Doctoral Thesis

Efficient algorithms for Lagrangian visualization of flow structures

Author(s): Schindler, Benjamin

Publication Date: 2013

Permanent Link:
https://doi.org/10.3929/ethz-a-009988701

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
EFFICIENT ALGORITHMS FOR LAGRANGIAN VISUALIZATION OF FLOW STRUCTURES

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
BENJAMIN SCHINDLER
MSc Computer Science,
ETH Zurich

born May 10th, 1983
Citizen of Zurich, Switzerland

accepted on the recommendation of

Prof. Dr. Ronny Peikert, examiner
Prof. Dr. Renato Pajarola, co-examiner
Prof. Dr. Olga Sorkine-Hornung, co-examiner
Prof. Dr. Holger Theisel, co-examiner
ACKNOWLEDGEMENTS

First, I would like to thank Ronny Peikert for taking me as a Ph.D. candidate. His door was always open for discussions or when problems arose and he was always able to spare time to help out, something I’ve learned from other colleagues one can not always take for granted. The balance achieved in directing me into a certain direction while letting me do self-motivated research was very healthy.

I want to thank the entire Visdom [2] crew. In no particular order, Raphael Fuchs, for convincing me on using Visdom for my Ph.D. work, even though I had no idea what I was up to. Jürgen Waser and Hrvoje Ribičić for the countless hours of stimulating discussions, support, bug fixing and of course the friendly atmosphere on visits to Vienna. The enduring close cooperation with another institute was a very positive experience for me. Robert Carnecky for the many fruitful discussions and for all the support on rendering in Visdom. Also, I want to thank my office colleagues, Robert Carnecky and Christian Heine. Without them, being at ETH would have been a lot more boring. The people at AFC (Air Flow Consulting), Stefan Barp, Daniel Gubler and Matthias Röthlin also deserve a special mention for providing a challenging application case with their revolving door simulation. Matthias Röthlin in particular for rerunning the simulation with the proposed modifications as this was key for getting this project published and presented at the IEEE Vis conference.

A special thank you goes to Jürgen Waser, Basil Fierz and Christian Heine for very helpful and detailed proofreading. Finally I would like to thank my wife Silja, who always supported me and even endured the times, when my home was rather located in the office than at our apartment.

This work was funded by the Swiss National Science Foundation (SNF) under grants 200021_127022 and 200020_140556 (SPHVis++ and extension) and by Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number 226042.
## CONTENTS

1 INTRODUCTION 1
  1.1 Contributions and Layout of this Thesis 3
  1.2 Publications 3

i VISUALIZATION OF SPH DATA 5

2 SPH FUNDAMENTALS 7

3 CONTOURING 11

4 POLYGONAL ISOSURFACE EXTRACTION FROM SPH DATA 13
  4.1 Algorithm Description 13
    4.1.1 Spatial Data Structure 13
    4.1.2 Virtual Grid 14
    4.1.3 Seed Cell Selection 15
    4.1.4 Surface Extraction 16
  4.2 Results 19
    4.2.1 Tsunami Data Set 19
    4.2.2 Dam Break Data Set 20
    4.2.3 Vortex Rings Data Set 21
    4.2.4 SPH Kernel Size 21
    4.2.5 Virtual Grid Size 22
    4.2.6 Effectiveness of the Correction Step 23

5 DISCUSSION 27

ii VISUALIZATION OF LCS 29

6 BACKGROUND 31
  6.1 Classical Vector Field Topology 31
  6.2 LCS and FTLE 32

7 RIDGE CONCEPTS 35
  7.1 Watersheds, Height Ridges, and Section-based Ridges 36
  7.2 Tensor Field Lines 36
  7.3 C-Ridges 37
  7.4 Error Metric for the Material Line/Surface Property 38
  7.5 Results 39
    7.5.1 “Double Gyre” Example 39
    7.5.2 Square Cylinder Example 42
    7.5.3 Francis Turbine Example 45
  7.6 Appendix: Overdetermined Ridge Definitions 47

8 RIDGE EXTRACTION 49
  8.1 Computation of the Flow Map 49
  8.2 Computation of FTLE 50
  8.3 Ridge Extraction 52
  8.4 Ridge Filtering 52
  8.5 Performance Analysis 56
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFD</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FTLE</td>
<td>Finite-Time Lyapunov Exponent</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>LCS</td>
<td>Lagrangian Coherent Structures</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation</td>
</tr>
<tr>
<td>SPH</td>
<td>Smoothed Particle Hydrodynamics</td>
</tr>
<tr>
<td>VFT</td>
<td>Vector Field Topology</td>
</tr>
<tr>
<td>VTK</td>
<td>Visualization Toolkit</td>
</tr>
</tbody>
</table>
ZUSAMMENFASSUNG


Normalerweise wählt man für Simulation eine dieser Beschreibungen ohne in Betracht zu ziehen, wie man die Resultate visualisieren möchte. Je nach Visualisierung ist aber die eine oder andere Beschreibung von Vorteil, was bedingt, dass man von der einen zur anderen Beschreibung wechseln muss. Diese Konvertierung braucht jedoch viel Rechenleistung. Die Wahl der Algorithmen ist daher enorm wichtig, um eine angemessene Effizienz zu erreichen.

