ that forms the domain of the function. In visualization, on the other hand, the Reeb graph is used to study the behavior of the function. The domain of interest is $\mathbb{R}^3$ but it is convenient to compactify it and consider functions on the 3-sphere, $\mathbb{S}^3$. All the Reeb graphs for such functions reveal the (un-exciting) connectivity of $\mathbb{S}^3$ by being trees, but the structure of the tree tells us how and at what level value the level sets of the chosen function $f$ change topology.

**History.** The Reeb graph was first introduced in [Ree46]. In the field of visualization, Boyell and Ruston [BR63] introduced the contour tree to summarize the evolution of contours on a map. In the interactive exploration of scientific data, Reeb graphs are used to select meaningful level sets [BPS97] and to efficiently compute them [vKvOB+97]. An extensive discussion of Reeb graphs and related structures in geometric modeling and visualization applications can be found in [FTLK97].

Published algorithms for Reeb graphs take as input a function defined on a triangulated manifold. We express their running times as functions of $n$, the number of simplices in the triangulation. The first algorithm for functions on 2-manifolds due to Shinagawa and Kunii [SK91] takes time $O(n^2)$ in the worst case. Algorithms to compute contour trees have received special attention because of the practical importance of simply-connected domains, and because the algorithms are simpler. An algorithm that constructs contour trees of functions on simply-connected manifolds of constant dimension in time $O(n \log n)$ has been suggested in [CSA03]. For the case of 3-manifolds, this algorithm has been extended to include information about the genus of the level surfaces [PCM03]. Cole-McLaughlin et al. [CMEH+03] return to the general case, giving tight bounds on the number of loops in Reeb graphs of functions on 2-manifolds and describing an $O(n \log n)$ time algorithm to construct them.

**Computing a contour tree.** We sketch the algorithm proposed by Carr et al. [CSA03] to compute the contour tree of a function defined on a simply-connected domain. Although their algorithm works for any dimension we restrict our description to $d = 3$. This algorithm in used in Sections 3.2.2 and by the time-varying Reeb graph algorithm described in Chapter 4.

The input to the contour tree algorithm is a triangulation $K$ of a simply-connected 3-manifold, with each vertex equipped with a distinct scalar function value. The output is the contour tree of the function represented as a collection of nodes and arcs that connect them. The algorithm proceeds in two passes: the first computes the *Join tree*, and the second the *Split tree*. Since these passes are similar we describe the construction of the Join tree. The Join tree encodes the merges experienced by the isocontour components as we sweep the isovalue from $-\infty$ to $\infty$; the Split tree does this for the sweep in the opposite direction.

To implement the sweep sort the vertices of $K$ in ascending order of function value, and iterate through the vertices. At each step maintain the collection of vertices visited in a union-find(UF) data structure [CLR94]. For each connected component in the collection, maintain a tree encoding the merge history of the component. Classify each vertex $v$ based on its index and handle each case as follows. Add a regular point to the set that contains its lower link vertex. A minimum creates a new component; start a UF set, and a new Join tree arc. An index-1 critical point locally merges two components. If the components are not connected globally, then create a join node that merges two arcs and starts a new one, else create a join node that ends a single arc, and starts a new one. The latter case corresponds to the component experiencing a genus change. An index-2 critical point is handled in the split sweep. Only the global maximum appears in the Join tree; it ends the arc corresponding to the component that disappears at the maximum.

Finally, construct the contour tree by merging the Join and Split tree [CSA03]. In Figure 3.5 we see two stages during the construction of the Join tree for a function defined on a 2-manifold.

### 3.2.2 Time-varying Contour Topology

The analysis of time-varying scalar fields often requires the tracking of isocontour components of a fixed isovalue over time, and detecting when and how these components change topology. For example, in the analysis of molecular dynamics simulation data isocontour components of a carefully chosen isovalue can represent individual molecules. Keeping isovalue fixed while varying time and tracking the these components can reveal possible bond formation between molecules when the components merge.

Szymczak [Szy05] utilizes his earlier work in computing subdomain aware contour trees [BS04] to compute how isocontours merge or split over a user specified interval of

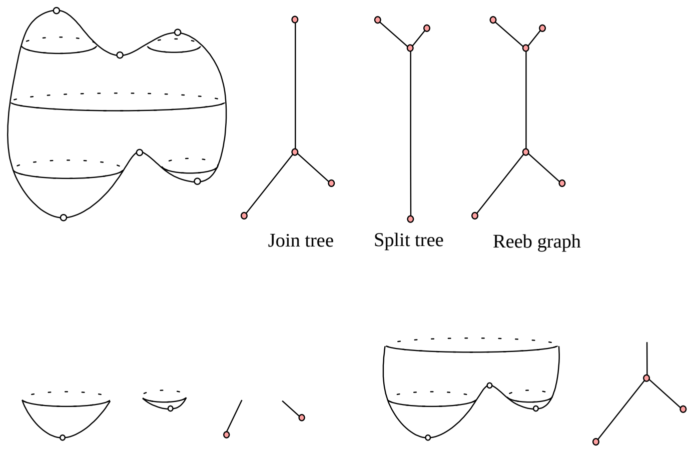


Figure 3.5: In the top row, a 2-manifold shown with three isocontours for the height function defined on it, and the Join, Split tree and Reeb graph. In the bottom row, two stages during the sweep to construct the Join tree.

time. Sohn & Bajaj [SB05] track isocontour components over time by computing the correspondence of contour trees over time. They assume that the scalar field can change unpredictably between two successive time steps, and define temporal correspondence of contour tree arcs for successive time steps using a notion of an overlap between an isocontour at time $t$ with an isocontour at time $t+1$. They develop an algorithm to compute correspondence between successive contour trees based on this definition, and use the correspondence to track isocontour components and their topology over time.

**Temporal contour correspondence.** Before we define temporal contour correspondence we need some notation. Define $\mathbb{X}$ and $\mathbb{Y}$ as the restrictions of the domain to time $t$ and $t+1$ respectively, function $f_t$ as the restriction of function $f$ to $\mathbb{X}$, and isocontours $I = f_t^{-1}(s)$ and $J = f_{t+1}^{-1}(s)$. Isocontour $I$ has connected components $I = \{I_1, \cdots, I_m\}$, and lies on the intersection of $\mathbb{X}_{\leq s} = f_t^{-1}(-\infty, s]$ and $\mathbb{X}_{\geq s} = f_t^{-1}[s, \infty)$. Identify each $I_i$ with one component of $\mathbb{X}_{\leq s}$ and $\mathbb{X}_{\geq s}$; $I_i$ belongs to their intersection. Sohn & Bajaj call these components the *lower object* and *upper object* of $I_i$, respectively. Similar definitions hold for the isocontour $J$.

Two contour components $I_i$ and $J_j$ exhibit *temporal correspondence* if their corresponding upper objects overlap and their corresponding lower objects overlap. Note, that an overlap between $\mathbb{X}_{\leq s}$ and $\mathbb{Y}_{\leq s}$ makes sense only if we assume that both $f_t$ and $f_{t+1}$ are defined on the same domain. See Figure 3.6 for an example.

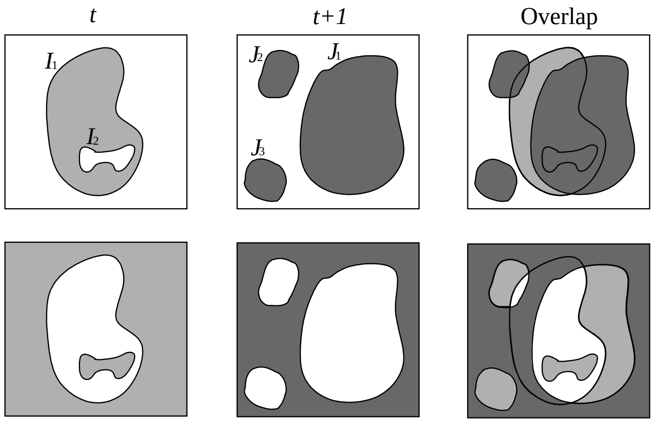


Figure 3.6: Temporal correspondence between isocontours at successive time steps. In the top row, upper objects for each isocontour and their overlap, in the bottom row, lower objects and their overlap. From the definitions in [SB05], we get the following correspondence: $(I_1 \to J_1)$, $(I_1 \to J_2)$, $(\emptyset \to J_3)$, $(I_2 \to \emptyset)$.

**Algorithm for contour correspondence.** Since each isocontour component corresponds to a point on an arc on the contour tree, temporal correspondence between isocontour components can be used to compute a correspondence between arcs of the contour tree at $t$ with arcs of the contour tree at $t+1$. Sohn & Bajaj compute this correspondence by modifying the contour tree algorithm of Carr et al. [CSA03]. In the description that follows, we use $JT_t$, $ST_t$, $CT_t$ to denote the join tree, split tree, and contour tree at time $t$ respectively. The input is a simplicial mesh in $\mathbb{R}^3$ . Each point of the mesh is equipped with $T$ scalar values. The output is a collection of $T$ contour trees, with each arc of $CT_{t+1}$ labeled with arcs of $CT_t$. The labels indicate the temporal correspondence information. Since the join and split trees are symmetric we use the join tree for the description. For each time step $t$, pre-compute the contour tree, $CT_t$ [CSA03]. Label each tree arc with a unique id. The correspondence information is computed as follows.

1. Augment $JT_t$ with the nodes that appear only in $ST_t$. Carr et al. [CSA03] call the resulting join tree the augmented join tree.

2. Equip each arc $a$ of $JT_t$ with the ids of the corresponding arcs from $CT_t$. An isocontour component corresponding to a point on $a$, lies on the boundary of a connected component of $\mathbb{X} \leq s$ (the lower object), which may contain other isocontour components. Each of these other isocontour components correspond to an arc of $CT_t$. Equip $a$ with the ids of these arcs of $CT_t$. This step can be done by simultaneously scanning the nodes of $CT_t$ and $JT_t$ in increasing order of

$f_t$, and incrementally maintaining the arc id lists for $JT_t$.

3. This is the step that computes the correspondence information for the arcs of $JT_{t+1}$. Simultaneously sweep the functions $f_t$ and $f_{t+1}$ in increasing order, as if constructing the join trees $JT_t$ and $JT_{t+1}$. The sweep can be thought of as incrementally generating the lower objects simultaneously in time $t$ and $t+1$, and detecting overlap. Recall that two isocontour components from successive time-steps exhibit temporal correspondence if their respective lower objects and upper objects overlap. This sweep tests lower object overlap, and the reverse sweep, for $ST_t$ and $ST_{t+1}$, will test overlap for upper objects. During the sweep, maintain a collection of lower objects in $\mathbb{X}_{\leq s}$ and $\mathbb{Y}_{\leq s}$. Each lower object corresponds to an arc of the join tree in its respective time-step. For lower objects in $\mathbb{X}_{\leq s}$ we know their corresponding contour tree arcs from step 2. When a lower object in $\mathbb{X}_{\leq s}$ overlaps one in $\mathbb{Y}_{\leq s}$ we can create the mapping between the arcs of $JT_{t+1}$ with the corresponding arcs of $JT_t$. For a more detailed description of this step see [SB05].

4. Map labels from the arcs of $JT_{t+1}$ to the corresponding arcs of $CT_{t+1}$. This step is similar to step 2.

After the above steps are repeated for $ST_{t+1}$, each arc of the contour tree $CT_{t+1}$ has two arc lists, one from $JT_{t+1}$ and the other from $ST_{t+1}$. The final arc list is the intersection of the two lists.

**Topology change graph.** Consider the isocontour for isovalue $s$ at time $t$. If we keep the isovalue fixed at $s$ and proceed forward in time to $t+1$, then the isocontour undergoes topological changes in the following possible ways: a component is created or destroyed, two components merge into one, a component splits into two, a component changes genus. The *Topology change graph(TCG)* captures the change in topology as a graph constructed as follows. At each time step compute the arcs of $CT_t$ that contain the isovalue $s$. Create a node in the TCG for each these arcs. Use the correspondence information computed for an arc of $CT_t$ to create a connection between the corresponding nodes in the TCG. See Figure 3.7. Genus change can be detected by examining the Betti numbers of the component at time $t+1$, which can computed using the algorithm of Pascucci et al. [PCM03].

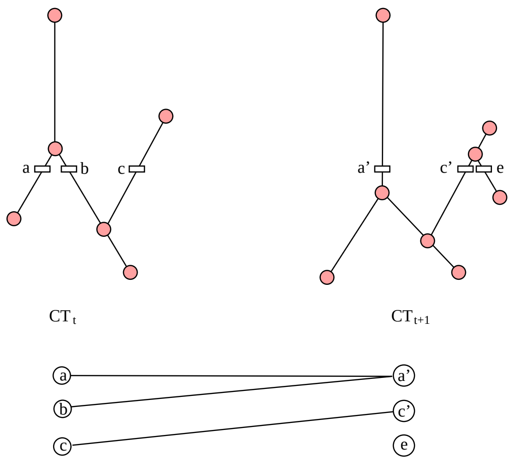


Figure 3.7: In the top row, a contour tree at successive time steps, with arcs containing isovalue $s$ labeled. In the bottom row, the topology change graph showing how contour components change topologically.

## 3.3 Conclusions

Isocontour extraction algorithms for time-varying scalar fields use spatial techniques, span space techniques, and topological techniques to increase efficiency. Spatial techniques organize space using an octree decomposition to detect regions that intersect and reject regions that do not intersect the isocontour. Span space techniques organize cells in the space of function values to efficiently detect cells that intersect the isocontour. Topological techniques organize a subset of cells, called the seed set, in a search structure. The seed set contains one intersecting cell for each connected component of each isocontour, and the component can be extracted by propagation from the intersecting cell. While span space techniques and topological techniques can be used for both regularly and irregularly sampled data, spatial techniques can be used only for regularly sampled data. Unlike spatial techniques and span space techniques, topological techniques allow isocontour extraction using contour propagation and produce coherent triangulations that are amenable to compression [IG03], simplification [ILGS03], and streaming [IL04]. This feature is useful for large data set visualization when the isocontours themselves might be too large to fit in memory.

Bajaj & Sohn [SB05] extend the use of the Reeb graph from static data to dynamic data by defining a overlap heuristic to connect the Reeb graph of each time slice; it works well when the time sampling rate is high relative to the phenomenon under study so that there is good temporal coherence.

# Chapter 4

# Time-varying Reeb Graphs for Continuous Space-time Data

In this chapter, I characterize the evolution of the Reeb graph of a time-varying continuous function defined in three-dimensional space. I maintain the Reeb graph over time and compress the entire sequence of Reeb graphs into a single, partially persistent data structure. I envision this data structure as a useful tool in visualizing real-valued space-time data obtained from computational simulations of physical processes.

## 4.1 Introduction

Physical processes that are measured over time, or modeled and simulated on a computer, can produce large amounts of data that must be interpreted with the assistance of computational tools.

Graphical visualization, often through level sets or iso-surfaces of a continuous function, is useful for interpreting the data. A level set consists of all points in the domain whose function values are equal to a chosen real number $s$. In $\mathbb{R}^3$, this is generically a surface that is then displayed. By varying $s$, we can study the variation in the data. Topological features of the level sets, such as connected components, handles, and voids, can be important aids in interpreting the data. The Reeb graph encodes the evolution of these features and compactly represents topological information of all level sets, as described in Section 2.3 and Section 3.2. As we move forward in time, the Reeb graph goes through an evolution of its own, undergoing structural changes at birth-death points and at interchanges of critical points. The evolution of the Reeb graph thus represents a 2-parameter family of level sets. I suggest that this 2-parameter family, encoded in a compact data structure, is a useful representation of space-time data.

In this chapter, I study how the Reeb graph of a smooth function defined on three-dimensional space evolves over time. It will be convenient to compactify the space to a closed manifold, which we do by adding the point at infinity, effectively creating the topology of the 3-sphere, denoted by $\mathbb{S}^3$. I establish a connection between Reeb graphs and Jacobi curves, which are the paths traced by the critical points of a time-varying function, in Section 4.3. I develop a complete enumeration of the type of combinatorial changes the Reeb graph experiences:

- nodes disappear in pairs, contracting arcs to points (inversely, node appear in pairs, anti-contracting arcs);
- nodes swap their positions along the arcs of the graph.

The second type of change falls into more sub-types and is algorithmically more difficult to handle than the first type. Based on my classification, I develop an algorithm that maintains the Reeb graph through time and stores its evolution in a partially persistent data structure in Chapter 4.4 and Chapter 4.5. The size of this data structure is proportional to the size of the initial Reeb graph at time zero, plus the number of changes it experiences through time. I close with a few open problems in Section 4.6.

## 4.2 Jacobi Curves

I use the Reeb graph to understand a function at moments in time and Jacobi curves, as introduced in [EH02], to help track the evolution of the Reeb graph through time. I define Jacobi curves of two Morse functions, $f, g \colon \mathbb{M} \to \mathbb{R}$, then specialize to a time-varying function on the 3-sphere.

For a regular value $t \in \mathbb{R}$, we have the level set $g^{-1}(t)$ and the restriction of $f$ to this level set, $f_t : g^{-1}(t) \to \mathbb{R}$. The *Jacobi curve* of $f$ and $g$ is the closure of the set of critical points of the functions $f_t$, for all regular values $t \in \mathbb{R}$. The closure operation adds the critical points of $f$ restricted to level sets at critical values, as well as the critical points of $g$, which form singularities in these level sets. Figure 4.1 illustrates the definition by showing the Jacobi curve of two smooth functions on a piece of the two-dimensional plane.

I can now specialize the definition of Jacobi sets to a time-varying function. Consider a 1-parameter family of generic Morse functions on the 3-sphere, $f : \mathbb{S}^3 \times \mathbb{R} \to \mathbb{R}$, where the extra dimension in the domain is time. I can use the general definition of Jacobi

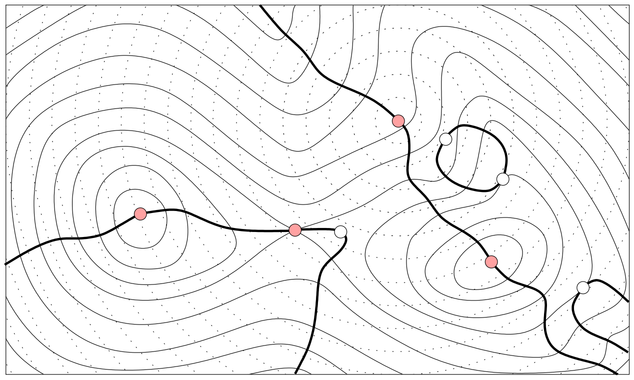


Figure 4.1: The functions $f$ and $g$ are represented by their dotted and solid level curves. The Jacobi curve is drawn in bold solid lines. The birth-death points and the critical points of the two functions are marked by white and shaded dots, respectively.

sets described above by introducing an auxiliary function $g : \mathbb{S}^3 \times \mathbb{R} \to \mathbb{R}$ defined by $g(x, t) = t$. A level set of $g$ has the form $g^{-1}(t) = \mathbb{S}^3 \times t$, and the restriction of $f$ to this level set is $f_t : \mathbb{S}^3 \times t \to \mathbb{R}$. Generically, the function $f_t$ is Morse, but there are discrete values of $t$ at which it violates one or both Genericity conditions of Morse functions. In the next section, we will see that the Reeb graph of $f_t$ undergoes combinatorial changes at these values. The Jacobi curve of generic Morse functions $f$ and $g$ may consist of several components, each component a closed 1-manifold.

We can identify the *birth-death points* where the level sets of $f$ and $g$ and the Jacobi curve have a common normal direction. To understand these points, imagine a level set in the form of a (two-dimensional) sphere deforming, sprouting a bud, as we go forward in time. The bud has two critical points, one a maximum and the other a 2-saddle. At the time when the bud just starts sprouting there is a point on the sphere, a birth point, where both these critical points are born. Run this in reverse order to understand a death point. I decompose the Jacobi curve into *segments* by cutting it at the birth-death points. The index of the critical point tracing a segment is the same everywhere along the segment. The indices within two segments that meet at a birth-death point differ by one:

Index Lemma. Let $f : \mathbb{M} \times \mathbb{R} \to \mathbb{R}$ be a 1-parameter family of Morse functions. The indices of two critical points created or destroyed at a birth-death point differ by exactly one.

Proof. At time $t$, let $f_t$ have a single birth point. We can choose a small positive $\varepsilon$ such that there are no other birth-death points with time in $[t - \varepsilon, t + \varepsilon]$. Denote by $x$ and $y$ the two newly created critical points in $f_{t+\varepsilon}$ and let $k = \operatorname{index} x \leq \operatorname{index} y$.

The point $x$ either destroys a homology class of dimension $k-1$ or it creates one of dimension $k$. The former case is ruled out as index $y \geq k$, and a cell of dimension larger than or equal to $k$ cannot compensate for the destroyed dimension $k-1$ class. In the latter case, when $x$ creates a homology class of dimension $k$, we need a $(k+1)$-cell to destroy the homology class, which implies that index $y = k+1$. The claim follows. ◻

## 4.3 Time-varying Reeb Graphs

In this section, I characterize how the Reeb graph of a function changes with time. Specifically, I give a complete enumeration of the combinatorial changes that occur for a Morse function on $\mathbb{S}^3$.

**Jacobi curves connect Reeb graphs.** Let $R_t$ be the Reeb graph of $f_t$, the function on $\mathbb{S}^3$ at time $t$. The nodes of $R_t$ correspond to the critical points of $f_t$, and as we vary $t$, they trace out the segments of the Jacobi curve. The segments connect the family through time, giving us a mechanism for identifying nodes in different Reeb graphs. I illustrate this idea in Figure 4.2. I define the *time-varying Reeb graph* as the family of Reeb graphs parameterized by time $t$, stored compactly in a partially persistent data-structure supporting access to Reeb graph $R_t$ at any time $t$.

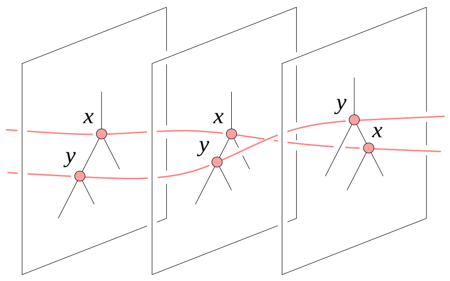


Figure 4.2: Reeb graphs at three moments in time whose nodes are connected by two segments of the Jacobi curve.

Generically, the function $f_t$ is Morse. However, there are discrete moments in time at which $f_t$ violates one or both Genericity conditions of Morse functions and the Reeb graph of $f_t$ experiences a combinatorial change. Since we have only one varying parameter, namely time, we may assume that there is a single violation of the Genericity Conditions at any of these discrete moments, and no violations at all other times. Condition I is violated whenever $f_t$ has a birth-death point at which a cancellation annihilates two converging critical points or an anti-cancellation gives birth to two diverging critical points. Condition II is violated whenever $f_t$ has two critical points $x \neq y$ with $f_t(x) = f_t(y)$ that form an interchange. The two critical points may be independent and have no effect on the Reeb graph, or they may belong to the same level set component of $f_t$ and correspond to two nodes that swap their positions along the Reeb graph. I now analyze the changes caused by birth-death points and by interchanges in detail.

**Nodes appear and disappear.** When time passes the moment of a birth point, we get two new critical points and correspondingly two new nodes connected by an arc in the Reeb graph. By the Index Lemma, the indices of the two critical points differ by one, leaving three possibilities: 0-1, 1-2, and 2-3. Consider first the 0-1 case in which a minimum and a 1-saddle are born. In the Reeb graph, we get a new degree-1 node that starts an arc ending at a new degree-3 node. In other words, the Reeb graph sprouts a new arc downward from an existing branch; see Figure 4.3. The 2-3 case is upside-down symmetric to the 0-1 case, with the Reeb graph sprouting a new arc upward from an existing branch.

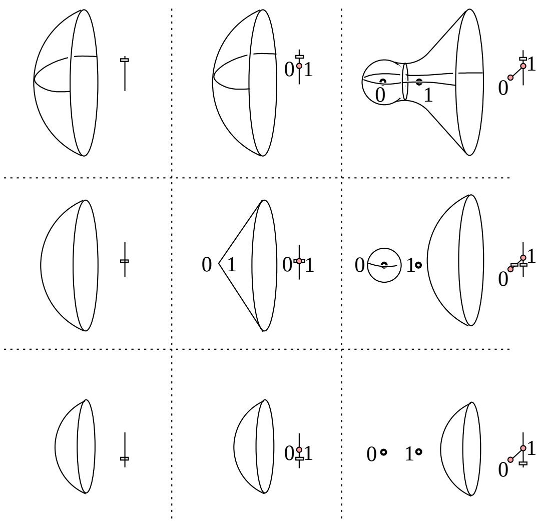


Figure 4.3: Level sets and Reeb graphs around a 0-1 birth point. Time increases from left to right and the level set parameter, indicated by a rectangular slider bar, increases from bottom to top. Going forward in time, we see the sprouting of a bud, while going backward in time we see its retraction.

Consider second the 1-2 case in which a 1-saddle and a 2-saddle are born. In the Reeb graph we get two new degree-2 nodes that effectively refine an arc by decomposing it into three arcs. As illustrated in Figure 4.4, this event corresponds to the appearance of a short-lived handle in the evolution of level sets. Turning the picture upside-down does not change anything, which shows that the case is symmetric to itself. We have

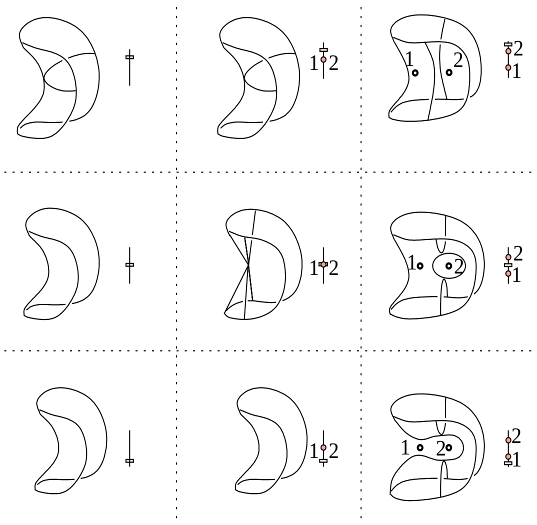


Figure 4.4: Level sets and Reeb graphs around a 1-2 birth point. Time increases from left to right and the level set parameter increases from bottom to top. Going forward in time, we see a refinement of an arc in the Reeb graph and going backward we see a coarsening.

three similar cases when time passes the moment of a death point. Two critical points of $f_t$ converge and annihilate when they collide, and correspondingly an arc of the Reeb graph contracts to a point, effectively removing its two nodes. The 0-1 and 2-3 cases are illustrated in Figure 4.3, which we now read from right to left, and the 1-2 case is illustrated in Figure 4.4, which we also read backward, from right to left.

**Nodes swap.** Nodes of the Reeb graph swap position in the Reeb graph when the corresponding critical points, $x$ and $y$, form an interchange and, at that moment, belong to the same level set component. Assume without loss of generality that $f_{t-\varepsilon}(x) < f_{t-\varepsilon}(y)$ and $f_{t+\varepsilon}(x) > f_{t+\varepsilon}(y)$. We have four choices for each of $x$ and $y$ depending on whether they add or remove a handle, merge two level set components or split a level set component. This gives a total of sixteen configurations. I analyze possible before and after combinations and pair them, giving us the cases illustrated in Figure 4.5. It is convenient to group the cases with similar starting configurations together. I use $+, -, \mathrm{M}, \mathrm{S}$ to denote 'handle addition', 'handle deletion', 'component merge', and 'component split', respectively, for each of $x$ and $y$.

`Case 1` $(++, +-, -+, --)$ Both $x$ and $y$ change the genus and their corresponding

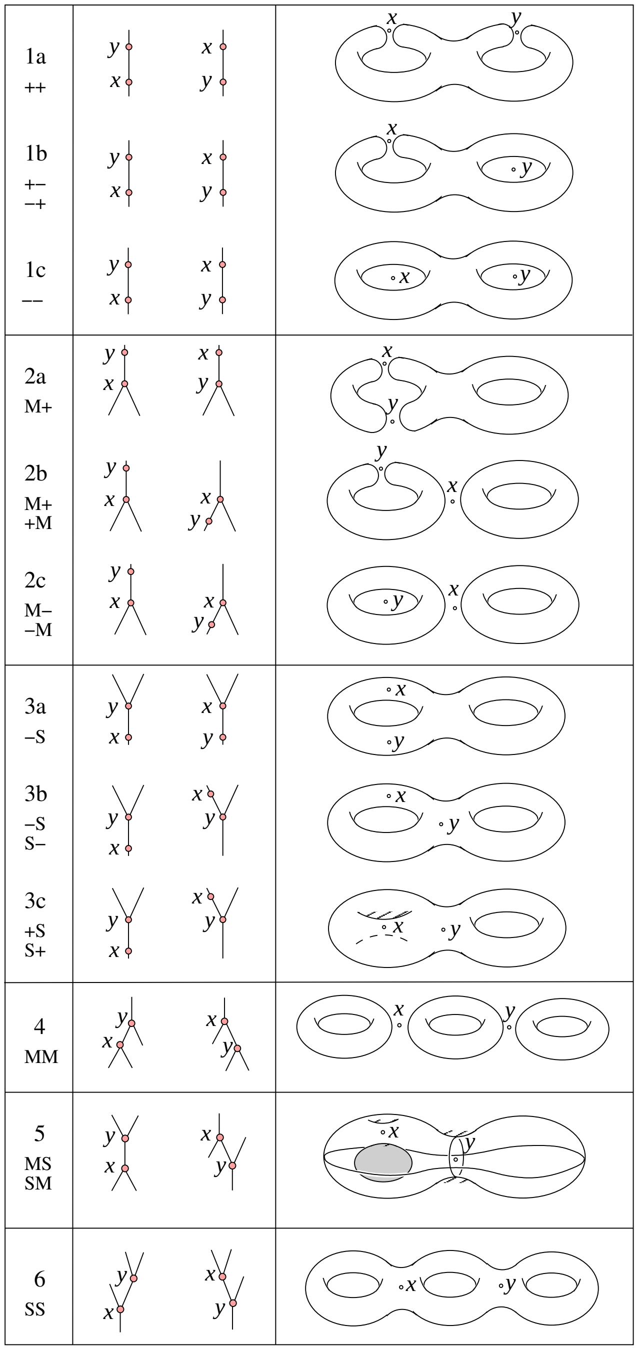


Figure 4.5: On the left, Reeb graph portions before and after the interchange of $x$ and $y$. On the right, level sets at a value just below the function value of $x$ and $y$. In each case, the index of a critical point can be inferred from whether the level set merges (index 1) or splits (index 2) locally at the critical point.

nodes simply swap their positions in the Reeb graph. [I pair ++ with itself to get `Case 1a`, +− with −+ to get `Case 1b`, and −− with itself to get `Case 1c`].

`Case 2` (+M, M+, −M, M−) I consider two sub-cases.

(M+) Before the swap, $x$ merges two components and $y$ adds a handle. Either $y$ involves the two components that were merged by $x$, so $x$ and $y$ just swap, or $y$ involves only one of the two components, so $y$ goes down one of the branches at $x$. [In the first configuration, we pair M+ with itself to get `Case 2a`, and in the second we pair M+ with +M to get `Case 2b`.]

(M−) Before the swap, $x$ merges two components and $y$ removes a handle. After the swap, $y$ moves down one of the branches at $x$. [I pair M− with −M to get `Case 2c`].

`Case 3` (−S, S−, +S, S+) I consider two sub-cases.

(−S) Before the swap, $x$ deletes a handle and $y$ splits the component. Either $y$ breaks a handle and $x$ splits the component into two, so the nodes $x$ and $y$ swap, or $x$ involves only one of the two components split by $y$, so node $x$ goes up one of the branches at node $y$. [In the first configuration, we pair −S with itself to get `Case 3a`, and in the second we pair −S with S− to get `Case 3b`.]

(+S) Before the swap, $x$ adds a handle and $y$ splits the component. After the swap, $x$ moves up one of the branches at $y$. [I pair +S with S+ to get `Case 3c`].

`Case 4` (MM) Three components merge into one, and the only change between before and after is the order of merging. [I pair MM with itself.]

`Case 5` (MS, SM) Before the swap, $x$ merges two components and $y$ splits the merged component. After the swap, $y$ splits one of the components which merge at $x$ before the swap, and $x$ merges one of the split components with the remaining component. [I pair MS with SM.]

`Case 6` (SS) A component splits into three, and the only change between before and after is the order of splitting. [I pair SS with itself.]

The pairing of cases indicates a symmetry between before and after configurations. There is also the symmetry we observe when we exchange inside with outside. Equivalently, we substitute $-f$ for $f$, which turns the Reeb graph upside-down, exchanging minima with maxima and 1-saddles with 2-saddles.

## 4.4 Algorithm for Time-varying Reeb Graphs

In this section, I introduce an algorithm for maintaining a Reeb graph through time. The algorithm is explained at the abstract level without going into implementation details. Section 4.5 will describe an adaptation of the algorithm to the piecewise linear case.

**Data types.** I represent **time** by a conventional priority queue storing *birth-death* and *interchange events* prioritized by the moments in time they occur. At a given moment, $t$, the time data type supports the following operations:

INSERT($e$) : add the future event $e$ (it occurs after time $t$);

NEXTEVENT : return the earliest, top priority event and delete it from the queue;

DELETE($e$) : delete the event $e$ from the queue.

I maintain the **Reeb graph** as a collection of nodes, and arcs that connect the nodes. Each node knows about its incident arcs and about the segment of the Jacobi curve that contains the corresponding critical point. Each arc knows its start-node and end-node and the time when they will die at a death point or swap at an interchange. At a given moment in time, $t$, the Reeb graph data type supports the following operations:

SEGMENT($x$) : return the segment of the Jacobi curve that contains the critical point that corresponds to node $x$;

NODES($a$) : return the start-node and the end-node of arc $a$;

REMOVEARC($a$) : remove the arc $a$ from the Reeb graph;

ADDARC($x, y$) : add an arc connecting the nodes $x$ and $y$ in the Reeb graph.

We have similar operations for removing and adding nodes, which are invoked whenever we remove or add arcs. The **Jacobi curve** is stored as a collection of segments joined at shared birth-death points. Each segment knows its endpoints, the index of its critical point, and the corresponding node in the Reeb graph, if any. Each birth-death point knows its incident segments. At a given moment in time, $t$, the data type for the Jacobi curve supports the following operations:

NODE($\gamma$) : return the node in the Reeb graph that corresponds to the critical point on the segment $\gamma$;

NEXTXING($\gamma, \gamma', t$) : return the next interchange (after time $t$) of the critical points tracing the segments $\gamma$ and $\gamma'$.

Finally, the Jacobi curve data type supports the operation BDPOINTS that returns all birth-death points of the function $f$.

**Sweeping time.** I use the operations provided by the various data types to maintain the Reeb graph of $f_t$ through time. I assume that data is available in a finite range, from time 0 to 1. Starting with the Reeb graph $R_0$ at time $t = 0$, I maintain $R_t$ by sweeping forward in time, using the Jacobi curve as a path for its nodes. The time data type is initialized by inserting all birth-death points provided by BDPOINTS. Interchange events are inserted and deleted as arcs appear and disappear in the Reeb graph. Events are processed in the order they are returned by repeated execution of the NEXTEVENT operation.

`Case` birth event. The type is 0-1, 1-2, or 2-3 and can be determined from the indices of the two segments that meet at the birth point $u$ on the Jacobi curve. Next, I determine the arc $a$ to modify. Arc $a$ contains the representative point of the level set through birth point $u$. I defer the description of how to determine this arc till the next section. Finally, for cases 0-1, and 2-3 I refine $a$ and sprout a bud, and for case 1-2 I refine $a$ by decomposing it into three arcs.

`Case` death event. I retrieve the nodes in the Reeb graph that correspond to the two segments $\gamma$ and $\gamma'$ that share the death point on the Jacobi curve: $x = \text{NODE}(\gamma)$ and $y = \text{NODE}(\gamma')$. Then I contract the arc connecting $x$ and $y$ to a point and finally delete this point by removing three arcs and adding one.

`Case` interchange event. I swap the two nodes $x$ and $y$ that correspond to the critical points of the interchange by removing and adding arcs as indicated in Figure 4.5. Determining which arcs below $x$ or above $y$ to remove is equivalent to deciding between sub-cases of the interchange event. Such a decision is needed in `Cases 2` to `6`; I defer the description of how to make these decisions to the next section.

As mentioned earlier, each arc removal implies the deletion of an interchange event, and each arc addition implies the insertion of one into the time data type. Determining arcs to modify during birth events and interchange events requires global information on level set connectivity. In the next section, I will use a PL mesh to specify the Path operation required to determine these arcs.

## 4.5 PL Implementation

In this section, I describe how to implement the algorithm of Section 4.4 for a piecewise linear function defined on a triangulation $K$ of the 3-sphere cross time.

**Data structures.** I can now describe specific data structures implementing the three abstract data types: time, Reeb graph, and Jacobi curve. Time can be implemented as a standard priority queue, such as a binary heap, and the Reeb graph can be stored in a standard graph representation [AHU74]. The Jacobi can be represented by cyclic lists of edges in the input triangulation, $K$. Each cycle is decomposed into segments of maximal linear lists of edges that are monotonic in time.

Let $K_t$ denote the three-dimensional slice at time $t$ of the four-dimensional triangulation $K$. The vertices of $K_t$ are points on edges of $K$, the edges are slices of triangles, etc. I require the following operation to determine arcs to modify during birth events and to distinguish between the various configurations at an interchange event.

Path($u$) : return a monotone path connecting leaves in the Reeb graph. The path contains the point representing the level set component of $f_t$ passing through the vertex $u$, which is either on the Jacobi curve or in the link of such a point.

To compute this path, I walk in the 1-skeleton of $K_t$, in the direction of increasing $f_t$, until I reach a maximum vertex $x$. Similarly, I walk in the direction of decreasing $f_t$ until I reach another minimum vertex $y$. Observe that $x$ and $y$ are also leaf nodes in the Reeb graph, $R_t$, and delimit the desired path.

**Determining arcs to modify at birth and interchange events.** A Birth event modifies an arc $a$ which I can find on Path($u$), where $u$ is the birth point. Interchange events have various subcases which I can distinguish using the Path operation in the following manner: I consider `Case 2` illustrated in Figure 4.5. I distinguish between `Case 2a` and `2b` using the Path operation; Case 2c can be distinguished using the index of $y$. In $K_t$, points $x$ and $y$ have the same function value and the lower link of $y$ has two components. Letting $u$ and $v$ be a vertex in each, I compute Path($u$) and Path($v$). Next, I compute arc $a$ in Path($u$), and arc $b$ in Path($v$), both arcs incident to and below $x$. We have `Case 2a` iff $a \neq b$. I distinguish between `Cases 2b` and `2c` using the index of $y$ and identify $a = b$ as the arc below $x$ to be refined by $y$ in the new Reeb graph. `Case 4` is similar, except that the only decision to be made is which arc below $x$ gets refined by $y$. `Cases 3` and `6` are upside-down symmetric to `Cases 2` and `4` and are distinguished by calling PATH for vertices in the upper link components of $x$. Finally, `Case 5` is slightly different and decided by calling PATH for $x$ and for $y$. I determine arc $a$ in PATH$(x)$ incident to and above $y$, and arc $b$ in PATH$(y)$ incident to and below $x$, to modify at the interchange.

**Initialization, sweep, and construction.** I begin by constructing the Jacobi curve as a collection of edges in $K$ using the algorithm in [EH02]. This provides the collection of birth-death points, which I use to initialize the priority queue representation of time. I also construct the Reeb graph at time zero from scratch, using the algorithm in [CSA03], which is similar to the forward-backward sweep algorithm for computing Betti numbers in [DE95]. The latter algorithm also detects when 1-cycles are created and destroyed, which is the information I need to add the degree-2 nodes to the Reeb graph, which is not part of the former algorithm. The last step in preparation for the sweep through time inserts the interchange events that correspond to arcs in the Reeb graph into the priority queue. Specifically, for each arc $a$ in $R_0$, I get $(x, y) =$ NODES$(a)$, $\gamma =$ SEGMENT$(x)$, $\gamma' =$ SEGMENT$(y)$ and I insert the interchange returned by NEXTXING$(\gamma, \gamma', 0)$ into the priority queue.

The sweep is now easy, repeatedly retrieving the next event, updating the Reeb graph, and deleting and inserting interchange events as arcs are removed and added. I think of the sequence as the evolution of a single Reeb graph. Following Driscoll et al.[DSST89], I accumulate the changes to form a single data structure representing the entire evolution, which I refer to as the *partially persistent Reeb graph*. I adhere to the general recipe, using a constant number of data fields and pointers per node and arc to store time information and keep track of the changes caused by an update. In addition, I construct an array of access pointers that can be used to retrieve the Reeb graph at any moment in time proportional to its size.

**Analysis.** The running time of the algorithm can be expressed in terms the following parameters

$N =$ number of simplices in $K$, the triangulation of the space-time data;

$n =$ upper bound on the number of simplices in a slice $K_t$ of $K$;

$E =$ number of birth-death and interchange events;

$k =$ number of edges of the Jacobi curve.

We have $n \leq N$, $k \leq N$, and $E \leq k^2$, assuming the triangulation is fine enough to resolve the Jacobi curve as a disjoint collection of simple cycles. For typical input data, the left side will be significantly smaller than the right side in all three inequalities. To construct the Jacobi curve, I compute the reduced Betti numbers of the lower link of each edge in time $O(\ell)$, where $\ell$ is the size of the link. The total size of all links is proportional to $N$, which implies a running time of $O(N)$. The birth-death points are inserted into the priority queue in $O(\log n)$ time each. The initial Reeb graph is constructed in time $O(n \log n)$, inserting the initial batch of interchange events in time $O(n \log n)$ also. The sweep iterates through $E$ events, each in time $O(n)$ needed to determine the after configuration of the event. In addition, we use time $O(k)$ to move the nodes of the Reeb graph along the chains of edges representing the segments of the Jacobi curve. In total, the running time is $O(N + En)$. I construct the partially persistent data structure representing the evolution of the Reeb graph in the same time. The size of that data structure is proportional to the size of the initial Reeb graph plus the number of events, which is $O(n + E)$.

An obvious place to improve the running time is to improve the time needed to do a Path operation. Is there a data structure that can return (the endpoints of a) path in time $O(\log n)$? If so then the total running time would improve to $O(N + E \log n)$.

## 4.6 Conclusion

In this chapter I describe the classification of the combinatorial changes in the evolution of the Reeb graph of a generic time-varying Morse function on $\mathbb{S}^3$. I establish a connection between the time-series of Reeb graphs and the Jacobi curve defined by the time-varying function. Using this connection, I describe an algorithm that maintains the Reeb graph for piecewise linear data. Letting $n$ be the upper bound on the number of simplices in the triangulation of $\mathbb{S}^3$, this algorithm takes time $O(n)$ per combinatorial change in the Reeb graph. While maintaining the Reeb graph, I construct a partially persistent data structure of size proportional to the initial Reeb graph plus the number of events that represent the entire evolution. Given a moment in time, $t$, I can use this data structure to retrieve the Reeb graph $R_t$ in time logarithmic in the number of events plus linear in its size.

Both the case analysis of events and the algorithm are limited to $\mathbb{S}^3$ and to a function on $\mathbb{S}^3$ that varies with time. It would be interesting to extend the analysis and the algorithm to a time-varying function on a general 3-manifold, for which the Reeb graph may have loops. Beyond this extension, it would be interesting to generalize the analysis and the algorithm to a function $f$ restricted to the level sets of another function $g$ defined on the same 4-manifold. Additional problems can be formulated by increasing the number of dimensions and the number of functions.

# Chapter 5

# Computing Level Set Topology and Path Seeds Over Time

Betti numbers of a level set provide information about its topology, and path seeds of a level set allow us to build it. In this chapter I show how to augment the time-varying Reeb graph to encode this information. The topology of level set components of a fixed level value evolves over time. I present an algorithm to compute and encode this evolution in a level graph.

## 5.1 Betti Numbers of Level Sets

The Betti numbers of a space count various topological features. For the level sets of regular values of $f_t$, which are 2-manifolds, $\beta_0$ is the number of connected components, $\beta_1$ is the number of tunnels, and $\beta_2$ is the number of voids. (For a d-dimensional space, only $\beta_0$ through $\beta_d$ may be non-zero.)

There is coherence for the set of Betti numbers along an arc of $R_t$. There is also coherence for how these number change as the Reeb graph moves through time. Because level set components change topology only at critical points, all level set components in the family represented by an open Reeb graph arc (excluding its end-nodes) of $R_t$ have the same set of Betti numbers. If we equip each arc of the time-varying Reeb graph with the set of Betti numbers of its family of level set components, then we can obtain Betti numbers for any level set component at any time. In other words, the Betti numbers can be parametrized by time $t$ and level value $s$ as $\beta_i(s,t)$. The arc of Reeb graph $R_t$ that represents a component of the level set at level value $s$ contains the set of Betti numbers for that component. I will however continue to use the unparameterized notation $\beta_i$ for simplicity.

For a regular value of $f_t$ each level set component is a closed 2-manifold, giving $\beta_0 = \beta_2 = 1$ for each arc; only the value of $\beta_1$ must be computed. Before I compute the value of $\beta_1$ for each arc of the time-varying Reeb graph, I first consider the function $f_t$ and classify the effect of its critical points on level sets, and their $\beta_1$ value. I can then vary $t$ and understand how each birth-death, and interchange event that modifies the Reeb graph changes the $\beta_1$ value of its arcs.

**Critical points change the $\beta_1$ value of level set components.** I study the action of a critical point $x$ on the topology of level set components as I sweep function value $f_t$ past its value from below. This will tell us how the $\beta_1$ value of the arcs incident to and above $x$ are related to the $\beta_1$ value of the arcs incident to and below $x$.

If $x$ is a minimum it creates a new component homeomorphic to $\mathbb{S}^2$; the arc incident to and above it has Betti number $\beta_1 = 0$.

If $x$ has index-1 then it can merge two level set components into one, or add a handle to a single level set component. When two level set components merge, we have the connected sum of two 2-manifolds. We derive the relationship between their Betti numbers before and after the merge.

Connected sum Lemma. The Betti number $\beta_1$ of the connected sum of 2-manifolds $\mathbb{M}$, and $\mathbb{N}$, without boundary, is the sum of their Betti number $\beta_1$: $\beta_1^{\mathbb{M}\#\mathbb{N}} = \beta_1^{\mathbb{M}} + \beta_1^{\mathbb{N}}$.

Proof. The Euler characteristic of the connected sum is $\chi(\mathbb{M}\#\mathbb{N}) = \chi(\mathbb{M}) + \chi(\mathbb{N}) - 2$. Expressing each of the Euler characteristics in terms of Betti numbers, and noting that $\beta_0 = \beta_1 = 1$ for a 2-manifold without boundary yields the formula $\beta_1^{\mathbb{M}\#\mathbb{N}} = \beta_1^{\mathbb{M}} + \beta_1^{\mathbb{N}}$. ◻

If $x$ merges two components then according to the connected sum lemma the $\beta_1$ value of the arc incident to and above it is the sum of the $\beta_1$ value of the arcs incident to and below it.

If $x$ adds a handle we can understand the change in the $\beta_1$ value using the relationship between it and the genus $g$: $\beta_1 = 2g$. Adding a handle increases the value of $\beta_1$ by 2; the $\beta_1$ value of the arc incident to and above $x$ is larger by 2 than the $\beta_1$ value of the arc incident to and below it.

If $x$ is a maximum or an index-2 critical point its action on value of $\beta_1$ is upside down symmetric to that when it is a minimum, or an index-1 critical point respectively.

With this understanding of the action of critical points on the value of $\beta_1$, I investigate how to update it over time at birth-death and interchange events.

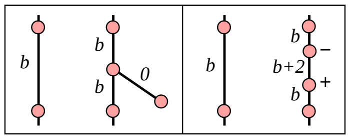


Figure 5.1: Updating the value of $\beta_1$ at a birth-death event. As described in section 4.3, a '+' means handle addition, and a '−' means handle deletion. For each picture time increases to the right. The $\beta_1$ value of arcs at a 0-1 birth event at left, and at a 1-2 birth event at right. A 2-3 birth is upside down symmetric to the 0-1 birth. Running in reverse gives us the corresponding death cases.

**The $\beta_1$ value changes over time.** At a 0-1 birth event, set the $\beta_1$ value of the new arc to 0 according to our classification; the $\beta_1$ value of the other arcs incident to the new index-1 saddle are the same as that of the deleted arc. Handle a 2-3 birth similarly. Figure 5.1 shows $\beta_1$ value at a 0-1 birth on the left; the 2-3 birth is upside down symmetric to it.

At a 1-2 birth event, set the $\beta_1$ value of the new arc to the $\beta_1$ value of the arc incident to and below its index-1 node, and increment it by 2; the $\beta_1$ value of the other arcs incident to the new nodes are the same as that of the deleted arc. Figure 5.1 shows the $\beta_1$ value at a 1-2 birth on the right.

Handle a death event as if running the corresponding birth event in reverse. Set the $\beta_1$ value of the new arc to the $\beta_1$ value of one of the deleted arcs that did not connect the destroyed pair of nodes. Going backwards in time in Figure 5.1 shows the $\beta_1$ value at death events.

We now consider updates during interchange events. We can analyze each interchange case using the action of critical points on the value of $\beta_1$ developed earlier, to yield the updates to value of $\beta_1$ shown in Figure 5.2.

Consider `case 1b` in the Figure 5.2 and observe how the $\beta_1$ value of the level set component changes as we sweep up the level value from below. Before the interchange, node '+' adds a handle increasing the $\beta_1$ value by 2, and node '−' deletes a handle restoring the $\beta_1$ value to the start value. After the interchange, the effects are reversed; node '−' deletes a handle, following which node '+' adds a handle. For a more complicated case, consider `case 2b`. Before the interchange, the merge node first creates a connected sum of two components with their $\beta_1$ values summed, following which node '+' adds a handle. After the interchange, because node '+' slides down one of the arcs

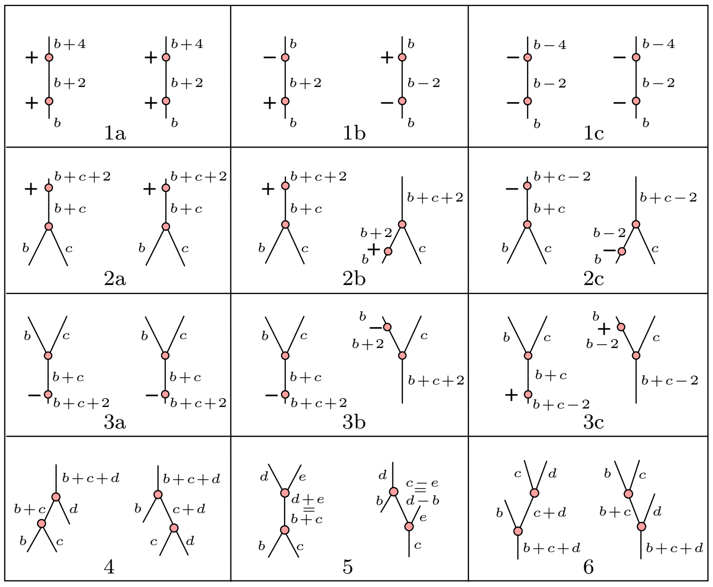


Figure 5.2: Updates to $\beta_1$ value for each interchange case. As described in section 4.3, a '+' means handle addition, and a '$-$' means handle deletion. For clarity, I omit the $x, y$ labels of the interchanging nodes. The letters $b, c, d, e$ denote the $\beta_1$ values of the arcs.

incident to the merge node, it adds a handle to one of the merging components first, followed by the action of the merge node.

We also observe upside-down symmetry in several cases; `case 1a` with `1c`, `case 2` with `3`, and `case 4` with `6`.

We are now ready to compute the $\beta_1$ value for each arc of the time-varying Reeb graph.

**Computing the $\beta_1$ value over time.** We can compute $\beta_1$ value for each arc during the forward sweep in time that constructs the time-varying Reeb graph. Equip each arc of the Reeb graph $R_0$ at time $t = 0$ with its $\beta_1$ value using the algorithm by Pascucci et al. [PCM03]. Maintain $\beta_1$ value for each arc of $R_t$ during every birth-death and interchange event of the sweep according to the update policy developed before.

## 5.2 Path Seeds over Time

Path seeds, developed by Carr et al. [CS03], can be used for fast level set extraction from a static dataset; I wish to compute path seeds for fast level set extraction from time-varying data.

### 5.2.1 Static Path Seeds

We will define static path seeds and compute them; our description will enable us to define and compute time-varying path seeds.

**Definition 5.2.1 (Descending path)** A *descending path* beginning at vertex $u$ of the slice $K_t$ is an edge connected sequence of vertices with decreasing value of $f_t$.

For the next definition, recall that each node $x$ of the Reeb graph $R_t$ maps to a critical point of $f_t$, which I also call $x$.

**Definition 5.2.2 (Path seed)** A *path seed* of an arc of $R_t$ with upper node $x$ is a vertex $v \in K_t$ such that, every descending path which begins at $x$ and contains $u$ intersects all the level set components that the arc represents.

This definition does not suggest either how such a vertex $u$ may be found, or how such a descending path that contains both $x$ and $u$ may be found. As will be explained soon, vertex $u$ turns out to be either $x$ itself or a vertex in its lower link; finding a descending path through $x$ and $u$ becomes straightforward.

**Definition 5.2.3 (Seed edge)** A *seed edge* of a level set component is an edge in $K_t$ that intersects it.

If each arc of the Reeb graph is equipped with a path seed, we can find a seed edge for any level set component it represents in any descending path which begins at its upper vertex $x$ and contains its path seed. This level set component can be extracted by visiting only those simplices that intersect it, starting from the simplices incident to the seed edge.

A path seed maps an individual arc to its level sets' component. In their flexible isosurface interface, Carr et al. use this mapping to selectively extract user specified components for one, or several level values. Thus path seeds have an advantage over conventional seed set methods [vKvOB+97] as the latter do not maintain the required mapping.

**Choosing path seeds.** I will describe how to choose a path seed for an arc $a$ of a static Reeb graph depending on the type of the arc's upper node $x$.

`Case` S1: If $x$ is a maximum, a split node, a '+' node, or a '$-$' node, it has one arc incident to and below it; choose vertex $x$ itself as the path seed for arc $a$ because all descending paths that begin at $x$ will intersect all level set components that arc $a$ represents.

`Case` S2: If $x$ is a merge node, it has two arcs incident to and below it, and vertex $x$ has two lower link components. A descending path that starts at $x$ and contains a vertex $u$ of one of its lower link components intersects all the level set components represented by one of the two arcs; a descending path from $x$ that contains a vertex of the other lower link component intersects all the level set components of the remaining arc; match a descending path to its arc and use the lower link vertex in that path as the arc's seed path. Thus path seeds also map each lower link component of $x$ to its two arcs below.

We can choose path seeds in case S2 using the Path operation described in Section 4.5 and restated here.

Path($u$) : return a monotone path connecting leaves in the Reeb graph. The path contains the point representing the level set component of $f_t$ passing through the vertex $u$, which is either on the Jacobi curve or in the link of such a point.

Letting $u$ and $v$ be a vertex in each lower link component of $x$, compute the arcs $a$ and $b$ incident to and below $x$ in Path($u$) and Path($v$); choose $u$ as the path seed for $a$, and $v$ as the path seed for $b$.

### 5.2.2 Time-varying Path Seeds

We are ready to extend the definition of path seeds to time-varying Reeb graphs. Define the *lifespan* of an arc in the time-varying Reeb graph as the time interval between its creation and its deletion.

**Definition 5.2.4 (Time-varying Path Seeds)** A *time-varying path seed* of an arc $a$ of the time-varying Reeb graph is an edge in the triangulation $K$ such that it contains a path seed for arc $a$ in every Reeb graph $R_t$ in its lifespan.

If we equip each arc of the time-varying Reeb graph with a time-varying path seed then, we can find a seed edge for any level set component at any time. But first, we must choose the time-varying path seed of an arc $a$ depending on its upper node $x$ just like we did in the static case; I will prove a lemma on the link structure of a vertex in the slice $K_t$ to assist us in our choice. This analysis will enable us to choose the time-varying path seed of new arcs created at birth-death and interchange events, and we can compute time-varying path seeds for each arc of the time-varying Reeb graph.

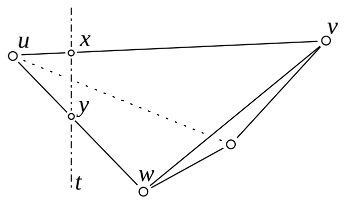


Figure 5.3: In slice $K_t$, vertex $y$ belongs to the link of $x$.

**Choosing time-varying path seeds.** For the slice $K_t$ shown in Figure 5.3, I analyze the condition under which vertex $y$ belongs to the lower link of vertex $x$. In the proof, recall that the function $g$ represents the time.

Lower Link Lemma. Let vertices $x$ and $y \in K_t$ lie on edges $uv$ and $uw \in K$ respectively. Vertex $y$ is in the lower link of critical vertex $x$ whenever vertex $w$ is in the lower link of Jacobi edge $uv$.

Proof. Vertex $y$ is in the lower link of $x$ if and only if the following condition holds.

$$\frac{f(w) - f(u)}{g(w) - g(u)} < \frac{f(v) - f(u)}{g(v) - g(u)}$$

Letting $\lambda = (f(u) - f(v))/(g(v) - g(u))$, this reduces to $f(w) + \lambda g(w) < f(u) + \lambda g(u)$. This is exactly the condition for $w$ to be in the lower link of $uv$ when it is a Jacobi edge [EH02]. $\boxed{}$

As described in the analysis next, this lemma will help us choose time-varying path seeds for merge nodes.

`Case` T1: If $x$ is a maximum node, a split node, a '+' node, or a '−' node, it has one arc incident to and below it for its lifespan; choose the Jacobi edge traced by $x$ itself as the time-varying path seed for $a$ because taking slices in its lifespan gives us case S1 discussed before.

`Case` T2: If $x$ is a merge node, we can use the lower link lemma to choose time-varying path seeds for arcs incident to and below it. For the slice $K_t$ shown in Figure 5.3, one vertex $y$ from each lower link component of $x$ can be a path seed for each of its arcs in $R_t$ according to case S2. The lower link lemma implies that vertex $y$ is in the lower link of $x$ for the time interval $(g(u), g(w))$ spanned by edge $uw$. From each lower link component of $uv$ choose a vertex $w$ and match edge $uw$ to the arc incident to and below $x$ if vertex $y$ is a path seed for that arc in any slice $K_t$ in this time interval. (This matching can be computed as explained in the static case using the PATH operation). Edge $wv$ is the time-varying path seed for that arc for the time interval $[g(w), g(v))$

We can choose time-varying path seeds for each new arc created at a birth-death and interchange event; simply determine the case based on the upper node of the new arc and choose as described.

**Computing path seeds for the time-varying Reeb graph.** We can now describe how to compute time-varying path seeds for every arc of the time-varying Reeb graph. The computation is done during the forward sweep in time that constructs the time-varying Reeb graph. Equip each arc of the Reeb graph $R_0$ with a time-varying path seed according to case T1, or T2. Update the time-varying path seeds of new arcs created at each birth-death and interchange event as described before.

## 5.3 The Level Graph

The level set components of a fixed level value merge, split, gain, or lose a handle over time. The level graph, also called *topology change graph* [SB05], captures this evolving topology and can assist in analyzing the time-varying function. We will connect the level graph to the Jacobi set and time-varying Reeb graphs, and develop an algorithm to compute it.

At a fixed time, the *level graph* of a chosen level value $s$ contains one point for each level set component of $s$. These points sweep out the arcs of the level graph as we keep the value of $s$ fixed and vary time; an arc ends at a node when its level set component disappears, an arc begins at a node when its level set component appears, two arcs join at a node when their level set components merge, and an arc bifurcates at a node when its level set component splits into two. It is possible to define the level graph as the Reeb graph of a function, and to connect it to the Jacobi set and the time-varying Reeb graphs.

### 5.3.1 Jacobi Sets Connect Level Graphs

We use Morse functions to define the level graph, as in the definition of the Reeb graph. Let $f, g\colon \mathbb{M} \times \mathbb{R} \to \mathbb{R}$ be two Morse functions, and let function $g$ represent time. For a regular value $s \in \mathbb{R}$, we have the level set $f^{-1}(s)$ and the restriction of $g$ to this level set, $g_s\colon f^{-1}(s) \to \mathbb{R}$. The *level graph* of level value $s$ is the Reeb graph of $g_s$; I denote it as $L_s$.

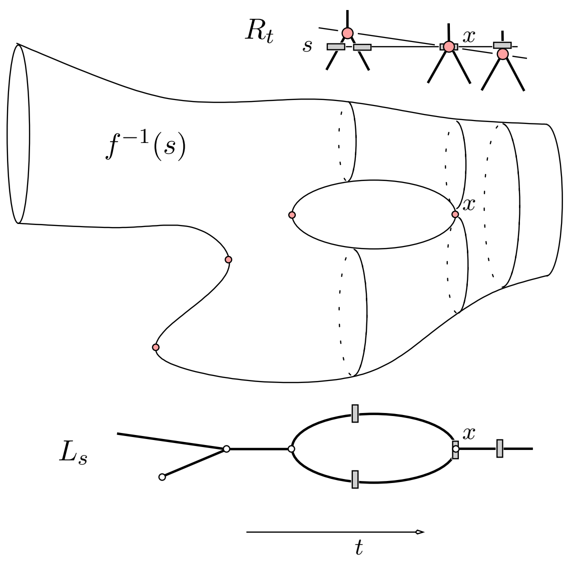


Figure 5.4: The level graph, and its connection to time-varying Reeb graphs. Top, Reeb graphs of $f$ restricted to $g^{-1}(t)$ at three instants of time with a Jacobi edge connecting the nodes; middle, the level surface $f^{-1}(s)$; bottom, the level graph which is the Reeb graph of time(function $g$) restricted to this surface. The sliders on the level graph and the time-varying Reeb graphs correspond to the level curves shown on the surface. These curves are the set of points at time $t$ with function value $s$.

In Figure 5.4, we see a portion of the level set $f^{-1}(s)$ for functions defined on $\mathbb{S}^2 \times \mathbb{R}$, and the level graph, $L_s$ below it. On top, we see Reeb graphs at three instants of time, with their nodes connected by a Jacobi edge. I will connect the level graph to the Jacobi set and the time-varying Reeb graph using this figure as a guide. To avoid confusion with the notation for the Reeb graph, I will call the nodes of the level graph *knots*, and its arcs *threads*. The knots of $L_s$ correspond to critical points of function $g_s$. These critical points sweep out the Jacobi set as the value of $s$ changes continuously. Thus, the Jacobi set connects the level graphs over changing level value $s$ into a 1-parameter family, just like it connects the Reeb graphs over changing time. In Figure 5.4, knot $x$ of $L_s$ maps to critical point $x$ of function $g_s$ defined on the level surface in the middle.

Shown on top, critical point $x$ is a point on a Jacobi edge where it has function value $f = s$.

The relationship between $L_s$ and time-varying Reeb graphs is that the level sets of $f_t$ at level value $s$ are the level sets of $g_s$ at time $t$. In Figure 5.4, the sliders on the level graph and the Reeb graphs map to the level curves on the surface, which provide a bijection between the 1-parameter family of level graphs, and the 1-parameter family of Reeb graphs. Each point at time value $t$ on $L_s$ maps to a distinct level set component in space-time, which maps to the point for level value $s$ on $R_t$. This mapping suggests an algorithm to compute $L_s$ from the time-varying Reeb graph; track the points for $s$ on $R_t$ over time.

### 5.3.2 Computing the Level Graph

We first classify how the level set components of $s$ evolve over time, and the corresponding action on $L_s$.

**Level sets appear or disappear.** A level set component appears if a minimum node drops below value $s$, or if a maximum node rises above value $s$. In both cases, the arc adjacent to the node contains a point at value $s$ after the change in order; add a knot to $L_s$ and start a thread from it. The left of Figure 5.5 shows the first case; the second case is upside down symmetric to it. Running the example in reverse shows how level set components disappear. When a component disappears, end its thread at a knot.

**Level sets split or merge.** Level set components split or merge only if their arc is incident on a merge or split node of $R_t$, and if such a node changes order with $s$. A level set component splits into two if the upper split node of its arc, drops below $s$, or the lower merge node rises above $s$; its point on $R_t$ splits into two; end its thread at a knot and start two new threads from this knot. The middle of Figure 5.5 shows the first case; the second case is upside down symmetric to it. Running the example in reverse shows how two level set components merge into one. When components merge, join their threads at a knot and start a new thread from this knot.

**Level sets gain or lose a handle.** Level set components change genus only if their arc is incident on a '+' (index-1) node, or '−' (index-2) node of $R_t$, and if such a node changes order with $s$. A level set component gains a handle if the upper '+' node of its arc drops below $s$, or the lower '$-$' node of its arc rises above $s$; end its thread at a knot and start a new thread from this knot. The right of Figure 5.5 shows the first case; the second case is upside down symmetric to it. Running the example in reverse shows how a level set loses a handle. Modify $L_s$ similar to when the component gains a handle.

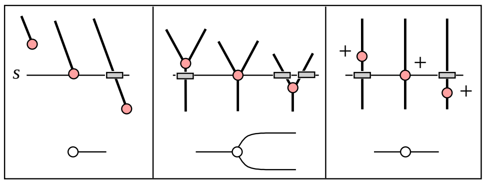


Figure 5.5: Reeb graphs indicate change in level set topology. In each picture, time increases to the right; on top, a portion of the Reeb graph before, at, and after the change; at bottom, the action on the level graph. Left, a new level set component appears when a minimum drops below $s$. Middle, a level set component splits into two when a split node drops below $s$. Right, a level set component gains a handle when a '+' node drops below $s$. Running each case in reverse shows how level sets disappear, merge, and lose a handle, respectively.

In all cases, the topology of a level set component changes at the time that the Jacobi edge traced by one of its end-nodes changes order with $s$. If we can compute this time, we will have all the pieces in place to compute the level graph. For an arc $a$ in $R_t$ let its level set component change topology at time $t' > t$. The end-nodes of arc $a$ can change in between time $t$ and $t'$ if another node incident to them interchanges in that time. In Figure 5.6, the upper node $x$ of arc $a$ interchanges with node $y$ before the level set changes topology. Compute time $t'$ as follows. For arc $a$ at the current time $t$, maintain the *transition time* which is the minimum of the next time (after $t$) at which any of its nodes interchanges and the next time when the Jacobi edge of any of its nodes changes order with $s$. As long as the next transition is an interchange. update the end-nodes of $a$ as described in Section 4.4. When updating the end-nodes compute the new transition time. The required time $t'$ is the transition time when we stop.

We are now ready to compute the level graph. The algorithm for computing the level graph takes as input the time-varying Reeb graph stored in a partially persistent data-structure, and a level value $s$, and produces as output the knots and threads of its level graph $L_s$, which can be represented as a conventional graph.

The algorithm proceeds by sweeping forward in time while maintaining a list $A$ of arcs of $R_t$ containing level value $s$. Each arc represents its point at value $s$, and each

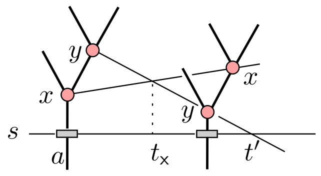


Figure 5.6: Finding topology change time. Time increases to the right and Reeb graphs are shown in bold. The Jacobi edges traced by node $x$ and $y$ change order at time $t_\times$. To find topology change time for arc $a$ of $R_t$ update its end-nodes at time $t_\times$, following which compute the time when the Jacobi edge traced by node $y$ changes order with $s$.

such point traces a thread in the level graph over the sweep. The classification of level set evolution tells us how and when arcs enter and leave the list; for each case add arcs that contain point $s$ to $A$, and delete those that do not. Store an event which identifies the arcs that enter and leave $A$ for each such update in a priority queue $Q$ prioritized by the time when it occurs.

**Initialization, sweep, and construction of $L_s$.** Initialize $Q$ with events for new level set components when Jacobi edges traced by maxima or minima change order with $s$. Initialize $A$ with those arcs of $R_0$ that contain value $s$. Start a new thread for each such arc, and equip the arc with a pointer to it. For each arc $a$ in $A$ find the time when its level set component change topology as described before, and store an event for that time in $Q$.

To implement the sweep, repeatedly retrieve the next event, update arc list $A$, and modify $L_s$ as specified in the classification of level set evolution. When adding a new arc to $A$ insert events for when its level set changes topology into $Q$, and equip it with a pointer to its thread. Finally, if each Reeb graph arc stores the Betti numbers of its level set, then one can also label its corresponding thread in level graph $L_s$ with this information.

**Analysis.** Let $k$ be the number of Jacobi edges, and $h$ be the size of the level graph $L_s$. Note that the number of arcs of the time-varying Reeb graph containing value $s$ at any time is bounded from above by $h$.

Initialization takes $O(h + \log k)$ time if the arcs of Reeb graph $R_0$, and the Jacobi edges are organized in an interval tree, based on their interval of values for function $f$. Because each event modifies the evolving $L_s$ by adding a node, and a constant number of threads, the total number of events is $O(h)$, and processing them in prioritized in time takes $O(h \log h)$ time. Each event takes constant time to delete and insert arcs into $A$, and to update $L_s$, and a variable, yet to be determined time to find the topology change event for each arc inserted into $A$. We bound this variable time over the entire sweep.

As described earlier, to compute the time at which the level set component of an arc $a$ in $R_t$ changes topology we must process all interchanges of the end-nodes of $a$ until $t'$. We can process these interchanges in constant time each using the input time-varying Reeb graph. Because each interchange is processed atmost a constant number of times over the entire sweep, the total time to find events for arcs inserted into $A$ is bounded from above by the by the total number of interchanges that change the end-nodes of these arcs. We can bound this number using the zone theorem theorem for line segments [AS95], which states that the complexity of the region of an arrangement of $k$ line segments in the plane that touches a specified line is $O(k\alpha(k))$, where $\alpha$ is the extremely slow growing inverse of Ackerman's function. See Figure 5.7.

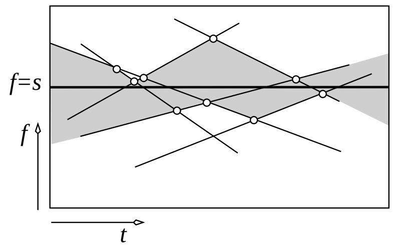


Figure 5.7: Jacobi edges projected to the plane spanned by function value $f$, and time. The zone of $f = s$ is shown shaded, and the circles correspond to interchanges that change the end-nodes of an arc before its update event.

Consider the two dimensional plane spanned by time on the x-axis, and function value $f$ on the y-axis. Project each Jacobi edge onto this plane. The total number of interchanges that must be visited to find valid events over the entire sweep is bounded by the number of vertices in the zone of $f = s$, which is bounded by $O(k\alpha(k))$. We can now state the running time of the sweep.

**Theorem 5.3.1 (Level Graph Theorem)** The level graph $L_s$ can be constructed in time $O(k\alpha(k) + h \log h)$.

## 5.4 Conclusion

In this chapter, I provide algorithms to augment the time-varying Reeb graphs with the Betti numbers of the level sets of all level values at all times, with the path seeds to accelerate the extraction of level sets of all level values at all times. I define the level graph $L_s$, which encodes the topology of the level set components of a given level value $s$ over time. I present an algorithm to compute the level graph using time-varying Reeb graphs. This algorithm is suited for repeatedly constructing $L_s$ for given $s$, from an already constructed time-varying Reeb graph. It would be interesting to construct time-varying level graphs using the Jacobi curve as a guide, just as in the construction of time-varying Reeb graphs, in future work.

The level graph is similar to the *topology change graph (TCG)* defined by Sohn et al. [SB05]. They construct the TCG from the correspondence mapping between arcs of Reeb graphs at these discrete time steps, as described in Section 3.2.2. This implies that the TCG cannot represent any change in topology that occurs in between discrete time steps, unlike the level graph.

# Chapter 6

# Jacobi Sets of Time-varying Interpolants on the Two-sphere

Jacobi sets assist in the analysis of time-varying scientific datasets because they trace the path followed by critical points, connect Reeb graphs over time, and can be used to compute time-varying Reeb graphs, as discussed in Chapter 4.

The algorithm that computes time-varying Reeb graphs requires that the Jacobi set be a 1-manifold, or can be unfolded into a 1-manifold, and that all events that modify the Reeb graph, which are computed from the Jacobi set, occur at distinct times. Scientific datasets which contain finite set of sample points in space-time are extended over the whole domain by an interpolant which is continuous, but often not generically smooth (non-Morse), and its Jacobi set is often not a 1-manifold.

In this chapter, I describe algorithms to compute the Jacobi sets of three time-varying interpolants defined on $\mathbb{S}^2$. I study how well their Jacobi sets approximate the structure of a 1-manifold, and if they produce events that modify the Reeb graph at distinct times to decide which interpolant is best suited for use in computing time-varying Reeb graphs.

## 6.1 Introduction

Many scientific datasets capture scalar values at a finite set of sample points in a space-time domain. An interpolation scheme extends these values to an *interpolant*: a function over the whole domain that agrees with the values at the sample points. Mesh-based interpolation schemes connect the sample points into a mesh, and define the interpolant over each mesh element.

The characteristics of the sampling often suggest natural interpolants. Data points sampled irregularly in space and time suggest to form a simplicial mesh and use piecewise linear (PL) interpolation. Data sampled at regular time intervals from fixed but irregular points in space suggest to use a PL interpolant, or to fix a simplicial mesh in space, extrude each simplex into a prism between time samples, and define a bilinear interpolant (Prism). Data sampled on a regular space time grid suggests PL, Prism, or trilinear interpolation (Tri). I will define these interpolants precisely in Section 6.2, and give algorithms to compute their Jacobi sets in Section 6.4.

Because each interpolant will produce a Jacobi set with different structural properties, I need a criterion for choosing the most suitable interpolant for computing time-varying Reeb graphs. Recall that the classification of birth, death, and interchange events that change the Reeb graph, and the algorithm to compute the time-varying Reeb graph discussed in Chapter 4 assume that the Jacobi set is a 1-manifold, and that these events occur at distinct times. I produce a wish-list of properties of the Jacobi set that will help me avoid complicated special case analysis to give a simple implementation of this algorithm, and compare each interpolant based on these.

[Manifold] The Jacobi set must be a 1-manifold, or can be unfolded into a 1-manifold.

[Distinct times] Birth, death, and interchange events must occur at distinct times.

[Few events] The number of Jacobi edges $k$ must be small to reduce the number of interchange events, which can be quadratic in $k$.

As we will see, the properties of the Jacobi sets, and the experiments with synthetic and simulation data, described in Chapter 6.5, suggest that Prism is a better choice than PL, and Tri for computing Jacobi sets and time-varying Reeb graphs.

## 6.2 Interpolants

We use interpolants to extend discrete samples of a function to all points in space-time. In this section, I describe PL, Tri, and Prism interpolants.

I introduce notation for each type of mesh decomposition for which we will define interpolants. Each mesh decomposition connects samples in $\mathbb{S}^2 \times \mathbb{R}$, where the last dimension is time; the samples lie at regular intervals of time. Let $K$ be a simplicial decomposition, $C$ denote a cubical decomposition, and $P$ denote a prismatic decomposition (to be defined).

### 6.2.1 Piecewise Linear Interpolant (PL)

A piecewise linear interpolant can be defined on the values of the vertices of a simplex $\sigma \in K$ as follows. Each point $x \in \sigma$ is a unique convex combination of its vertices $u_i$: $x = \sum_i w_i u_i$, where $\sum_i w_i = 1$, and $w_i \geq 0$ for all $i$. The function value at $x$ is the convex combination of the function values at the vertices of $\sigma$: $f(x) = \sum_i w_i f(u_i)$.

The critical points of a PL interpolant always lie on the vertices of $K$, and can be classified using the reduced Betti numbers of the lower link of a vertex as described in Section 2.4.

### 6.2.2 Piecewise Trilinear Interpolant (Tri)

A piecewise trilinear interpolant can be defined on the values of the vertices of a cube in terms of the linear interpolant which is defined for two values $f(0)$ and$f(1)$ and for $0 \leq \alpha \leq 1$ as $f(\alpha) = f(0) + \alpha(f(1) - f(0))$. The trilinear interpolant for a cube is defined recursively, as the linear interpolation of the two bilinearly interpolated values on two opposite faces of the cube. See Figure 6.1.

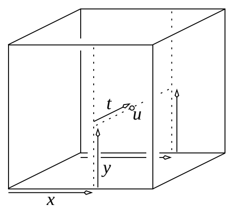


Figure 6.1: Trilinear interpolation inside a unit cube. The function value at $u$ is computed by linearly interpolating values at points on the front and back faces of the cube, using time as the interpolation parameter. The value on each face is computed recursively by bilinear interpolation.

Because we wish to compute the Jacobi set of a time-varying interpolant it is convenient to think of Tri as a time-varying piecewise bilinear interpolant; imagine restricting Tri to a slice orthogonal to the time axis to produce a piecewise bilinear interpolant in that slice, and move this slice through time to make it time-varying.

We must classify critical points of the piecewise bilinear interpolant before we can compute the Jacobi set, which is the path these critical points trace over time. We can classify critical points by comparing the function value of vertices to that of the other vertices connected to it.

We direct edges towards increasing function value, and classify vertices by examining adjacent edges in counter-clockwise order starting from the left. Bilinear interpolants can have critical points in the interior of a cell, unlike PL interpolants; we can detect if a cell contains such a point by examining the direction of its edges. Maxima and

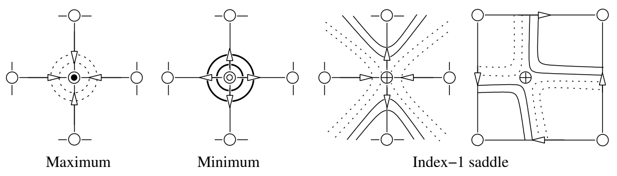


Figure 6.2: Critical points of a bilinear interpolant. The first three pictures show a maximum, a minimum, and a saddle on a vertex, respectively. The last picture shows a saddle that lies inside a cell. The arrows on the edges point in the direction of increasing function value. Only the edges connected to a vertex determine if it is critical; the other edges of the cells adjacent to the vertex are not shown in the first three pictures. Level sets, below the critical value are dotted, and above are bold.

minima lie on a vertex of the grid, and index-1 saddles can lie on a vertex or the interior of a cell, as shown in Figure 6.2. A maximum has all edges pointing into the vertex, a minimum has all edges pointing out of the vertex, a vertex saddle has edges alternating in direction. A cell contains an interior saddle if its edge directions alternate when examined in counter-clockwise order starting from the left. Letting $f_{00}, f_{01}, f_{11}, f_{10}$ be the function values at vertices of a unit cell containing an interior saddle, and $\delta = f_{00} - f_{01} + f_{11} - f_{10}$, the coordinates of the interior saddle, in terms of a local coordinate system based at the origin $(0, 0)$, can be shown to be

$$\left(\frac{f_{00} - f_{01}}{\delta}, \frac{f_{00} - f_{10}}{\delta}\right) \tag{6.1}$$

### 6.2.3 Prismatic Interpolant (Prism)

A prismatic interpolant is defined on a prismatic decomposition of space-time, which can be used when the same spatial sampling is used at regular time steps. Fix a simplicial mesh in space, extrude each simplex into a prism between time samples, and define a bilinear interpolant (Prism). For $d = 2$ this produces a triangle-based prismatic decomposition, shown in Figure 6.3.

A continuous prismatic interpolant can be defined on each prism as follows. At any point $p$ in the prism, a plane passing through $p$ and parallel to the base intersects the prism in a triangle. The function value at each vertex of this triangle is linearly interpolated along the edge of the prism on which the vertex lies, and the value at point $p$ is interpolated using its barycentric coordinates (in the triangle) from the values at the vertices of the triangle.

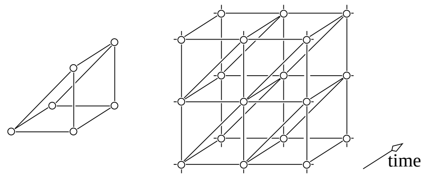


Figure 6.3: On the left, a triangle-based prism. On the right, the prismatic decomposition of regularly sampled space-time. We triangulate vertices within a time-step about the main diagonal and connect corresponding vertices at successive time-steps.

All critical points of the restriction $f_t$ lie on the vertices of $P_t$ because the slice $P_t$ is a simplicial complex. We can classify a vertex of $P_t$ into a regular or critical point using reduced Betti numbers as in the piecewise linear case, and present this classification in Figure 6.4 for completeness.

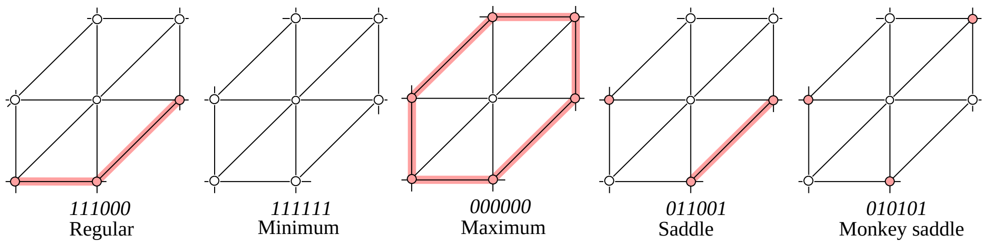


Figure 6.4: Vertex link in a slice of the prismatic decomposition. The link polarity is written as a binary string going clockwise from the left vertex. Lower link vertices are filled and represented by a 0, upper link vertices are unfilled and represented by a 1.

Because the critical points of the piecewise bilinear interpolant restricted to time $t$ can be determined from the direction of increasing function value on the edges incident to a vertex or a cell in the slice $C_t$, one can compute the path traced by the critical points if one can maintain the edge directions over time. Similarly, because the critical points of the piecewise prismatic interpolant restricted to time $t$ can be determined from the lower link of a vertex in the slice $P_t$, one can compute the path traced by the critical points if one can maintain the lower link over time.

In the next section, I will define interchange points that tell us when edges in $C_t$ change direction, and when vertices in $P_t$ enter or leave the lower link of another vertex. The definition uses the *function-time plane*, which is the plane spanned by time on the $x$-axis, and function $f$ on the $y$-axis. The description is valid for any of the three interpolants; I will use $K$ to denote the decomposition of space $\mathbb{S}^2 \times \mathbb{R}$.

## 6.3 Interchange points

Recall that the vertices in the decomposition lie at regular intervals of time. Consider two distinct edges in $K$, with no common vertex, connecting vertices in $t$ with vertices in $t+1$. An *interchange point* is the intersection (if any) of the projection of these edges onto the function-time plane. If an intersection exists we say the two edges interchange at that point. The time coordinate of the interchange point is the instant at which a point on each edge has the same function value as the function coordinate of the interchange point. See Figure 6.5.

Because the time coordinate of an interchange tells us the time when an edge in $C_t$ changes direction, or when a vertex in $P_t$ enters or leaves the lower link of another vertex, I would like each of these changes to occur at distinct times so that each change is simple; one edge flipping direction in $C_t$ or, one vertex entering or leaving a lower link in $P_t$. As we will see, some of these changes correspond to an instant when a critical point pair is born, dies, or jumps from one edge of the decomposition to another edge.

In degenerate cases, two interchange points, which are defined by different pairs of edges, may have identical time coordinates, but we can achieve this with some assumptions and a perturbation scheme.

**Assumptions on vertex indices.** We assign a unique positive integer index to each vertex of $K$, such that the index of any vertex at time $t$ is smaller than the index of any vertex at time $t+1$.

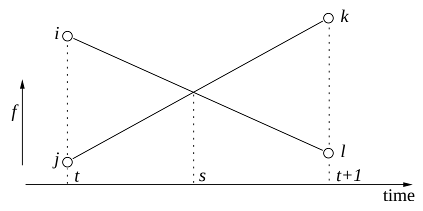


Figure 6.5: The projections of two line segments onto the function-time plane intersect at an interchange.

We use the following notation for the indices of the interchanging segments as shown in Figure 6.5: $i$ is the top-left vertex, $j$ the bottom-left vertex, $k$ the top-right vertex, and $l$ the bottom-right vertex. By construction the indices have the following properties.

I1: The indices $i, j, k, l$ are distinct.

I2: Each of $\{i, j\}$ is smaller than each of $\{k, l\}$.

**Perturbing function values.** Letting $0 < \epsilon < 1$, we perturb the function value at any vertex with index $i$, as $\hat{f}_i = f_i + \epsilon^{2^i}$.

Interchange Lemma Function values at the vertices of two interchanging segments which satisfy conditions I1 and I2 can be perturbed so that the interchange point has a distinct time coordinate.

We need some setup before we can prove this lemma. The time of interchange for the configuration shown in Figure 6.5 is given by the formula

$$\mathsf{t_x} = t + \frac{\hat{f}_i - \hat{f}_j}{(\hat{f}_i - \hat{f}_j) - (\hat{f}_l - \hat{f}_k)} \tag{6.2}$$

Consider another set of two interchanging segments, with primed indices for the corresponding vertices. We first develop an expression for the sign discriminant of $\mathsf{t_x} - \mathsf{t_x}'$, using $A = f_i - f_j,\ B = f_l - f_k$.

$$\begin{aligned}\Delta &= (A + \epsilon^{2^i} - \epsilon^{2^j})(A' - B' + \epsilon^{2^{i'}} - \epsilon^{2^{j'}} - (\epsilon^{2^{l'}} - \epsilon^{2^{k'}})) - \\ &\quad (A' + \epsilon^{2^{i'}} - \epsilon^{2^{j'}})(A - B + \epsilon^{2^i} - \epsilon^{2^j} - (\epsilon^{2^l} - \epsilon^{2^k}))\end{aligned}$$

Note that $\Delta = 0$ whenever $\mathsf{t_x} - \mathsf{t_x}' = 0$. Simplifying this expression gives us $\Delta = T_1 + T_2 + T_3$, where

$$\begin{aligned}T_1 &= A'B - AB' \\ T_2 &= A'(\epsilon^{2^l} - \epsilon^{2^k}) + B(\epsilon^{2^{i'}} - \epsilon^{2^{j'}}) \\ &\quad -A(\epsilon^{2^{l'}} - \epsilon^{2^{k'}}) - B'(\epsilon^{2^i} - \epsilon^{2^j}) \\ T_3 &= (\epsilon^{2^l} - \epsilon^{2^k})(\epsilon^{2^{i'}} - \epsilon^{2^{j'}}) - (\epsilon^{2^{l'}} - \epsilon^{2^{k'}})(\epsilon^{2^i} - \epsilon^{2^j})\end{aligned}$$

We are now ready to prove the Interchange Lemma. We show that if $\Delta = 0$ then $i = i', j = j', k = k'$, and $l = l'$.

Interchange Lemma Proof.

If $\Delta = 0$, then each of $T_1, T_2, T_3$ is identically zero.

$T_1 = 0$ implies $A'B = AB'$. We consider possible cases

Case 1: Values $A, B, A', B'$ all non-zero.

Simplifying $T_2 = 0$, we get $B\epsilon^{2^{i'}} + B'\epsilon^{2^j} + A\epsilon^{2^{k'}} + A'\epsilon^{2^l} = B'\epsilon^{2^i} + B\epsilon^{2^{j'}} + A'\epsilon^{2^k} + A\epsilon^{2^{l'}}$. By properties I1 and I2, term $T_2 = 0$ only if $A = A',\ B = B',\ i = i',\ j = j',\ k = k'$, and $l = l'$.

Case 2: Values $A, A' = 0$ and $B, B' \neq 0$

$T_2 = 0$ implies $B\epsilon^{2^{i'}} + B'\epsilon^{2^j} = B'\epsilon^{2^i} + B\epsilon^{2^{j'}}$,

giving us $B = B'$, $i = i'$ and $j = j'$.

This simplifies $T_3 = 0$ to $\epsilon^{2^l} + \epsilon^{2^{k'}} = \epsilon^{2^{l'}} + \epsilon^{2^k}$, which implies $l = l'$, and $k = k'$.

Case 3: Values $B, B' = 0$ and $A, A' \neq 0$

Similar to case 2.

Case 4: Values $A, B = 0$

Term $T_2 = 0$ implies $A'\epsilon^{2^l} + B'\epsilon^{2^j} = A'\epsilon^{2^k} + B'\epsilon^{2^i}$, which is true only when $A' = B' = 0$. This reduces to case 6 below.

Case 5: Values $A', B' = 0$

Similar to case 4, reduces to case 6 below.

Case 6: Values $A, B, A', B'$ all zero

Because we have a valid interchange $i < j$, $k < l$, $i' < j'$, and $k' < l'$.

Term $T_3 = 0$ implies $\epsilon^{2^i+2^{k'}} + \epsilon^{2^j+2^{l'}} + \epsilon^{2^{i'}+2^l} + \epsilon^{2^{j'}+2^k} = \epsilon^{2^i+2^{l'}} + \epsilon^{2^j+2^{k'}} + \epsilon^{2^{i'}+2^k} + \epsilon^{2^{j'}+2^l}$

Pairing each term in the L.H.S. with a term in the R.H.S., under the restrictions imposed by the index properties, gives us

$$\begin{array}{lcl} 2^i + 2^{k'} = 2^{i'} + 2^k & \text{or} & 2^i + 2^{k'} = 2^{j'} + 2^l \\ 2^j + 2^{l'} = 2^{j'} + 2^l & \text{or} & 2^j + 2^{l'} = 2^{i'} + 2^k \\ 2^{i'} + 2^l = 2^i + 2^{l'} & \text{or} & 2^{i'} + 2^l = 2^j + 2^{k'} \\ 2^{j'} + 2^k = 2^j + 2^{k'} & \text{or} & 2^{j'} + 2^k = 2^i + 2^{l'} \end{array}$$

For the first equality, choose $2^i + 2^{k'} = 2^{i'} + 2^k$. By the uniqueness of the binary representation, and property P2, this is true only if $i = i'$ and $k = k'$. For the second equality, choose $2^j + 2^{l'} = 2^{j'} + 2^l$ giving the contradiction $j = j'$ and $l = l'$. An alternate second choice $2^j + 2^{l'} = 2^{i'} + 2^k$ cannot be true as $j$ is distinct from $i' = i$ and $k$.

Alternately, for the first equality choose $2^i + 2^{k'} = 2^{j'} + 2^l \implies i = j'$ and $k' = l$. For the second equality, choose $2^j + 2^{l'} = 2^{j'} + 2^l \implies j = j'$ and $l = l'$, giving $i = j$ a contradiction. An alternate choice of $2^j + 2^{l'} = 2^{i'} + 2^k$ for the second equality implies $j = i'$ and $l' = k$. Because $i < j$ implies $j' < i'$, we have a contradiction with $i' < j'$. Thus the only possible relationship between the indices that results in $T_3 = 0$ is $i = i', j = j', k = k'$, and $l = l'$.

□

## 6.4 Computing Jacobi Sets of Time-varying Interpolants

I now describe algorithms for computing the Jacobi sets of the three interpolants: PL, Tri, and Prism.

### 6.4.1 Computing Jacobi Sets of PL Interpolants

The algorithm in this section is due to Edelsbrunner et al. [EH02]. It takes as input any two PL interpolants $f, g$ defined on a common simplicial complex, and produces a collection of edges of the complex that describes the Jacobi set of the functions. Because the Jacobi set can be described as the critical points of one function restricted to level sets of the other, we can choose function $g$ to represent time so that we can use this algorithm for the specific case when $f$ is a time-varying PL interpolant.

Recall that the definition of the Jacobi set of two functions $f, g\colon \mathbb{M} \to \mathbb{R}$ is the set of points in $\mathbb{M}$ where the gradients of both functions are parallel. This condition implies that the Jacobi set is the collection of critical points of the composite function $h = f + \lambda g$, for some $\lambda \in \mathbb{R}$. In the PL case, we must decide if each edge of the simplicial complex is critical by computing the Betti numbers of its lower link under the composite function $h_\lambda = f + \lambda g$. I restate the algorithm for checking criticality of edges.

```
integer isJacobi(Edge $uv$)
  $\lambda = [f(v) - f(u)]/[g(u) - g(v)]$;
  $\mathrm{Lk}_{-}uv = \{\tau \in \mathrm{Lk}\, uv \mid w \leq \tau \Longrightarrow h_\lambda(w) < h_\lambda(v)\}$;
  return isCritical($\mathrm{Lk}_{-}uv$).
```

The sub-routine isCritical can be implemented using the critical point classification described in Table 2.1. We can compute the Jacobi set by invoking isJacobi on each edge of the simplicial complex $K$. Assuming a constant bound on the number of vertices in the link of an edge, and letting $n$ be the number of edges in the complex, the algorithm takes $O(n)$ running time.

**Artifacts of the Jacobi set** Because the Jacobi set of the PL interpolant is restricted to the edges of $K$ it possesses certain undesirable artifacts. Figure 6.6 shows an example.

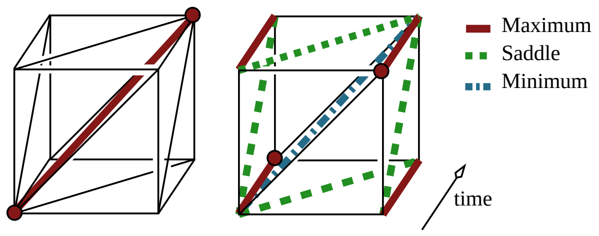


Figure 6.6: A critical point moving along the diagonal, as at left, gives a simple Jacobi curve, but one moving across the diagonal, as at right actually produces Jacobi curves with several possible paths.

Consider a triangulation of a cube about its diagonal from the front lower-left corner to the back top-right corner, and let time increase front to back as shown in Figure 6.6. We note that if the critical point moves in a path aligned with one of the edges connecting adjacent time steps then that single edge is sufficient to represent the trajectory. On the other hand, if the critical point moves along a path not aligned with any mesh edge then the Jacobi edges follow a zig-zag path to connect the critical points, as shown at the right in Figure 6.6. It is not difficult to assign function values to the vertices of the cube and of its neighbors(not shown in the figure) to make the front top-right vertex and the back bottom-left vertex a maximum and generate the zig-zag path. Such a path is undesirable in the implementation of the time-varying Reeb graph algorithm for two reasons.

First, the increase in the number of Jacobi edges required to represent a path is undesirable because the number of events that must be processed when computing time-varying Reeb graphs can be quadratic in the number of Jacobi edges.

Second, the zig-zag path can contain Jacobi vertices with more than two incident Jacobi edges. There are four edges incident to the front lower-left corner vertex, and to the back top-right corner vertex; the vertices are degenerate and contain two birth points and two death points, respectively. If a cube diagonally adjacent to each of these vertices has a similar zig-zag path then each of these vertices will have four more incident edges in that cube. I wish to process each birth or death point at a distinct time to avoid complicated case analysis while computing time-varying Reeb graphs. Therefore, I must unfold each degenerate vertex with multiple birth points into simple vertices with a single birth point, each incident to two Jacobi edges. Recall that the index lemma tells us the indices of the critical points born at a birth point differ by unity. In Figure 6.6, I can pair the incident maximum edge with one of the two saddle edges, but which one is not clear.

I will compare these artifacts that PL produces with any artifacts that Tri and Prism may produce and decide on which one is most suitable for use in computing time-varying Reeb graphs.

**Removing bias.** The zig-zag artifacts produced by PL are due to the bias introduced by the triangulation $K$ about the main diagonal. A natural step to remove this bias is to remove this diagonal and choose another mesh decomposition and interpolation appropriate to that decomposition. Removing the triangulation about the diagonal turns $K$ into the cubical mesh $C$ and we can use Tri for interpolation.

## 6.4.2 Computing Jacobi Sets of Piecewise Trilinear Interpolants

In this section, I describe an algorithm to compute the Jacobi set of a piecewise trilinear interpolant defined on a cubical cell decomposition of space-time.

Recall that Tri can be thought of as a time-varying piecewise bilinear interpolant, and that the critical points of a piecewise bilinear interpolant can be determined by examining the direction of increasing function value of the incident edges. The algorithm that I will describe maintains these edge directions over time to compute the path traced by critical points. Before I can state this algorithm I must classify possible changes to edge directions.

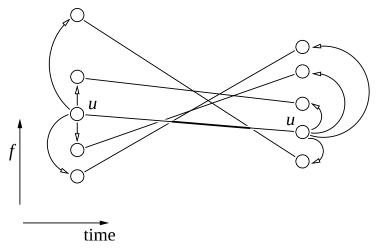


Figure 6.7: An edge in a slice $C_t$ changes direction when its end points swap order at the interchange point of the edges in $C$ that contain these end points. Vertex $u$ remains a minimum for the duration that its adjacent vertices are greater in function value. The segment of $C$ for this duration is shown in bold and is part of the Jacobi set.

**Edge directions in $C_t$ change at interchange points.** At any point in time, the vertices of the slice $C_t$ lie on the edges of $C$, and the edges of $C_t$ correspond to 2-cells of $C_t$. We direct these edges in the direction of increasing $f_t$, and understand how they change direction as we vary time.

In Figure 6.7, we see the edges of $C$ projected onto the function-time plane. The directed arrows drawn at the left end points are the directed edges of the vertex $u$ in slice $C_t$ at that time. As we sweep forward in time from left to right we see that the function value of a vertex adjacent to $u$ drops below or rises above $u$'s function value at the interchange point of the edges of $C$ which contain them. All directed edges point away from $u$ for an interval of time; during this interval $u$ is a minimum, and the segment of the edge of $C$ that contains $u$ and shown in bold is part of the Jacobi set. Because the Jacobi set is a connected curve, this segment is connected to other segments at its endpoints. We must understand and classify all possible connections.

**Critical points are born, die, and move in $C_t$ when an edge changes direction.** We can classify how the change in direction of edge $uv$ changes the criticality of $u$ and $v$, and changes the interior saddle (if any) in the cells adjacent to $uv$ using the classification of critical points based on edge directions developed before, Because there are a total of 9 edges adjacent to the vertices $u$ and $v$, and to the two cells adjacent to edge $uv$, an exhaustive case analysis of before and after combinations can be very complicated. I choose to categorize cases based on the criticality of vertex $u$ after the change, and further categorize each case into subcases depending on whether the cells adjacent to $uv$ contain an interior saddle or not. It is possible to reduce the cases using the following restriction of edge directions.

**Definition 6.4.1 (Forbidden edge directions)** An assignment of directions to the edges of a cell in $C_t$ is *forbidden* if it forms a cycle.

Forbidden edge directions are disallowed because it is impossible to have a cycle of increasing values. Whenever possible, I use this restriction to fix edge directions and reduce the number of cases. We can restrict edge directions to forbid or allow interior saddle by choosing edge directions according to the critical point classification; a cell with opposite edges pointed in the same direction cannot have interior saddles.

I analyze the cases as follows: Let edge $uv$ be directed from $u$ to $v$ before it changes direction; this is the only edge that changes direction in a single case. Fix the directions of the edges adjacent to $u$ to make it critical after the change; this fixes a total of 4 edge directions. Assign edge directions to the remaining 5 edges to achieve three subcases. First assign directions to the two horizontal edges incident on $v$ so that there can be no interior saddles in the two cells adjacent to $uv$, second assign directions so that there can be one interior saddle, and third assign directions so that there can be two interior saddles, one in each cell adjacent to $uv$. It is easy to assign directions to each horizontal edge of $v$ using the directions of the opposite edge incident to $u$; arrows pointing in the same direction ensure no interior saddle, and arrows pointing in the opposite direction ensure an interior saddle either before or after $uv$ changes direction. Assign directions to the remaining 3 edges depending on each subcase as described next.

Figure 6.8, and Figure 6.9 illustrate all cases described next. I classify the main cases based on the criticality of vertex $u$.

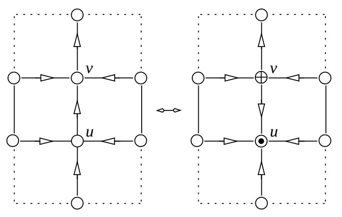
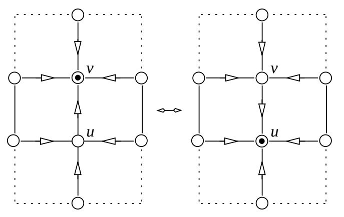
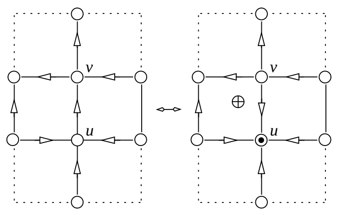
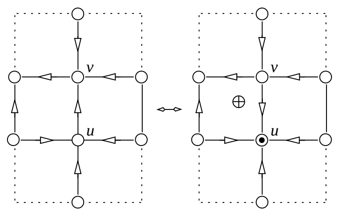
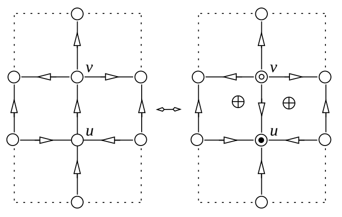
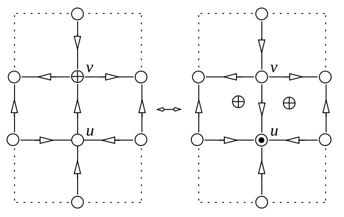
| | Maximum | |
|---|---|---|
| | a | b |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |

Figure 6.8: Edge $uv$ changes direction and $u$ becomes a maximum. Each row is labeled with the number of interior saddles, and each column is determined by the direction of the edge incident to and above $v$.

`Case` Maximum: Fix all edges adjacent to $u$, except edge $uv$, to point towards $u$; edge $uv$ will point towards $u$ after it changes direction.

`Case 0 --` Zero interior saddles**:** The only remaining edge whose direction can affect the criticality of $v$ is adjacent to and above it. Assign it the two possible directions to get the two cases in row 0 of Figure 6.8. The remaining two edges do not matter because they cannot affect $u$, $v$, or the interior saddles.

In the case at left, $v$ changes from a regular point to a saddle, and $u$ changes to a maximum; a pair of critical points are born with indices differing by unity as stated by the Index lemma.

In the case at right, $v$ changes from a maximum to a regular point, and $u$ changes to a maximum; the maximum jumps from $v$ to $u$.

`Case 1 --` One interior saddle**:** Fix edges so that the cell adjacent to and at the left of $uv$ can have an interior saddle. Note that the leftmost edge of this cell has only one valid direction for the assignment not to be forbidden. The only remaining edge whose direction can affect the criticality of $v$ is adjacent to and above it, and we assign to it the two possible directions and get the two cases in row 1 of Figure 6.8.

In both cases, $u$ becomes a maximum along with an saddle in the left cell.

`Case 2 --` Two interior saddles**:** Fix edges so that both cell adjacent to $uv$ can have an interior saddle. The only remaining edge whose direction can affect the criticality of $v$ is adjacent to and above it, and we assign to it the two possible directions and get the two cases in row 2 of Figure 6.8.

In the case at left, two pairs of critical points are born; a maximum at $u$ paired with one interior saddle, and a minimum at $v$ paired with the remaining interior saddle; the choice of which interior saddle to pair with which extremum is arbitrary.

In the case at right, $v$ changes from a saddle to a regular point, and $u$ becomes a maximum paired with one interior saddle. The saddle at $v$ moves to the other interior saddle.

`Case` Minimum**:** To understand this case simply invert the directions of all cases considered when $u$ is a maximum.

`Case` Saddle**:** The analysis is similar to the case in which $u$ is a maximum. Each case is decided by the number of interior saddles, and each subcase is decided by the direction of the edge incident to and above $v$.

`Case 0 --` Zero interior saddles**:** This is row 0 in Figure 6.9. At the left, a pair of critical points are born; a minimum at $v$ and a maximum at $u$. At the right, a saddle moves from $v$ to $u$.

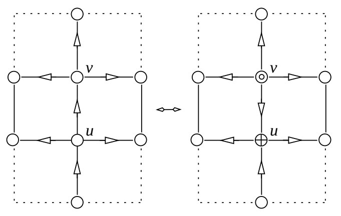
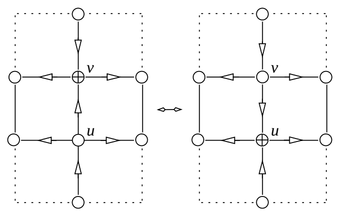
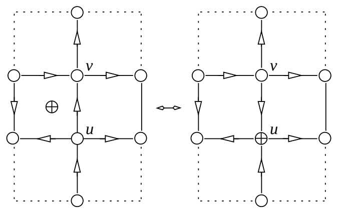
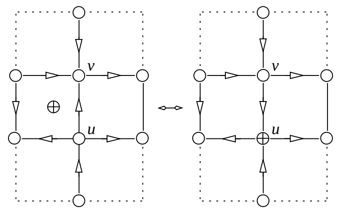
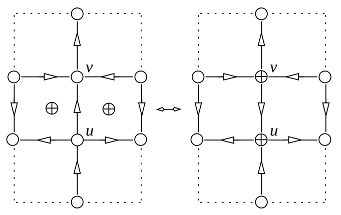
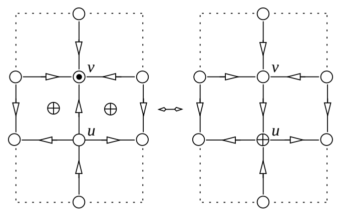
| | Saddle | |
|---|---|---|
| | a | b |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |

Figure 6.9: Edge $uv$ changes direction and $u$ becomes a saddle. Each row is labeled with the number of interior saddles, and each column is determined by the direction of the edge incident to and above $v$.

Case 1 -- One interior saddle: In both subcases, an interior saddle moves to $u$.

Case 2 -- Two interior saddles: At the left, one interior saddle moves to $u$, and the other interior saddle moves to $v$. At the right, we have a maximum at $v$ destroyed by one interior saddle, while the other interior saddle moves to $u$.

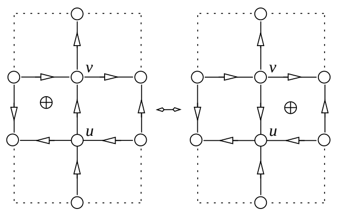


Figure 6.10:An interior saddle moves from one cell adjacent to $uv$ to the other.

Case Regular: There is only one subcase here that keeps both $u$ and $v$ regular before and after edge $uv$ changes direction; an interior saddle moves from one cell adjacent to $uv$ to the other, as shown in Figure 6.10. All other cases either map to one of the cases discussed before, or do not change any critical point.

Each case can be run in reverse; a birth is changed to a death when we do this. This classification of changes will assist us in computing the Jacobi set of a time-varying piecewise bilinear interpolant. We sweep forward in time, maintaining edge directions, and tracing the path of the critical points. There are two types of segments in the path: *critical segments* that connect two points in different times, and *connector segments* that connect two points with the same time coordinate. Forward segments can be sections of the edges of $C$ that connect vertices in successive times, or can be the path traced by an interior saddle. The former are linear segments; I describe the latter.

**Interior saddle path.** We derive the path of an interior saddle by making $f_{00}, f_{01}, f_{11}, f_{10}$ functions of time in Equation 6.1.
Letting $f_{ij}(t) = f_{ij}(0) + t(f_{ij}(1) - f_{ij}(0))$, for $0 \leq t \leq 1$, the coordinates of the interior saddle are

$$(\frac{f_{00}(t) - f_{01}(t)}{\delta(t)}, \frac{f_{00}(t) - f_{10}(t)}{\delta(t)}) \tag{6.3}$$

Each coordinate is a rational function with linear numerator and denominator. Equation 6.3 tells us that interior saddles do not follow a linear path. Unfortunately, this makes the Piecewise trilinear interpolant impractical for computing time-varying Reeb graphs, because to compute the time when nodes swap in the Reeb graph, we must compute the interchange time of two non-linear paths which can be expensive compared to computing the interchange time of linear segments.

I can now describe the algorithm to compute the Jacobi set.

**Algorithm for Jacobi set of** Tri. The algorithm takes as input a cubical grid with a time-varying piecewise bilinear interpolant defined on it, and produces the Jacobi set of the interpolant as a collection of critical segments and connector segments which join the endpoints of the critical segments.

For each vertex $u$ of $C_0$, we sweep forward in time maintaining the directions of a total of 12 edges of the four adjacent 2-cells. Because the edges change directions at interchange times which are distinct according to the interchange lemma only one edge can change direction at a time. Store all times at which the adjacent edges change direction in a priority queue. Retrieve the times in increasing order and use the classification of changes to identify when critical points are born, die or move from vertex $u$ or to vertex $u$, or from one adjacent cell to another adjacent cell. Generate critical segments and connector segments as described next and shown in Figure 6.11.

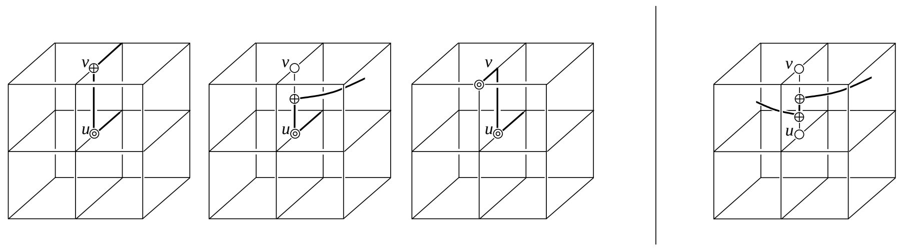


Figure 6.11: Tracing a critical point. Time increases into the figure. On the left, a minimum is born with a vertex saddle, an interior saddle and moves from $v$ to $u$. On the right, a saddle ends in the cell to the left of $uv$ and moves to the cell on the right. Although the saddle ends and begins on $uv$, the coordinates might not coincide.

`Case` Birth**:** Start a critical segment at each critical point in the pair, and connect them with a connector segment.

`Case` Death**:** End the critical segments tracing each critical point in the pair, and connect them with a connector segment.

`Case` Move**:** End the critical segment of the critical point before the move, and start a forward at the new location. Connect the previous critical segment and the new critical segment with a connector segment.

Letting $n$ be the number of edges in the regular grid, a sweep of each edge processes 12 interchange events, giving a running time of $O(n)$.

**Properties of the Jacobi set of** Tri**.** We list some properties of the Jacobi set of Tri for comparison with the wish-list presented before.

[T1] All saddles are simple.
This follows from the classification of critical points of piecewise linear interpolant. There is one configuration each, of adjacent edges, for a vertex saddle or an interior saddle as shown in Figure 6.2. Each configuration is that of a simple saddle.

[T2] No two maxima or minima move from one vertex to another at the same time.
This follows from the property that maxima and minima lie on the vertices of $C_t$ and move from one vertex to the next at an interchange time, which is distinct according to the Interchange lemma.

Property T1 implies that there are no degenerate Jacobi edges (multiplicity greater than unity), that must be unfolded. In some cases several events occur at the same time and can produce a degree 4 Jacobi vertex which must be unfolded: two pairs of critical points are born (case maximum 2a), one pair of critical points is born and one critical point moves (case maximum 2b), two critical points move (case saddle 2a), and one pair of critical points dies and one critical point moves(case saddle 2b).

**Removing non-linear segments.** As explained before, the crucial problem in using Tri for computing time-varying Reeb graphs is that it produces non-linear paths whose interchange events can be expensive to compute compared to interchange events of linear segments. If I can choose a mesh decomposition that forbids the occurrence of interior saddles and removes the bias found in PL, then I have some hope of producing Jacobi sets that I can use to build time-varying Reeb graphs. Prism does exactly that by adding a diagonal in every square of a time-slice to forbid interior saddles, while avoiding a triangulation about the main diagonal of the cube to remove the bias found in PL.

### 6.4.3 Computing Jacobi Sets of Piecewise Prismatic Interpolants

In this section, I describe an algorithm to compute the paths traced by critical points of prismatic interpolant defined on a prismatic decomposition of space-time. The algorithm is similar to the one for piecewise trilinear interpolant described earlier; it sweeps forward in time while maintaining the lower link of a vertex in the slice $P_t$ to detect when the vertex becomes critical. Because the piecewise prismatic interpolant can be considered as a time-varying PL interpolant, the classification of changes when a vertex enters or leaves the lower link of another vertex is simpler than the classification of change in edge direction for the piecewise trilinear interpolant; there are no interior saddles and critical points can only move from one vertex of $P_t$ to another.

We classify the changes that occur when a vertex $v \in P_t$ enters the lower link of another vertex $u \in P_t$; we restrict our classification to only those changes that make $u$ critical. As noted before, vertex $v$ enters or leaves the lower link of $u$ when the edges in $P$ on which they lie swap order at an interchange point. We use this classification in an algorithm to construct the Jacobi set.

**Classifying critical points over time.** We classify all possible changes when $u$ becomes critical after $v$ enters its lower link. We can run each in reverse to understand how $u$ changes to a regular point from a critical point. Figure 6.12 illustrates all cases.

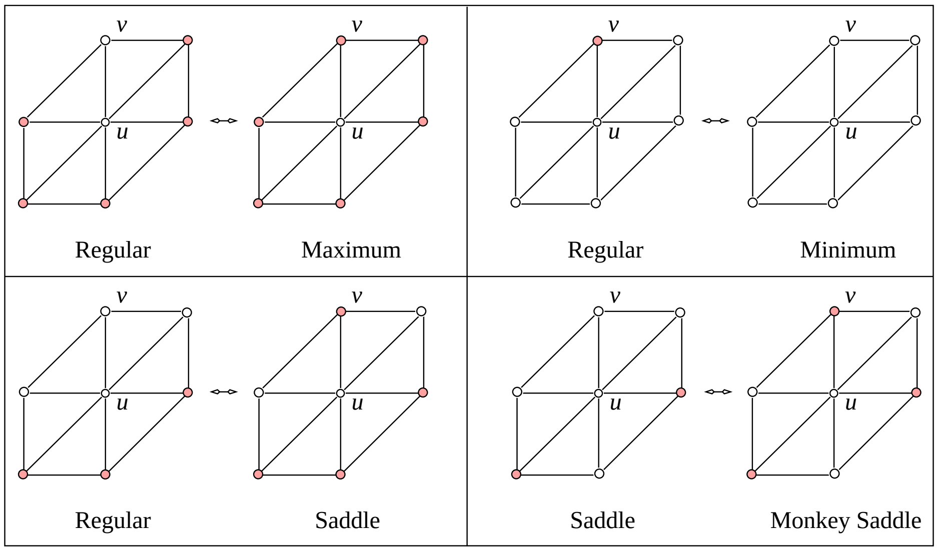


Figure 6.12: Vertex $u$ in $P_t$ can become critical when vertex $v$ enters or leaves its lower link. Lower link vertices are shaded. Top left, a regular vertex $u$ becomes a maximum when vertex $v$ becomes the last to enter its lower link. Bottom left, a regular vertex $u$ becomes a simple saddle when vertex $v$ becomes a component in the lower link by itself. Bottom left, a simple saddle vertex $u$ becomes a monkey saddle when vertex $v$ becomes a component in the lower link by itself.

`Case` Regular to Maximum: Vertex $u$ becomes a maximum after the change if all other vertices are already in the lower link. At the time of the change the edge $uv$ is critical because its lower link is the same as its link.

`Case` Regular to Minimum: Vertex $u$ becomes a minimum after $v$ enters the upper link if all other vertices are already in the upper link. At the time of the change the edge $uv$ is critical because its lower link is empty.

`Case` Regular to Saddle: Vertex $u$ becomes a saddle (simple) if it was a regular point before and if $v$ is in a lower link component by itself after. At the time of the change the edge $uv$ is critical because its lower link is empty.

`Case` Regular to Monkey Saddle: Vertex $u$ becomes a monkey saddle if it was a simple saddle before and if $v$ is in a lower link component by itself after. At the time of the change the edge $uv$ is critical because its lower link is empty.

The Jacobi edges of the piecewise prismatic interpolant are composed of critical segments and connector segments as that of the piecewise trilinear interpolant. All critical segments for the prismatic interpolant are linear unlike for the trilinear interpolant where some critical segments can be curves. The algorithm to compute the Jacobi set is straightforward.

**Algorithm for Jacobi set of Prism.** The algorithm takes as input a prismatic decomposition of space-time with a prismatic interpolant defined on it, and produces the Jacobi set as a collection of linear critical segments connected by connector segments.

For each vertex $u$ of $P_0$ , we sweep forward in time while maintaining its lower link. As mentioned before, each sweep event occurs when the two edges of $P$ containing the two vertices $u$ and $v$, which is in $u$'s link, change order at an interchange point. Store all times at which a vertex $v$ enters or leaves the lower link of $u$ in a priority queue. Retrieve the times in increasing order and use the classification of changes to $u$ to detect when it becomes critical, or regular. Start a critical segment at $u$ and create a connector segment $uv$ when $u$ becomes critical or changes from a simple saddle to a monkey saddle. End a critical segment at $u$ and create a connector segment $uv$ when $u$ becomes regular or changes from a monkey saddle to a simple saddle.

Letting $n$ be the number of edges in the prismatic grid, a sweep of each edge processes 6 interchange events, giving a running time of $O(n)$.

**Properties of the Jacobi set of Prism.** We list some properties of the Jacobi set of Prism for comparison with the wish-list presented before.

[P1] No two maximum or minimum move from one vertex to another at the same time.
A maximum or a minimum vertex $v$ moves to another vertex $u$ at the time when the edges of $P$ that contain the vertices change order at an interchange which must a distinct time coordinate according to the interchange lemma.

[P2] No two birth or death points occur at the same time. An interchange of two edges in $P$ changes the link of the two vertices $u$ and $v$ lying on the interchanging edges and can correspond to only one birth or death point.

The Prism interpolant can produce Jacobi edges with multiplicity 2 (monkey saddles), which must be unfolded. Property P2 satisfies criterion J2, which requires that birth, death, and interchange events occur at distinct times.

## 6.5 Experiments

In this section I describe some results from preliminary experiments with synthetic and simulation datasets. I compute Jacobi sets for each of the three interpolants, and compare them for running time, size, and use in the algorithm to compute time-varying Reeb graphs.

**Datasets.** Table 6.1 lists the data used. Each data-set is defined on a regular cubical grid. Each time step is converted to a sampling on a sphere by connecting the boundary points to a single point with function value $-\infty$. The synthetic dataset MovMax is

| Dataset | Dim.$(x \times y \times t)$ |
|---|---|
| MovMax | $64 \times 64 \times 10$ |
| Combust | $600 \times 600 \times 67$ |

Table 6.1:Datasets and their dimensions.

a sampling of a height field containing three maxima moving across a plane over time. The Combust dataset is from a simulation of fuel combustion inside an engine, and is courtesy of Jacqueline Chen of Sandia National Labs. The simulation computes various scalar quantities to understand the influence of turbulence on ignition, flame propagation, and burnout [EC01]. Upon compression, the inhomogeneity in the air-fuel mixture causes ignition at multiple spots. Depending on the air-fuel ratio, the flame propagates from these spots outwards or it burns out.

**Statistics.** In Table 6.2, I list statistics for the Jacobi sets computed using each of the three algorithms.

| Interpolant | MovMax | | | Combust | | |
|---|---|---|---|---|---|---|
| | # v | # e | time(s) | # v | # e | time(s) |
| PL | 720 | 998 | 0.084 | 436,306 | 693,118 | 61.769 |
| Tri | 640 | 624 | 0.340 | 685,432 | 682,915 | 211.972 |
| Prism | 614 | 602 | 0.201 | 914,713 | 913,144 | 117.423 |

Table 6.2: The table lists the number of vertices and edges in the Jacobi set, and the running times for the Jacobi set using PL, Tri, and Prism.

Although each algorithm has the same running time complexity, $O(n)$, where $n$ is the number of edges in the complex, the constants are largest for Tri, followed by Prism, and PL. The running times in Table 6.2 reflect this.

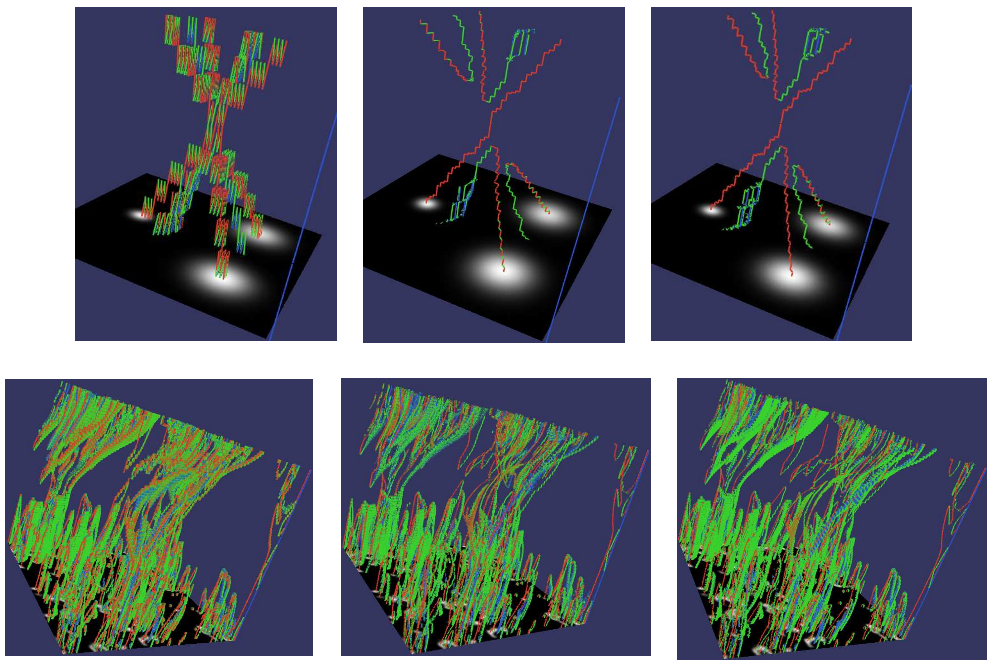


Figure 6.13: Top row, from left to right, Jacobi sets for MovMax computed using PL, Tri, and Prism respectively. The textured square is at time 0, and time increases upwards. Bottom row, the paths for Combust. For MovMax, the Jacobi set computed using PL exhibit severe zig-zag artifacts, but the Jacobi sets computed using Tri and Prism are cleaner, and lack the zig-zag artifacts. Combust is inherently turbulent and all paths look similar.

**Comparison.** I compare the structural properties of the Jacobi sets computed using PL, Prism, and Tri for use in computing time-varying Reeb graphs using the wish-list presented in Section 6.1.

[Manifold]

Both PL, and Prism can produce degenerate edges that must be unfolded, while Tri produces no degenerate edges. Both PL, and Tri can produce degenerate vertices that must be unfolded by pairing the correct edges according to the Index lemma to get birth, and death points; unfolding for Tri is easier than for PL because Tri can produce degenerate vertices with maximum degree of 4, but PL can produce degenerate vertices with degree 8, which increases the possible pairing options to consider.

[Distinct times]

PL (on a regular grid) produces all birth and death events at discrete time steps, and none in between; this breaks the assumptions of the time-varying Reeb graph algorithm and makes implementation cumbersome due to special case handling.

Tri produces at most 2 birth or death events at the same time (case maximum 2a), and can be handled without much difficulty. Prism produces all birth and death events at distinct times.

[Few events]

The analysis in Section 4.5 tells us that the number of events $E$ at which these changes occur is bound from above by the square of the number of critical segments $k$. Table 6.3 lists the number of such edges in the path. Both Tri and Prism fare better than PL. Note that all edges produced by PL connect vertices at different times, but Tri and Prism have connector segments whose end-points lie within the same time.

As noted before, a severe drawback for Tri is that it produces internal saddles with non-linear paths; computing their interchange points is expensive compared to the linear paths produced by Prism and PL.

From the discussion above it is clear that Prism is best suited for computing Jacobi sets, and time-varying Reeb graphs.

| Interpolant | MovMax | Combust |
|---|---|---|
| PL | 998 | 693,118 |
| Tri | 349 | 371,663 |
| Prism | 333 | 484,121 |

Table 6.3:Number of edges in the Jacobi set that connect vertices in different times.

## 6.6 Conclusion

In this chapter, I have used piecewise linear, piecewise trilinear, and piecewise prismatic interpolants to extend a discretely sampled function to all points in space-time. I have developed algorithms to compute the Jacobi set of these interpolants, and considered the pros and cons of using the Jacobi set of each interpolant in computing time-varying Reeb graphs. The properties of the Jacobi set and the experiments with synthetic and simulation datasets indicate that the prismatic interpolant is the best choice among the three for computing time-varying Reeb graphs.

Although I have restricted my study to interpolants of time-varying samples on the 2-sphere, the lessons learned while developing the classification of changes to critical points, and the algorithms to compute the Jacobi set can be extended to interpolants for samples of time-varying functions on the 3-sphere. I avoid using the time-varying piecewise trilinear interpolant to compute the Jacobi set in higher dimensions because it has non-linear segments unsuitable for use in computing time-varying Reeb graphs. The piecewise prismatic interpolant can be extended by considering prisms with simplices as bases, and classifying how critical points change by studying their lower link just as we did in this chapter. A sweep algorithm similar to the one used in this chapter to compute Jacobi sets is straightforward. I will develop this extension in future work.

# Chapter 7

# Constructing Time-varying Reeb Graphs

In this chapter, I describe key details from the implementation of the algorithm that constructs time-varying Reeb graphs. I believe that these details can be useful to readers who wish to implement this algorithm.

I describe the implementation for time-varying functions defined on $\mathbb{S}^2$; there are fewer types of critical points, degenerate cases, and interchange cases than for time-varying functions on $\mathbb{S}^3$. Moreover, I'm able to use existing graphics packages to visualize time-varying functions on $\mathbb{S}^2$ and their Jacobi sets because the total dimension of space-time is three. This ability is invaluable while debugging an implementation. Indeed, my experience shows that problem cases and artifacts that arise in lower dimensions, and the solutions developed, can be generalized to higher dimensions.

A complete description of an implementation for time-varying functions on $\mathbb{S}^2$ requires enumerating the possible birth, death, and interchange cases. I enumerate these cases is Section 7.1. Because the Jacobi set connects Reeb graphs over time, and is required to decide birth, death, and interchange events, it is constructed first. I describe how to compute the Jacobi set for the Prism interpolant in Section 7.2. The Jacobi set can have degenerate edges. I unfold these to make the Jacobi set a 1-manifold, which I then use to compute time-varying Reeb graphs. I describe how to unfold a degenerate edge of the Jacobi set into simple critical edges in Section 7.3. After unfolding the Jacobi set into a 1-manifold we construct the Reeb graph at time $t = 0$, and maintain it during the sweep forward in time. The Reeb graph changes at birth, death, and interchange events, which occur at distinct times in the sweep. Each event changes the Reeb graph according to the classification described in Section 7.1. At each event

I decide the correct connections between nodes and make modifications to the Reeb graph, as described in Section 7.4.

## 7.1 Birth, Death and Interchange Events

In this section, we enumerate the possible birth, death, and interchange events, and the modifications to the Reeb graph at these events.

**Nodes appear and disappear.** Nodes appear or disappear in pairs with indices differing by unity, at birth and death points. For $d = 2$ we have three types of critical points: minima with index 0, saddles with index 1, and maxima with index 2. By the Index lemma, we have 0-1 and 1-2 birth and death points. These cases are illustrated in Figure 7.1, which we can read left to right for the 1-2 birth point, upside down for the 0-1 birth point, and right to left (backwards in time) for the corresponding death points.

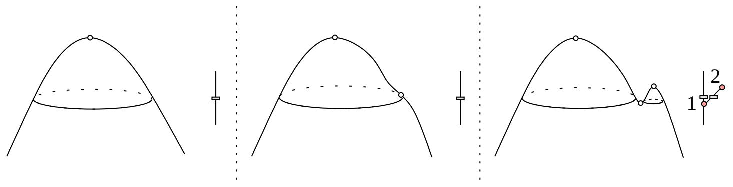


Figure 7.1: The function with level curves and Reeb graph around a 1-2 birth point. Time increases from left to right. Going forward in time, we see the sprouting of a bud; going backward in time we see its retraction.

**Nodes swap.** As in Section 4.3, we can enumerate all possible interchange events for a pair of critical points $x$ and $y$. There are fewer cases with two spatial dimensions than with three, because each critical point is either a merge (M) or split (S) for its level set components. Thus, there are four combinations that can occur before or after an interchange event: we list them as MM, MS, SM, SSfor the critical points in order of increasing function value. The possible before/after pairs give the cases illustrated in Figure 7.2. (Each case has two interpretations: one with time flowing left to right, and one with time flowing right to left.) It is convenient to group together the cases with similar configurations before the swap.

`Case 1` (MM) Both before and after the interchange, three components merge into one. The only change is the order of merging. [We pair MM with itself.]

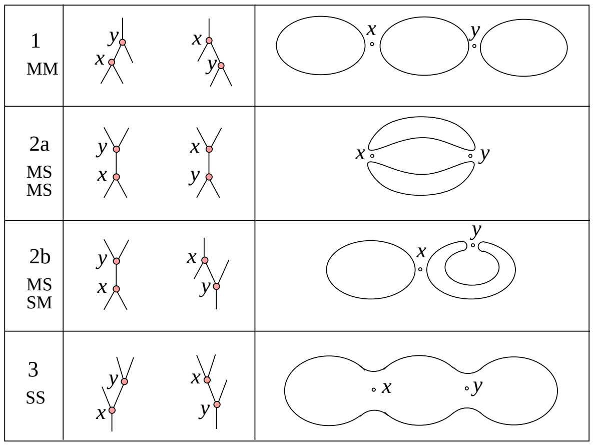


Figure 7.2: On the left, Reeb graph portions before and after the interchange of $x$ and $y$. On the right, level curves at a value just below the function value of $x$ and $y$.

`Case 2` (MS, SM) Before the swap, $x$ merges two components and $y$ splits the component. Either $y$ splits the two components that were merged by $x$, so after the interchange $x$ and $y$ just swap, or $y$ splits only one of the two components, so after the swap $y$ first splits that component and $x$ merges one of the split components with the remaining component. [In the first configuration, we pair MS with itself to get `Case 2a`, and in the second we pair MS with SM to get `Case 2b`.]

`Case 3` (SS) Both before and after the interchange, a component splits into three. The only change is the order of splitting [We pair SS with itself.]

## 7.2 Computing the Jacobi Set

We use a prismatic decomposition $P$ of space-time, and compute the Jacobi curve for the Prism interpolant using the algorithm described in Chapter 6. Recall that this algorithm sweeps each edge $e$ of $P$ by moving the slice $P_t$ forward in time. It maintains the link of the vertex $u = e \cap P_t$ to detect the time interval for which $u$ is critical. A vertex $v$ enters or leaves the link of vertex $u$ at distinct times at which edge $e$ interchanges with a neighboring edge that contains vertex $v$. I implement the sweep by processing events in increasing order of interchange times. I describe a data-structure to represent an interchange time, and provide a sub-routine to compare two interchange times.

**The Time data-structure.** The time of interchange is given by Equation 6.2. We restate it here with Figure 7.3 for convenience. In order to simulate simplicity [EM90] we perturb the function value at each vertex with index $i$, as $\hat{f}_i = f_i + \epsilon^{2^i}$ The interchange time is given by

$$\mathsf{t_x} = t + \frac{\hat{f}_i - \hat{f}_j}{(\hat{f}_i - \hat{f}_j) - (\hat{f}_l - \hat{f}_k)}$$

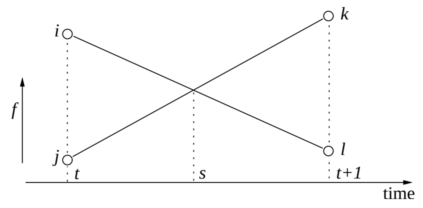


Figure 7.3: The projection of two line segments onto the function-time plane, intersect at an interchange.

We represent the interchange time using 7 real numbers: the base time $t$, two function differences $A = f_i - f_j$ and $B = f_l - f_k$, and the four indices $i, j, k$, and $l$. We can compare two interchange times, $\mathsf{t_x}$ and $\mathsf{t_x}'$ by determining the sign of the expression for the discriminant $\Delta = \mathsf{t_x} - \mathsf{t_x}'$ used in the proof of the Interchange lemma. The comparison is straightforward when $t \neq t'$ or $A'B - AB' \neq 0$. It becomes complicated otherwise, because we have to evaluate a symbolic expression in $\epsilon$ [EM90]. In this case, we can compare interchange times by sorting their indices and comparing them lexicographically according to the expression for the discriminant.

There is a pictorial interpretation of the perturbation that yields a simpler implementation. We need the properties I1 and I2, restated below.

I1: The indices $i, j, k, l$ are distinct.

I2: Each of $\{i, j\}$ is smaller than each of $\{k, l\}$.

Property I2 implies that the perturbation applied to vertices at time $t$ is always larger than that applied to the vertices at time $t+1$. By our particular choice of perturbation, the vertex with the smallest index among a set of vertices gets perturbed the most; the other vertices are considered unperturbed. Therefore, for two interchange points with $t = t'$ and $A'B - AB' = 0$ we use the vertex with the smallest index at time $t$ to determine the sign of $\Delta$.

I illustrate possible cases in Figure 7.4, where the edge connected to the vertex with smallest index is dotted.

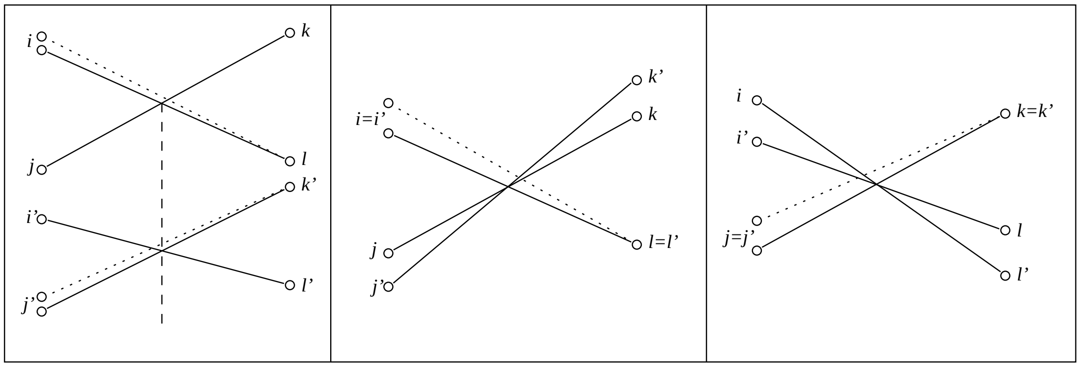


Figure 7.4: Comparing interchange times. The edge connected to a vertex with maximum perturbation is shown dotted. Left, all edges are distinct. If the smallest index is one of $i$ or $j'$ (both are perturbed in the picture), then $\mathsf{t_x} > \mathsf{t_x}'$, else $\mathsf{t_x} < \mathsf{t_x}'$. Middle and right, the interchanging pair of edges share a common edge. If the smallest index belongs to the shared vertex, then the order is determined by comparing $\hat{f}_k$ and $\hat{f}_{k'}$ for the middle case, and by comparing $\hat{f}_i$ and $\hat{f}_{i'}$ for the right case. The cases $i' = j$, or $i = j'$ are handled similar to the distinct edge case.

If all edges in the two interchanges are distinct we have the case at left. The relative time order is determined by computing the smallest index among $i, i', j$ and $j'$. If this index is one of $i$ or $j'$ then $\mathsf{t_x} > \mathsf{t_x}'$, else $\mathsf{t_x} < \mathsf{t_x}'$. I show edges connected to $i$ or $j'$ as dotted to illustrate both cases in the same figure.

If the interchanging pairs of edges share a common edge we have the middle case and the last case. If the minimum index does not belong to the common edge I proceed as in the first case. Otherwise, I compare $\hat{f}_k$ and $\hat{f}_{k'}$ for the middle case, and $\hat{f}_i$ and $\hat{f}_{i'}$ for the right case to determine the time order. The cases in which $i' = j$ or $i = j'$ do not need extra comparisons and are treated like the distinct edge case.

We are now ready to state the time comparison function.

```
boolean cmpTime(Time $\mathsf{t_x}$, Time $\mathsf{t_x}'$)
  $D = A'B - AB'$;
  if ($t = t'$ and $D = 0$)
    | $i_m$ = min($i, i', j, j'$);
    | if ($i = i'$ and $i_m = i$) return $\hat{f}_{k'} < \hat{f}_k$;
    | elseif ($j = j'$ and $i_m = j$) return $\hat{f}_i < \hat{f}_{i'}$;
    | elseif ($i_m = i$ or $i_m = j'$) return false;
    | else return true;
  elseif ($t \neq t'$) return $t < t'$;
  else return $D < 0$;
```

Next, we compute a *critical segment* of the Jacobi set. segment connects two points in different times and is a section of an edge in $P$ that contains a critical point for this duration.

Recall that a critical segment is an interval of an edge $e$ in $P$ that has a critical point in the slice $P_t$ for every time $t$ in the interval. Note that only edges that connect points in different times may have critical segments.

**Computing critical segments of the Jacobi set.** I describe an algorithm to compute the critical segment (if any) on an edge $e$ of $P$. A critical segment of and edge $e$ is specified by the interval of time for which the vertex $u = e \cap P_t$ is critical. The input is an edge of $P$ that connect points in different times, and the output is the interval of time (if any) for which that edge is in the Jacobi set. This algorithm computes interchanges between edge $e$ and its neighbors, and processes them in increasing order of time. At each event, it decides if vertex $u$ is critical by examining its lower link.

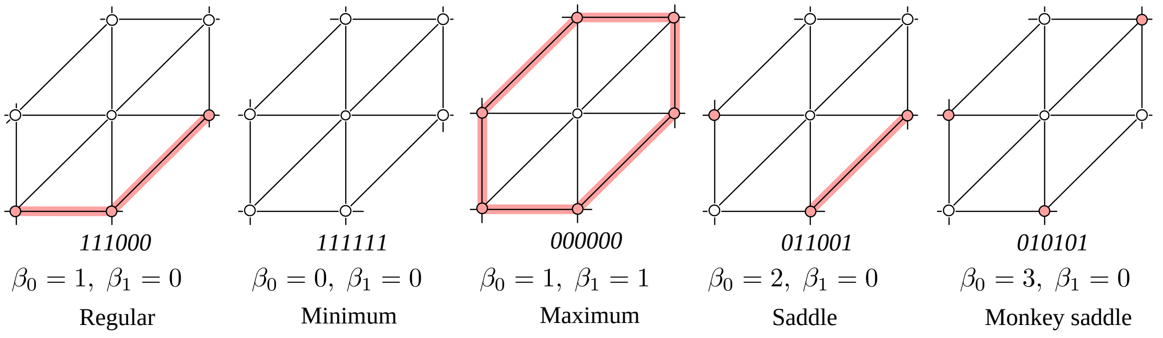

111000
$\beta_0 = 1,\ \beta_1 = 0$
Regular
111111
$\beta_0 = 0,\ \beta_1 = 0$
Minimum
000000
$\beta_0 = 1,\ \beta_1 = 1$
Maximum
011001
$\beta_0 = 2,\ \beta_1 = 0$
Saddle
010101
$\beta_0 = 3,\ \beta_1 = 0$
Monkey saddle

Figure 7.5: To determine the critical segment of an edge $e$ of $P$, I look at the link of its vertex in slice $P_t$, for varying $t$. This figure shows the link of the center vertex in a slice of the prismatic decomposition. Lower link vertices and edges are shaded. The link is written, below each case, as a binary string going clockwise from the left vertex with a 0 for a vertex in the lower link, and a 1 for a vertex in the upper link. The Betti numbers $\beta_0$ (number of components) and $\beta_1$ (number of cycles) of the lower link distinguish the different cases; they are shown below each case.

Figure 7.5 shows the link of a vertex when it is a regular point, a maximum, a minimum, a saddle, or a monkey saddle (degenerate). The Betti numbers $\beta_0$ (number of components) and $\beta_1$ (number of cycles) of the lower link distinguish the different cases; they are shown below each case in Figure 7.5.

I represent the underlying mesh in a MESH data-structure as a list of arrays of real values; each array stores the function for a single time-step. The MESH data-structure provides access to individual vertex function values and also provides the following functions.

GETLNK$(E, t)$ : return the link of vertex lying on edge $E$ at time-slice $t$ as a binary string.

NUMX$(E, t)$ : return the number of interchanges with edge $E$ after time $t$.

NEXTXEDGE$(E, t)$ : return the edge that has an interchange with edge $E$ after time $t$.

NEXTXTIME$(E, t)$ : return the time of the next interchange with edge $E$ after time $t$.

If $P$ is defined on a regular grid, then each vertex in $P_t$ has a fixed sized link. In this case, I represent the link by a fixed size binary bit string with a 0 for a vertex in the lower link and 1 for a vertex in the upper link.

I can now present sub-routine CRITICALSEG which detects when vertex $u = e \cap P_t$ is critical and returns this time interval.

```
(Time, Time) CRITICALSEG(Edge $E$)
    $L = $ GetLnk$(E, 0)$; $sT = pT = 0$; $pI = $ isCritical$(L)$;
    while NumX$(E, pT) \neq 0$
        $F = $ NextXing$(E, pT)$; $pT = $ NextXtime$(E, pT)$;
        FlipBit$(L, F)$; $I = $ isCritical$(L)$;
        if $(pI = $ REGULAR and $I = $ CRITICAL$)$
            $pI = I$; $sT = pT$;
        elseif $(pI = $ CRITICAL and $I = $ REGULAR$)$
            return$(sT, pT)$;
```

I compute the Jacobi set by invoking CRITICALSEG for every edge of $P$ which connects points at different times, and represent it by a collection of critical segments, connected by connector segments, as explained in Section 6.4.

## 7.3 Unfolding Degenerate Jacobi Edges

The interchange classification presented in Section 4.3 and Section 7.1 assumes that all critical points are simple. This assumption is required to avoid complicated case analysis and also results in a simpler implementation. In practice, the Jacobi set can contain degenerate saddles (monkey saddles), as shown in Figure 7.5, In fact, such saddles can exist for a length of time on a degenerate Jacobi edge, and have to be unfolded into simple saddle edges. I adapt the general unfolding strategy for degenerate critical points explained in [CSA03, EHNP03], to unfold degenerate edges.

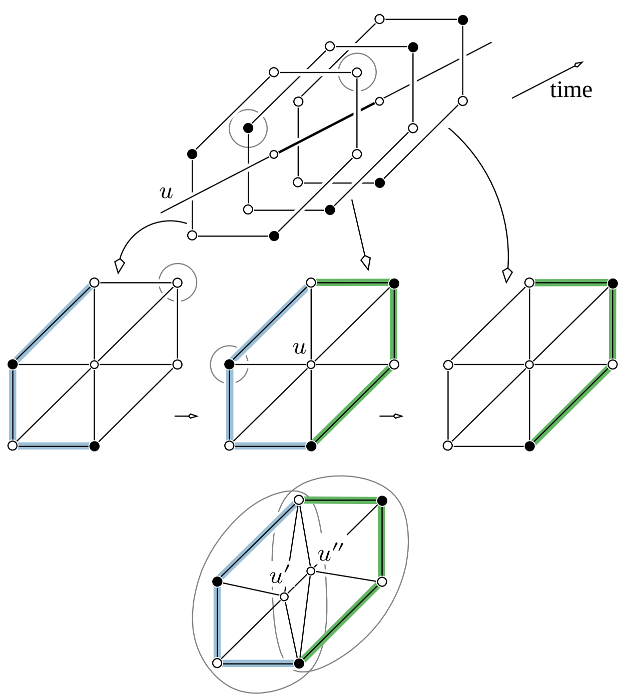


Figure 7.6: Unfolding a monkey saddle into simple saddles. Lower link vertices are filled, and upper link vertices are unfilled. Top, links inside three time-slices showing a simple saddle changing into a monkey saddle and back into a simple saddle. Middle, the same links shown side-by-side, with their vertices that make and break the monkey saddle circled. Bottom, the monkey saddle unfolded into two simple saddles such that the unfolded links match the simple saddle links before and after the monkey saddle.

A critical segment that contains a monkey saddle, is always adjacent at each endpoint to a critical segment that contains a simple saddle, as shown in Figure 7.6. I call a critical segment that contains a monkey saddle a *monkey-saddle segment*, and a critical segment that contains a simple saddle a *simple-saddle segment*. Because the lower link (and the upper link) of a monkey saddle in any slice of the monkey-saddle segment does not change, I choose an unfolding of a monkey saddle in any slice and apply it to the entire segment and unfold it. There are several possible unfoldings of a monkey saddle; I choose a particular unfolding that maintains continuity of the unfolded links with the links of the adjacent simple saddle segments.

I need some definitions to explain what I mean by continuity of links.

A vertex $u$ in the slice $P_t$ *maps* to vertex $v$ in another slice $P_{t'}$ if both vertices lie on the same edge of $P$.

A vertex $v$ in a slice $P_t$ is said to *change polarity* with respect to vertex $u$ at time $t$ if it enters or leaves the lower link of vertex $u$ in the slice.

If a monkey-saddle at $u$ is created by a vertex $v$ changing polarity at time $t$, then $v$ is a *make vertex*, and time $t$ is the *make time*. The circled vertex in the bottom-left picture in Figure 7.6 is a make vertex because it changes polarity to make $u$ a monkey saddle.

If a monkey-saddle at $u$ is changed back to a simple saddle by a vertex $v$ changing polarity at time $t$, then $v$ is a *break vertex*, and time $t$ is the *break time*. The circled vertex in the bottom-middle picture in Figure 7.6 is a break vertex because it changes polarity to make $u$ a simple saddle.

Consider the link of the monkey saddle $u$ in a slice of the monkey-saddle segment, and the link of the simple saddles $u'$ and $u''$ in the same slice after the monkey-saddle segment is unfolded in any one of the possible ways. In the slice, the link of each unfolded vertex shares a sequence of alternating vertices and edges with the link of the monkey saddle, shown in matching colors at the bottom row and the central picture in the middle row of Figure 7.6. This shared sequence of vertices also maps (via the edges of $P$ on which they lie) to a sequence of vertices in the link of a slice of a simple-saddle segment adjacent to the monkey-saddle segment, shown in matching colors in the middle row of Figure 7.6.

A link of an unfolded vertex *matches* a link in the slice of the adjacent simple-saddle segment, if the vertices in these two links that map to each other are in the lower link together, or in the upper link together.

Because unfoldings are not unique we can choose an unfolding that matches each unfolded link to that of the adjacent simple saddle. The choice is implemented by a simple strategy: choose the unfolding that places each make vertex or break vertex in exactly one unfolded link. The simple saddle edge adjacent to the monkey saddle at its make time is connected to the unfolded edge whose link does not contain the make vertex, and the simple saddle edge adjacent to the monkey saddle at its break time is connected to the unfolded edge whose link does not contain the break vertex,

## 7.4 Deciding Between Interchange Cases

After unfolding the Jacobi set into a 1-manifold we construct the Reeb graph at time $t = 0$, and maintain it during birth, death, and interchange events, which occur at distinct times in the sweep forward in time. At each event we decide the correct connections between nodes and make modifications to the Reeb graph using the Path function described in this section.

The Path function is instrumental in distinguishing between the various configurations at an interchange event. Recall, from Section 4.5, that Path($u$) returns a monotone path connecting leaves in the Reeb graph. This path contains the point representing the level set component of $f_t$ passing through the vertex $u$, which is either on the Jacobi curve or in the link of such a point. We can implement this function using three subroutines:

Descend($u$) : return a minimum that is reached from vertex $u$ by a path monotonically decreasing in $f_t$.

Ascend($u$) : return a maximum that is reached from vertex $u$ by a path monotonically increasing in $f_t$.

FindPath($x, y$) : return a path in the Reeb graph connecting nodes $x$ and $y$.

The function Path($u$) is implemented by first calling Descend($u$) and Ascend($u$) to find minimum $y$ and maximum $x$. These extrema lie on Jacobi edges which are used to find the corresponding Reeb graph nodes. Then, FindPath($x, y$) returns the required path in the Reeb graph.

I provide the sub-routine for Descend; Ascend is similar, and FindPath is implemented using a depth-first search of the Reeb graph. In the description I represent a vertex by an edge and time pair; slicing the edge at the time gives the vertex. The sub-routine repeatedly steps to a vertex that is in the lower link of the current vertex until it reaches a minimum.

```
Edge Descend(vertex $u$ = (Edge $E$, Time $t$))
  while($u$ is not a minimum)
    $L$ = GetLnk($E, t$);
    foreach vertex (v = (Edge $F$, Time $t$)) $\in \mathrm{Lk}(u)$
      if (GetBit($L, F$) = 0)
        $u = v$; breakfor;
  return($u$);
```

I describe how to use the Path function to distinguish between `Case 2a` and `Case 2b` in the interchange events shown in Figure 7.2, and to decide which arcs to remove and add when $x$ and $y$ swap in the Reeb graph.

When nodes $x$ and $y$ swap in $R_t$ their critical points in $P_t$ have the same function value and the lower link of $y$ has two components. Letting $u$ and $v$ be a vertex in each, we compute the arcs $a$ and $b$ incident to and below $x$ in $\textsc{Path}(u)$ and $\textsc{Path}(v)$. We have `Case 2a` iff $a \neq b$. In `Case 1` and `Case 2b`, I decide which arc below $x$ gets reconnected at $y$ by computing the arc $a$ incident to and below $x$ in $\textsc{Path}(u)$ and $\textsc{Path}(v)$. `Case 3` is upside down symmetric to `Case 1`.

## 7.5 Conclusion

I describe key pieces in the implementation for time-varying functions defined on $\mathbb{S}^2$. The algorithms described in this chapter can be extended to time-varying functions on $\mathbb{S}^3$. Because interchange times for edges in any spatial dimension are defined on the 2D function-time plane, the TIME data-structure with its comparison sub-routine directly extends to higher dimensions. To compute Jacobi sets in a higher spatial dimension one has to classify the transitions of the link of a vertex in that dimension. Once that is done the sweep algorithm to compute critical segments described in this chapter works. Unfolding Jacobi edges seems more difficult in higher dimensions; the link is a 2-sphere and it is not clear how to split degenerate saddle-segment while maintaining continuity.

# Chapter 8

# Hierarchical Presentation of Time-varying Reeb Graphs

Reeb graphs can be presented in visualization systems as a summary of data [BPS97], and as an interface for the selection of its level sets [CS03]. Their effectiveness in both roles in depends on a clear presentation which is hindered by their size. Reeb graphs of datasets common in medical imaging and simulation can contain thousands of nodes and arcs; without simplification their presentation is of little practical value.

Carr et al. [CSvdP04] simplify the Reeb graph of static data using local geometric measures, and successfully reduce its size while maintaining the essential features of many datasets. Pascucci et al. [PCMS04] develop a multi-resolution hierarchy of the Reeb graph to produce simplified versions depending on a user-specified threshold. They also compute a layout in space that guarantees no self-intersections for an uncluttered presentation. I wish to extend their multi-resolution hierarchy to time-varying Reeb graphs, and develop a presentation that is coherent over time.

## 8.1 Static Hierarchy and Presentation

I build a multi-resolution hierarchy and presentation for time-varying Reeb graphs by extending the hierarchy and presentation developed by Pascucci et al. [PCMS04] for static Reeb graphs, which I describe in this section.

Their hierarchy is computed from a *branch decomposition* of the Reeb graph.

**Definition 8.1.1 (Branch)** A *branch* is a monotone path in the Reeb graph traversing a sequence of nodes with non-decreasing (or non-increasing) function values.

The first and last nodes in the path are the *endpoints* of the branch; all other nodes are *interior* to the path. Interior nodes are saddle nodes; they can merge or split level sets, and add or remove a handle.

**Definition 8.1.2 (Branch Decomposition)** A set of branches is called a *branch decomposition* of the Reeb graph if every arc of the graph appears in exactly one branch of the set.

One branch decomposition is the standard representation of a graph as a list of nodes and a list of arcs – trivially, each arc can be considered a branch. A more interesting branch decomposition is hierarchical.

**Definition 8.1.3 (Hierarchical Branch Decomposition)** A branch decomposition of the Reeb graph is a *hierarchical branch decomposition* if

1. There is exactly one branch connecting two leaves of the Reeb graph. This branch is called the *root branch*;
2. Every other *child branch* is a branch that connects one leaf to an interior node of its *parent branch*.

A branch with no children is a *leaf branch*, and a non-root branch with children is an *internal branch*.

Figure 8.1 shows a example hierarchical branch decomposition.

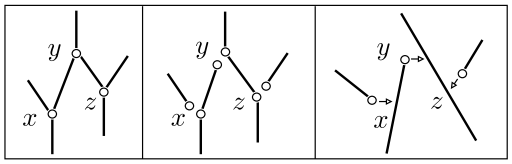


Figure 8.1: Left, a Reeb graph; middle, its branch decomposition. At right, we illustrate the branches by straight lines as in [PCMS04]. The arrows show where each branch attaches to its parent (we omit arrows in subsequent illustrations). Branches attached at $x$ and $z$ are leaf branches, and the branch attached at $y$ is an internal branch.

One can easily form a hierarchical branch decomposition from a standard representation of a Reeb graph [PCMS04]. First, we can suppress degree 2 nodes by allowing each arc to represent a monotone path. Then we can consider removing any leaf node $y$ whose parent $x$ connects to other nodes both above and below $x$. When we remove a leaf $y$, we suppress its parent $x$ if its degree is reduced to 2, and consider any other leaf incident to $x$ for removal. It is not difficult to maintain a queue with leaves that are candidates for removal, in $O(1)$ time apiece, giving a linear time algorithm.

I illustrate this algorithm in Figure 8.2.

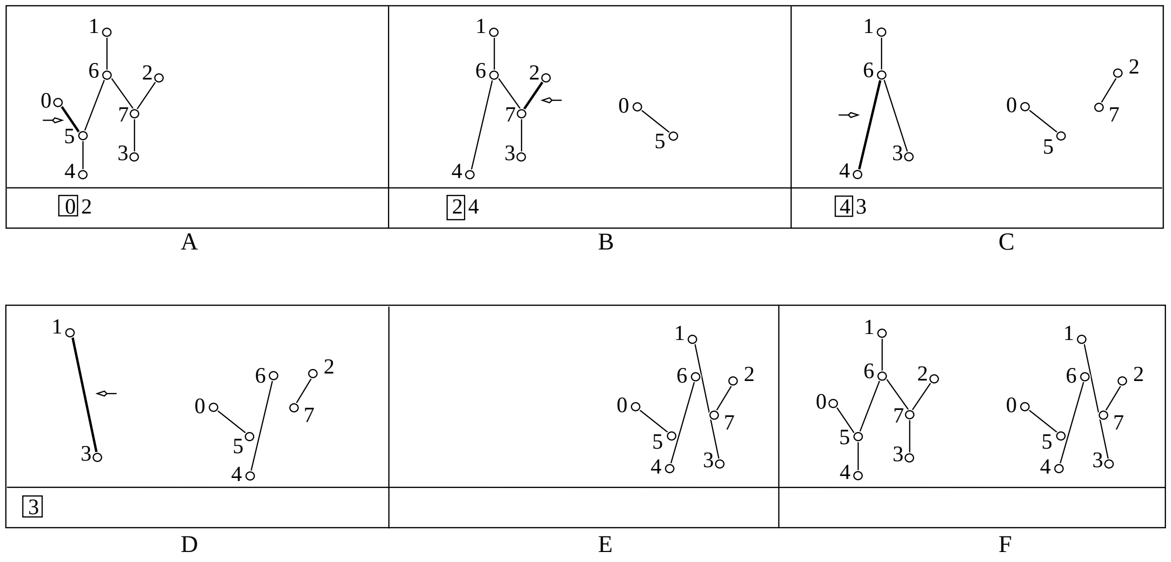


Figure 8.2: Computing the hierarchical branch decomposition of the tree in Figure 8.1. The queue of candidate leaves is shown under each frame, and the current candidate and its arc are shown in bold. Notice that, in the frame B, leaf node 4 becomes a candidate after node 0 is removed in the first frame. The last frame shows the Reeb graph and its hierarchical branch decomposition side-by-side.

Once we have computed a hierarchical branch decomposition we can compute a coarse-to-fine approximation to the Reeb graph by starting at the root branch and recursively including child branches. We attach a measure with each branch to enable cut-off at some user specified threshold. We can recursively define a *span* for each branch as the smallest interval containing the span of all its children and the interval of function values bounded by its endpoints. For a threshold $S$, a simplified branch decomposition that includes all branches with span $s \geq S$ is produced by traversing the hierarchy depth-first starting from the root branch and visiting a child branch if its span is larger than $S$.

For progressive, interactive presentation of large Reeb graphs we perform a breadth-first traversal; traverse the hierarchy and display branches with decreasing spans until we run out of time or the user changes the viewing parameters. Next, I list some goals of such a presentation, and describe an algorithm due to Pascucci et al. [PCMS04] that computes a presentation with these goals in mind.

**Presentation of Reeb graphs.** An effective visual presentation of a Reeb graph should: (i) highlight the separation of branches with few or no self-intersections for clarity; (ii) preserve the function values of the nodes (e.g. by setting their vertical coordinate equal to their function value); (iii) display arcs with large span that may potentially indicate important regions of the data and suppress (under user control) small and potentially noisy arcs (iv) remain stable during interaction when the user may navigate through coarser or finer approximations of the Reeb graph - successive presentations must be coherent so that the user retains context.

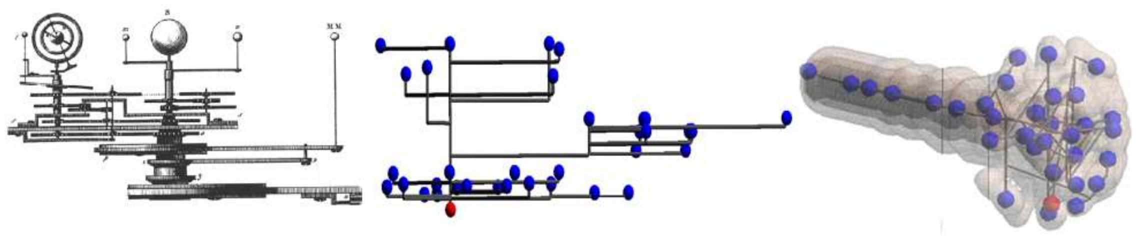


Figure 8.3: Left, an orrery presents the hierarchical relationship between the orbits of the sun, the planets and their moons. (Original design (1812) by A. Janvier reprinted by E. Tufte [Tuf90].) Center, a Reeb graph drawn as an orrery with celestial bodies replaced by nodes, and orbits replaced by arcs. Right, the inverse air density distribution of a fuel injected into a combustion chamber with its Reeb graph, displayed as several transparently rendered level sets and the critical points and their arcs from the Reeb graph embedded in the same space. From [PCMS04]

Pascucci et al. [PCMS04] compute a three-dimensional embedding of the hierarchical branch decomposition of a Reeb graph that is inspired by the orrery, which presents the hierarchical relationship between the sun, the planets, and their moons as shown in Figure 8.3. Their presentation computes a radial layout of graphs [Ber81, Ead92, dBETT99, YFDH01] and achieves many of the goals set forth earlier; there are no self-intersections of arcs, the function values of nodes are mapped to one of its spatial coordinates, and the presentations of successively coarse or fine approximations of the Reeb graph are coherent. I describe how to compute this presentation.

We have a hierarchical branch decomposition in which each node has a natural $z$ coordinate, and we need to define its $x$ and $y$ coordinates. We do this so each branch is an L-shaped path (maybe upside down), with all of its nodes on the vertical, except for its non-leaf endpoint. Furthermore, all $x$ and $y$ coordinates will lie on concentric circles according to their depth in the branch decomposition.

Before we give a precise definition of the "orrery presentation" of a hierarchical branch decomposition, we attach two numbers to each branch $b$.

The *hierarchy depth* for branch $b$ is the number of times we have to take the parent branch to reach the root branch $r$; that is, if that $r = \mathit{parent}^k(b)$, then $\mathit{hdepth}(b) = k$. Thus, $\mathit{hdepth}(r) = 0$. The depth of a hierarchy is the maximum *hdepth* of its nodes.

The *subbranch size* for branch $b$ is the number of branches that have $b$ as an ancestor; that is, $\mathit{subbranch}(b) = |\{c : \exists i \geq 0 \text{ such that } b = \mathit{parent}^i(c)\}|$. Thus, for any leaf $d$ $\mathit{subbranch}(d) = 1$, and for the root $\mathit{subbranch}(r) =$ the total number of branches.

We lay nodes out on concentric circles, choosing the circle for a node by its hierarchy depth, and its placement on an arc by its subbranch size relative to its parent's. See Figure 8.4. Both hierarchy depth and subbranch size can easily be computed in linear time by traversing the hierarchy.

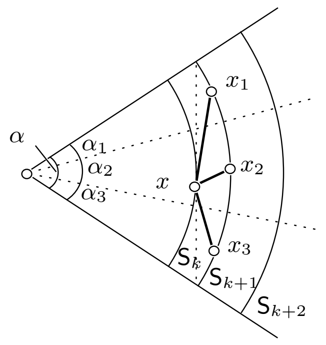


Figure 8.4: Defining wedge angle nodes projected on the plane. The nodes $x_1$, $x_2$, and $x_3$ connected to $x$ are each assigned a portion of $x$'s wedge angle $\alpha$ proportional to their fraction of its descendants.

The orrery presentation can now be defined recursively, as follows. The root branch is presented as a vertical line segment along the $z$ axis, with each vertex $v$ appearing at its natural $z = f(v)$ coordinate. Each child branch $c$ is assigned an angular wedge of $360 \cdot \mathit{subbranch}(c)/(\mathit{subbranch}(r) - 1)$ degrees around the origin so that the wedges assigned cover the $xy$ plane.

In the recursion, place the nodes of a branch $b$ to project onto the circle with index $\mathit{hdepth}(b)$, at the center of its angular wedge. The exception is the node shared with $\mathit{parent}(b)$, which was previously placed. (Each node $v$ gets its natural $z = f(v)$ coordinate, as mentioned above). If $b$ is not a leaf branch, give each child $c$ its fraction $\mathit{subbranch}(c)/(\mathit{subbranch}(b) - 1)$ of the angular wedge of $b$ and recursively place each child.

This construction can be projected to the $xy$ plane with no self-intersections, since each subbranch of the hierarchy is assigned to its own angular wedge.

Figure 8.5 shows the steps to construct the presentation for a Reeb graph. This particular graph cannot be drawn in the plane without self-intersections; it must be drawn in three-dimensional space.

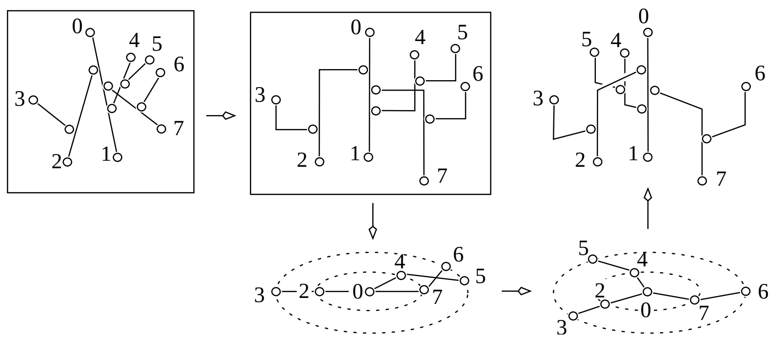


Figure 8.5: Orrery-like presentation of the hierarchical branch decomposition.

By computing a presentation for the entire hierarchical branch decomposition of the Reeb graph we are also computing presentations for all its approximations. Therefore the position of the branches in the presentation of a coarse approximation of the Reeb graph do not change as more branches are include to present a finer approximation; the result is a coherent presentation.

## 8.2 Maintaining a Hierarchical Branch Decomposition over Time

If we maintain a hierarchical branch decomposition for the Reeb graph $R_t$ over time, we can represent multiple approximations, and present them coherently for each $R_t$. Moreover it desirable to maintain a stable presentation over time; a Reeb graph that changes little over a short period of time must be presented coherently over that period. There are some challenges in maintaining a hierarchy and in maintaining a stable presentation over time.

To maintain the hierarchy we must develop a set of functions that modify the branch decomposition when the Reeb graph changes; the classification of changes to the Reeb graph will guide us.

We would also like to maintain the span of each branch of the hierarchy to generate approximations of the Reeb graph. Because the span, which is a function interval, can grow and shrink in between structural changes to the Reeb graph we prefer not to maintain the span over time, but recompute it whenever needed.

To maintain a stable presentation over time we must dynamically shrink and grow wedge angles as the Reeb graph changes. Recall that the wedge angle assigned to a branch is proportional to its fraction of the subtree rooted at its parent. If the subtree rooted at a node changes, all its siblings' wedge angles must be recalculated along with its own wedge angle, and the recalculation can spread to the entire tree. We choose to recalculate the presentation when required. If we can maintain the hierarchy by modifying it locally at each change, then there is some hope that it will be coherent over time, and its presentation might also be stable over time.

In this section, I define functions that modify a hierarchical branch decomposition, and describe how to use them at each event that changes the Reeb graph to maintain the hierarchy over time.

**Modifying the branch decomposition.** It is possible to list a set of functions that modify a hierarchical branch decomposition by considering how the Reeb graph changes over time.

A pair of nodes connected by an arc can appear and disappear; we need functions to create, attach and detach branches, and nodes.

AttachNode$(x, b, y)$ : Attach node $x$ onto branch $b$ above node $y$.

DetachNode$(x)$ : Detach the degree 2 node $x$ from its branch.

MkBranch$(x, y)$ : Make a branch connecting nodes $x$ and $y$.

AttachBranch$(c, b, x)$ : Attach branch $c$ to branch $b$ just above node $x$. Branch $c$ cannot be the root branch, node $x$ must be on branch $b$. Recall that branches are attached at their saddle nodes.

DetachBranch$(c)$ : Detach branch $c$ from its parent.

A pair of nodes connected by an arc can swap order in the Reeb graph; there can two consequences for the hierarchy when this happens. (i)Branches must be detached and reattached elsewhere; use the DetachBranch and AttachBranch functions. (ii) A branch can become invalid because the path it maps to in the Reeb graph is no longer monotone; break the branch at one of the nodes that swap and re-route the path to make it monotone again. The choice of where to break and how to re-route is made depending on the type of swap, and will be explained in detail soon. Once we have re-routed a path to create a valid branch we detach and reattach arcs as before.

<span style="font-variant:small-caps">ReRoute</span>$(b, c)$ : Break branch $b$ at the attaching node of child $c$ and merge one of the pieces with $c$ to form a new branch. The chosen piece forms a monotone path when merged with $c$. The remaining piece becomes a child of the new branch.

Before we can explain how to use these functions for each type of change to the Reeb graph at a birth-death and interchange event we will define two more functions; one helper to make our description concise, and one function to handle a special requirement of interchange case 2a.

MvAbove$(x, y)$ : Move node $x$ with any subtree attached there immediately above node $y$. The branches attached at $x$ and $y$ are now siblings. This function is a concise placeholder for the sequence of functions to detach and attach branches and nodes required to perform this modification.

SwapSubtrees$(x, y)$ : Swap the subtrees attached at nodes $x$ and $y$. The node identities are not swapped. This function is required specially for interchange case 2a.

Figure 8.6 shows the action of ReRoute, MvAbove, and SwapSubtrees.

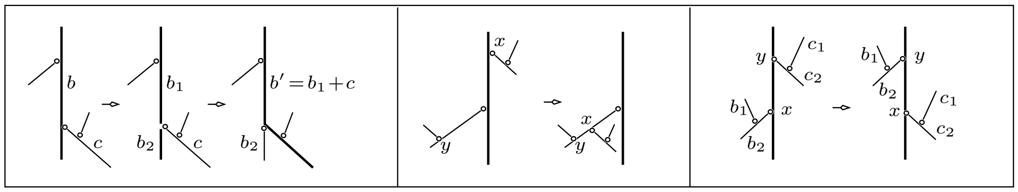


Figure 8.6: Left, ReRoute$(b, c)$ splits branch $b$ into $b_1$ and $b_2$ at child branch $c$'s attaching node, and creates branch $b'$ by merging $b_1$ with branch $c$. The piece $b_2$ now becomes a child branch of $b'$. Middle, MvAbove$(x, y)$ detaches node $x$ along with the subtree attached there and reattaches it just above node $y$. Right, SwapSubtrees$(x, y)$ swaps the subtrees attached at nodes $x$ and $y$; the node labels are not swapped.

**Maintaining the hierarchy.** We can now use the functions just described to modify the hierarchical branch decomposition at each birth-death and interchange event and maintain it over time.

To handle a 0-1 and 2-3 birth, we invoke MkBranch to create a new branch, and invoke AttachBranch to attach the new branch at the appropriate position on an existing branch. To handle a 1-2 birth, we invoke AttachNode to attach the new nodes and refine a branch. Similarly, we invoke the DetachBranch function for a 0-1 and 2-3 death, and invoke DetachNode to the coarsen a branch for a 1-2 death.

Handling interchange events requires a little more effort as described next. Figure 8.7 illustrates all cases except symmetric ones. Each interchange case (including interchange subcases) can result in several hierarchy subcases depending on the branch decomposition. We enumerate these subcases in the description below.

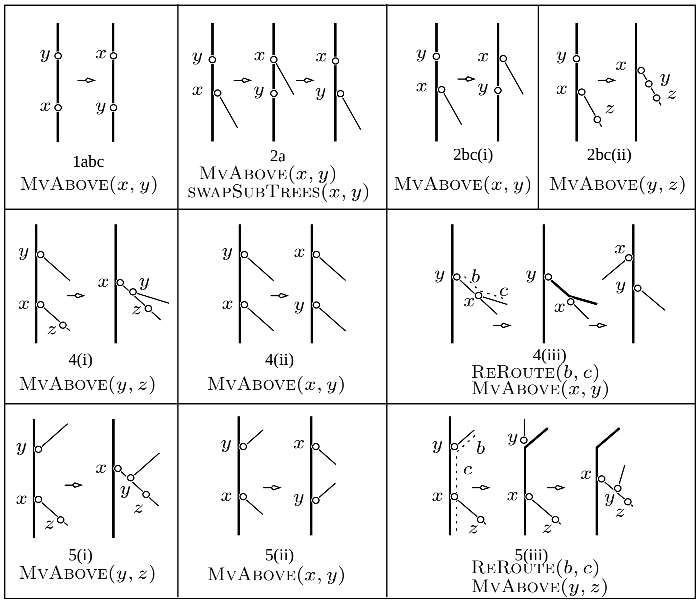


Figure 8.7: Maintaining the hierarchical branch decomposition at an interchange case. We handle each case by invoking the functions displayed below its picture. Cases `3` and `6` are upside-down symmetric to cases `2` and `4` respectively, and are not illustrated.

`Case 1abc` We handle all the subcases by invoking MvAbove$(x, y)$.

`Case 2a` We need to swap nodes $x$ and $y$; invoke MvAbove$(x, y)$ followed by SwapSubtrees$(x, y)$. We need the call to SwapSubtrees because MvAbove moves the subtree with its node.

`Case 2bc` There can be two possible subcases depending on where node $y$ moves. If node $y$ moves below node $x$ but onto the same branch as before then we have subcase `(i)`; invoke MvAbove$(x, y)$. If node $y$ moves to the branch attached at $x$ then we have subcase `(ii)`. Letting $z$ be the first node below $x$ on its branch, we invoke MvAbove$(y, z)$.

`Case 3abc` These cases are upside down symmetric to `Case 2abc`, and are handled similarly.

`Case 4` The subcases can be enumerated by considering how the branches at $x$ and $y$ are attached to their respective parent branch.

The branches at $x$ and $y$ could be attached to the same parent branch giving us cases `4(i)` and `4(ii)`. In `4(i)` node $y$ moves onto the branch attached at $y$; invoke MvAbove$(y, z)$. In `4(ii)` node $y$ moves below $x$ but onto the same branch as before; invoke MvAbove$(x, y)$.

On the other hand, node $x$'s parent branch could be the branch attached at node $y$. If after the interchange, the branch attached at $x$ corresponds to the Reeb graph arc that moves with $x$ then we have the reverse of `4(i)`; invoke MvAbove$(x, y)$. We have a more interesting case if the arc that moves with $x$ corresponds to the portion of the parent branch of $x$ below where $x$ is attached. This configuration is `Case 4(iii)`. As shown in Figure 8.7, invoke ReRoute$(b, c)$ to split the parent branch and create a new branch that includes the branch attached at $x$. We now have a configuration similar to the reverse of `4(i)`, and invoke MvAbove$(x, y)$ to complete the interchange.

`Case 5` In this case, branches at $x$ and $y$ are attached to the same parent node. We have two simple cases, node $y$ with its branch moves onto the branch at node $x$ in `5(i)`, or onto the same branch as before but below node $x$ in `5(ii)`. We have a symmetric case if node $x$ and its branch moves onto the branch at node $y$. Similar to `Case 4(iii)`, `Case 5(iii)` is an interchange when the Reeb graph arc that moves with node $y$ corresponds to the upper portion of $y$'s parent branch; invoke ReRoute$(b, c)$ to break the parent and reduce it to one of `Case 5(i)` or `Case 5(ii)`. We can have a symmetric case that requires splitting the parent branch at node $x$.

`Case 6` This case is upside down symmetric to `Case 4`.

In Figure 8.8, we see the orrery-like presentation of the Reeb graph of the Combust dataset at three time-steps along with the data displayed as a terrain.

## 8.3 Conclusion

In this chapter, I have discussed a hierarchical organization of the Reeb graph that can provide multiple approximations, and present them coherently. I maintain this hierarchy over time for time-varying Reeb graphs by modifying it at the times when the Reeb graph changes. I hope that maintaining a coherent hierarchy will result in a coherent presentation.

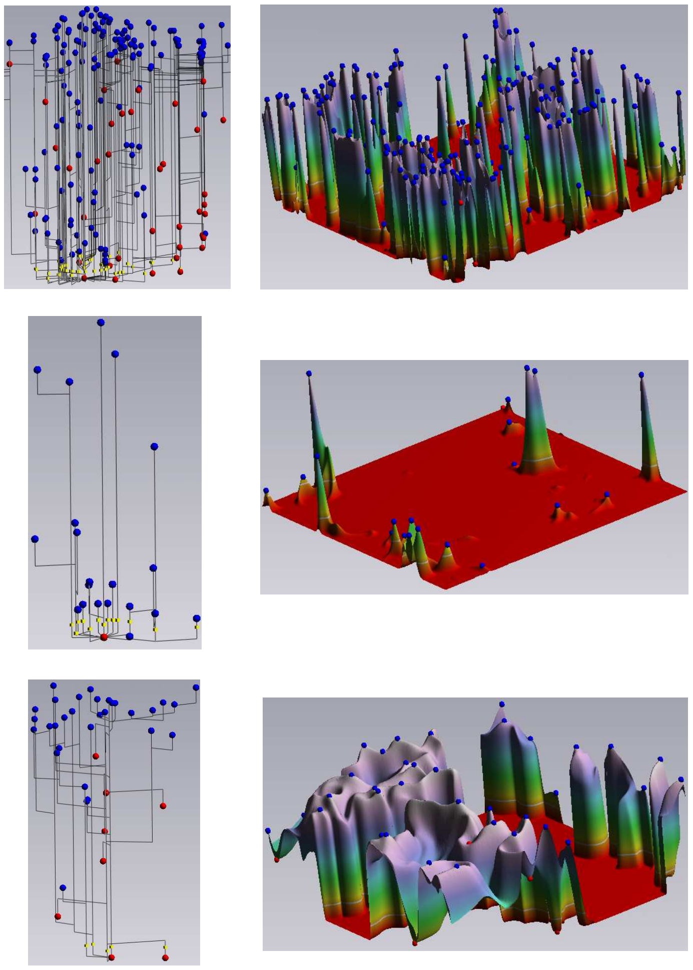


Figure 8.8: Screenshots from the presentation of the Reeb graphs of Combust at times 0 (top), 20 (middle), and 66 (bottom) along with the data shown as a terrain.

As future work, I would like to study the stability of the evolving presentation; are successive presentations coherent over time? Can small local changes in the Reeb graph lead to large-scale global changes in the presentation? Can we maintain a lazy presentation that does not change wedge angles for every modification and allows over-crowding within a wedge upto a threshold? Does knowing the time-varying Reeb graph in advance assist in choosing the branch decomposition so that the presentation is relatively stable? I hope that the answers to these questions will ultimately result in a more coherent presentation that will improve the effectiveness of time-varying Reeb graphs as a tool to analyze data.

# Chapter 9

# Conclusion

In this dissertation, I develop time-varying Reeb graphs as a framework to support the analysis of time-varying data. Time-varying Reeb graphs extend the benefits of static Reeb graphs to the analysis of time-varying data by recording information that encodes the topology of level sets of all level values at all times, that maintains the correspondence between level set components over time, and that accelerates the extraction of level sets for a chosen level value and time.

I rigorously define time-varying Reeb graphs using Morse theory and smooth functions, and establish a connection between Reeb graphs and Jacobi sets, which is the path traced by critical points of a time-varying functions. Using this connection, and by translating concepts from the smooth world of mathematics to the continuous, but not always smooth world of interpolated scientific datasets, I develop a robust combinatorial algorithm to compute time-varying Reeb graphs.

Because different interpolants produce Jacobi sets that approximate the structure of smooth Jacobi sets to different degrees, I evaluate which interpolant produces Jacobi sets consistent with the smooth case.

I evaluate piecewise linear, piecewise trilinear, and piecewise prismatic interpolants for scientific data sampled on a regular grid, and show that piecewise prismatic is the best suited for constructing Jacobi sets and time-varying Reeb graphs.

I present preliminary results on how to present time-varying Reeb graphs over time, with simplification and coherence between simplifications. I envision such a presentation as crucial to the success of a visualization system that supports the analysis of time-varying scientific data.

I close with some promising areas of future work, and open questions.

- The prismatic interpolant that I recommend applies only to samples that do not change their spatial position over time. The PL interpolant applies to general space-time samples but produces Jacobi sets that are difficult to unfold into a 1-manifold and do not produce a simple and robust construction of time-varying Reeb graphs. What interpolants for general space-time samples produce Jacobi sets that either are 1-manifolds or can be unfolded into 1-manifolds and produce simple and robust constructions of time-varying Reeb graphs?

- Many scientific datasets are inherently noisy. Such noise can produce an overwhelming number of critical points and make the construction of Jacobi sets and time-varying Reeb graphs infeasible.

  One could apply a low-pass filter to the data in a pre-processng stage, but it is possible that filtering the data loses important features. It is therefore important to allow the scientist to select features to preserve and control the de-noising process. Topological de-noising techniques [EHZ03, ELZ02] provide such control and work well for static data. How can we extend them to de-noise time-varying data? A possible starting point is extending the concept of topological persistence [ELZ02] to time-varying data.

- Constructing Jacobi sets and time-varying Reeb graphs for many large scientific datasets is challenging. Their sheer size increases running time and requires careful management of main memory and disk storage.

  It is possible to reduce running time by parallelizing the construction of Jacobi sets and time-varying Reeb graphs. The edges in a Jacobi set can be computed in parallel because testing if a point is critical is a local operation. The time-varying Reeb graph can be constructed in parallel for intervals of time and stitched at interval boundaries to produce the time-varying Reeb graph for the entire time span.

  It is important to reduce expensive disk access while working with large datasets that do not fit into main memory. The data access pattern during the construction of time-varying Reeb graphs may not be coherent because successive events that change the Reeb graph can access data in disparate spatial locations. I/O efficient techniques often change data layout to reduce disk access [AE99]. Can we use these techniques to develop I/O efficient versions of the algorithms discussed in this dissertation? Can we adapt the construction of time-varying Reeb graphs to work on a streaming data model?

- In this dissertation I maintain a hierarchical branch decomposition and recompute a presentation whenever required. Preliminary experiments suggests that a coherent visual presentation of Reeb graphs over time is not always possible even if the underlying hierarchical branch decomposition is coherent. A possible solution is to maintain a presentation over time. How can we extend the techniques used to maintain the hierarchy over time to maintain the layout over time?

# BIBLIOGRAPHY

[AE99] J. Abello and J. Vitter (Eds.). External memory algorithms and visualization. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Providence, Rhode Island, 1999. American Mathematical Society Press,.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[Ale98] P. S. Alexandrov. *Combinatorial Topology.* Dover, Mineola, New York, 1998.

[Art79] E. Artzy. Display of three-dimensional information in computed tomography. In *Computer Graphics and Image Processing*, volume 9, pages 196–198, 1979.

[AS95] P. K. Agarwal and M. Sharir. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, 1995.

[Ban70] T. F. Banchoff. Critical points for embedded polyhedral surfaces. In *Amer. Math. Monthly*, volume 77, pages 457–485, 1970.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM*, volume 18(9), pages 509–517, 1975.

[Ber81] M. A. Bernard. On the automated drawing of graphs. In *Proc. of third Caribbean Conf. on Combinatorics and Computing*, pages 43 – 55, 1981.

[Bli82] J. F. Blinn. A generalization of algebraic surface drawing. In *ACM Transactions on Graphics*, volume 1(3), pages 235–256, 1982.

[BPS97] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel Schikore. The contour spectrum. In *IEEE Visualization*, pages 167–174, 1997.

[BR63] R. L. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proc. of 1963 Fall Joint Computer Conference (IEEE)*, pages 445–458, 1963.

[BS04] Nathanael Berglund and Andrzej Szymczak. Making contour trees subdomain-aware. In *CCCG*, pages 188–191, 2004.

[BSBS02] C. L. Bajaj, A. Shamir, and S. Bong-Soo. Progressive tracking of isosurfaces in time-varying scalar fields. Technical report, Univ. of Texas, Austin, 2002. http://www.ticam.utexas.edu/CCV/papers/Bongbong-Vis02.pdf.

[BSRF00] K. G. Bemis, D. Silver, P. A. Rona, and C. Feng. Case study: a methodology for plume visualization with application to real-time acquisition and navigation. In *Proc. IEEE Conf. Visualization*, pages 481–494, 2000.

[CE97] M. Cox and D. Ellsworth. Application controlled demand paging for out-of-core visualization. In *IEEE Proc. of Vis. '97*, pages 235–244, 1997.

[Chi03] Y.-J Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proc. IEEE Visualization 2003*, pages 217–224, 2003.

[CLR94] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1994.

[CM97] T. Chiueh and K-L. Ma. A parallel pipelined renderer for time-varying volume data. In *Proc of Parallel Architecture, Algorithms, Networks*, pages 9–15, 1997.

[CMEH$^+$03] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in Reeb graphs of 2-manifolds. In *Proc. 14th Ann. Sympos. Comput. Geom.*, pages 344–350, 2003.

[CMM$^+$97] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Trans. on Vis. and Computer Graphics*, 3(2):158–170, /1997.

[CS03] Hamish Carr and Jack Snoeyink. Path seeds and flexible isosurfaces: Using topology for exploratory visualization. In *Proc. of Eurographics Visualization Symposium*, pages 49–58, 285, 2003.

[CSA03] Hamish Carr, Jack Snoeyink, and Ulrike Axen. Computing contour trees in all dimensions. In *Computational Geometry*, volume 24(2), pages 75–94, 2003.

[CSS98] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. of the Symp. for Volume Vis.*, pages 167–174, 1998.

[CSvdP04] Hamish Carr, Jack Snoeyink, and Michael van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proc. of IEEE Visualization 2004*, pages 497–504, 2004.

[dBETT99] Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[DE95] C. J. A. Delfinado and H. Edelsbrunner. An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere. In *Comput. Aided Geom. Design*, volume 12, pages 771–784, 1995.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *J. Comput. Sys. Sci*, volume 38, pages 86–124, 1989.

[Dür88] Martin J. Dürst. Additional reference to "marching cubes". *SIGGRAPH Comput. Graph.*, 22(5):243, 1988.

[Ead92] P. D. Eades. Drawing free trees. In *Bulletin of the Institute for Combinatorics and its Applications*, volume 5, pages 10 – 36, 1992.

[EC01] T. Echekki and J. Chen. Direct numerical simulation of auto-ignition in inhomogeneous hydrogen-air mixtures. In *Combustion and Flame*, volume 134(3), pages 169–191. Elsevier Science, August 2001.

[EH02] H. Edelsbrunner and J. Harer. Jacobi sets of multiple morse functions. In F. Cucker, R. DeVore, P. Olver, and E. Sueli, editors, *Foundations of Computational Mathematics*, pages 37–57. Cambridge Univ. Press, England, 2002.

[EHNP03] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Morse-Smale complexes for piecewise linear 3-manifolds. In *Proc. 19th Ann. Sympos. Comput. Geom.*, pages 361–370, 2003.

[EHZ03] Herbert Edelsbrunner, John Harer, and Afra Zomorodian. Hierarchical morse complexes for piecewise linear 2-manifolds. In *Discrete and Computational Geometry*, volume 30(1), pages 87–107. Springer-Verlag, 2003.

[ELZ02] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Discrete Computational Geometry.*, volume 28, pages 511–533, 2002.

[EM90] H. Edelsbrunner and E.P̃. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. In *ACM Trans. Graphics*, volume 9, pages 66–104, 1990.

[FKU77] H. Fuchs, Z. Kedem, and S. Uselton. Optimal surface reconstruction from planar contours. In *Communications of the ACM*, volume 20, pages 693–702, 1977.

[FTLK97] A. T. Fomenko and (eds) T. L. Kunii. *Topological Methods for Visualization.* Springer-Verlag, Tokyo, Japan, 1997.

[Gal91] R. S. Gallagher. Span filtering: An efficient scheme for volume visualization of large finite element models. In G. M. Neilson and L. Rosenblum, editors, *Proc. of Vis. '91*, pages 68–75, Oct 1991.

[GG73] M. Golubitsky and V. Guillemin. *Stable mappings and their singularities.* Springer-Verlag, New York, 1973. Graduate Texts in Mathematics, Vol. 14.

[GW94] Allen Van Gelder and Jane Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375, 1994.

[Han90] P. Hanrahan. Three-pass affine transforms for volume rendering. In *Computer Graphics*, volume 24(5), pages 71–78, 1990.

[HB94] C. T. Howie and E. H. Black. The mesh propagation algorithm for isosurface construction. In *Computer Graphics Forum 13, Eurographics '94 Conf. Issue*, pages 65–74, 1994.

[HL79] G. T. Herman and H. K. Lun. Three-dimensional display of human organs from computed tomograms. In *Computer Graphics and Image Processing*, volume 9, pages 1–21, 1979.

[IG03] Martin Isenburg and Stefan Gumhold. Out-of-core compression for gigantic polygon meshes. In *Proc. of SIGGRAPH 2003*, pages 935–942, July 2003.

[IL04] M. Isenburg and P. Lindstrom. Streaming meshes. In *Manuscript*, April 2004.

[ILGS03] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In *Proc. of Vis. 2003*, pages 465–472, Oct 2003.

[KH84] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In *Computer Graphics*, volume 18(3), pages 165–174, 1984.

[KRS03] Lutz Kettner, Jarek Rossignac, and Jack Snoeyink. The safari interface for visualizing time-dependent volume data using iso-surfaces and contour spectra. *Comput. Geom. Theory Appl.*, 25(1-2):97–116, 2003.

[KS86] A. Kaufman and E. Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *1986 Workshop on Interactive 3D Graphics*, pages 45–75, 1986.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proc.)*, volume 21, pages 163–169, July 1987.

[Lev90] M. Levoy. Efficient ray tracing of volume data. In *ACM Transactions on Graphics*, volume 9(3), pages 245–261, 1990.

[LSJ96] Y. Livnat, H. W. Shen, and C. R. Johnson. A near optimal iso-surface extraction algorithm for unstructured grids. In *IEEE Trans. on Vis. and Computer Graphics*, volume 2(1), pages 73–84, 1996.

[Mat94] Sergey V. Matveyev. Approximation of isosurface in the marching cube: ambiguity problem. In *Proceedings of the Conference on Visualization '94*, pages 288–292. IEEE Computer Society Press, 1994.

[Mat02] Y. Matsumoto. *An Introduction to Morse Theory (Translated from Japanese by K. Hudson and M. Saito).* American Mathematical Society, 2002.

[MCW00] N. Max, R. Crawfis, and D. Williams. Visualization for climate modeling. In *IEEE Computer Graphics Applications*, pages 481–494, 2000.

[MDB87] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), 1987.

[Mil63] J. Milnor. *Morse Theory.* Princeton Univ. Press, New Jersey, 1963.

[MSS94] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *Proceedings of the Conference on Visualization '94*, pages 281–287. IEEE Computer Society Press, 1994.

[Mun84] J. R. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, Redwood City, California, 1984.

[Nat94] B. K. Natarajan. On generating topologically consistent isosurfaces from uniform samples. *Vis. Comput.*, 11(1):52–62, 1994.

[NH91] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceedings of the 2nd Conference on Visualization '91*, pages 83–91. IEEE Computer Society Press, 1991.

[PCM03] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. In *Algorithmica*, volume 38(1), pages 249–268, 2003.

[PCMS04] Valerio Pascucci, Kree Cole-McLaughlin, and Giorgio Scorzelli. Multi-resolution computation and presentation of contour trees. In *Proc. IASTED Conference on Visualization, Imaging, and Image Processing*, pages 452–290, 2004.

[Ree46] G. Reeb. Sur les points singuliers d'une forme de pfaff complèment intégrable ou d'une fonction numérique. In *Comptes Rendus de L'Académie ses Séances, Paris*, volume 222, pages 847–849, 1946.

[SB05] Bong-Soo Sohn and Chandrajit L. Bajaj. Time-varying contour topology. In *IEEE Transactions on Visualization and Computer Graphics*, volume 12 (1), pages 14–25, 2005.

[SH99] P. Sutton and C. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree(T-BON). In *IEEE Proc. of Vis. '99*, pages 147–153, 1999.

[She98] H. W. Shen. Iso-surface extraction in time-varying fields using a temporal hierarchical index tree. In *IEEE Proc. of Vis. '98*, pages 159–166, Oct 1998.

[SHLJ96] H. W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proc. of Vis. '96*, pages 287–294, 1996.

[SK91] Y. Shinagawa and T. L. Kunii. Constructing a Reeb graph automatically from cross sections. In *IEEE Comput. Graphics Appl.*, volume 11, pages 44–51, 1991.

[Szy05] Andrzej Szymczak. Subdomain aware contour trees and contour evolution in time-dependent scalar fields. In *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI' 05)*, pages 136–144, Washington, DC, USA, 2005. IEEE Computer Society.

[TO91] Nils Thune and Bjorn Olstad. Visualizing 4-d medical ultrasound data. In *Proceedings of the 2nd Conference on Visualization '91*, pages 210–215. IEEE Computer Society Press, 1991.

[Tuf90] E. R. Tufte. *Envisioning Information.* Graphics Press LLC, Cheshire, Connecticut, 1990.

[vK94] M. van Kreveld. Efficient methods for isoline extraction from digital elevation model based on triangulated irregular networks. In *Sixth Inter. Symp. on Spatial Data Handling*, pages 835–847, 1994.

[vKvOB$^+$97] M. van Kreveld, R. von Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for iso-surface traversal. In *The 13th ACM Sym. on Computational Geometry*, pages 212–220, 1997.

[Wes89] L. Westover. Interactive volume rendering. In *Chapel Hill Workshop on Volume Visualization*, pages 9–16, 1989.

[WG92] J. Wilhelms and Van Gelder. Octrees for faster isosurface generation. In *ACM Trans. on Graphics*, volume 11(3), pages 201–227, 1992.

[WMW86a] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating soft objects. *Visual Computer*, 2:235–242, 1986.

[WMW86b] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *Visual Computer*, 2:227–234, 1986.

[YFDH01] Ka-Ping Yee, Danyel Fisher, Rachna Dhamija, and Marti Hearst. Animated exploration of dynamic graphs with radial layout. In *IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, pages 43–50, 2001.