ABSTRACT

The visualization of fluids is a very useful tool for science and engineering applications alike. A very central aspect of flow visualization as well as simulation is the representation of the fluid data. There are fundamentally two ways of representing flow fields – Lagrangian or Eulerian. For efficiency reasons, the Eulerian approach was typically favored. However, there are many scenarios, where the Lagrangian approach is more natural. With the availability of processing power nowadays, using Lagrangian techniques is becoming increasingly popular.

The choice of representation for a simulation is typically made without considering the type of visualization applied to the results. For visualization, it therefore becomes often necessary to convert the data from one representation to the other. This conversion is computationally very intensive. Providing proper algorithms to achieve reasonable efficiency is therefore crucial.

In this thesis, we present several techniques for dealing with either Lagrangian data or Lagrangian visualizations. First, we present an algorithm for the extraction of isosurfaces from Lagrangian simulation data. The second part focuses on Lagrangian Coherent Structures (LCS). We begin with analyzing different ways of defining LCS and analyze their properties. In particular, we propose a new definition which is numerically more stable. After presenting the LCS extraction algorithm in great detail, we analyze the flow around a revolving door combined with an air curtain. The LCS-based analysis was able to reveal a leak where heat can escape the interior. In the last part, we present our research in the area of visualization systems. Converting between the two flow representations is inherently a time-dependent process and time-dependent algorithms are surprisingly difficult to achieve in visualization systems. The presented system not only makes processing time-dependent data simple, it minimizes the amount of computation performed. Also, simulation components can be included very easily, eliminating the need for having two tools for creating and analyzing simulation data.
Computational Fluid Dynamics (CFD) simulations are a highly popular tool in science and engineering. They make it possible to study phenomena for which experiments are impossible, like the simulation of super-novae or galaxy formations. In engineering, they are frequently employed for design optimizations prior to construction, such as in airplane, water turbine or car designs. Even though real experiments can be applied in the classical engineering applications, computer simulations make it possible to investigate the behavior of the flow at any point in space and time, which is difficult to achieve in real experiments. The analysis of CFD simulations, however, is not an easy task either. The most apparent challenge is the size of typical simulations, easily reaching gigabytes to terabytes. Simple plots or statistical measures are useful to reduce the complexity and size to a minimum, but are of little help when trying to understand why the flow is behaving the way it does. In contrast, flow visualization makes use of the fact that the human eye is much better at interpreting images than numeric data. But dataset sizes already moved beyond the limits of human perceptual capabilities.

One possible approach to deal with the enormous data size of flow simulations is feature-based flow visualization. Examples of such features are shock waves, vortices or separatrices. The result is a sparse visualization with comparably little visual complexity. At the same time, it becomes possible to focus on the chosen phenomena; this allows showing semantic information about the flow.

Features are detected by analyzing the flow field. Therefore the question on how a flow field is represented is central to the design of feature detection algorithms. In physics, there are two fundamental ways of representing flow fields: the Eulerian and the Lagrangian specification. In the Eulerian case, the flow field can be imagined as an array of sensors, measuring the velocity over time at fixed positions. In the Lagrangian case, the sensor moves with the flow field, recording its position as the sensor travels with the flow. Mathematically, in the Eulerian specification, velocity is a function of time and space:

\[ \mathbf{v}(\mathbf{x}, t) \tag{1} \]

In the Lagrangian case, the sensor records its position, therefore the position is a function of time and the sensors initial position:

\[ X(x_0, t) \tag{2} \]
By using the equations of motion, it is possible to convert a flow field from one specification to the other:

$$\mathbf{v}(X(x_0, t), t) = \frac{\partial}{\partial t}X(x_0, t)$$

Depending on the simulation or visualization method requested, one representation becomes more natural than the other. Regular or unstructured grids are common examples used for Eulerian CFD simulations. On the Lagrangian side, the best known example is Smoothed Particle Hydrodynamics (SPH), where the flow is represented using particles. Both representations have their advantages and disadvantages: For dense flows, Eulerian representations work better because the static grid makes spatial computations, such as gradient estimations, more efficient. Lagrangian flow simulations on the other hand are able to represent free flows, such as waves or jets much more naturally than fixed grids. At the same time, dealing with boundaries such as walls is more difficult in the Lagrangian case and still an active research topic in the case of SPH. For visualization, the difference in the two specifications is of importance when dealing with simulations of either type. However, depending on the type of visualization required, one or the other representation may work better. For example, extracting an isosurface is an Eulerian visualization while a pathline is the most natural Lagrangian visualization. Another well-known example are topological separatrices for unsteady flows. There have been attempts to define coherent structures using an Eulerian approach, by segmenting the flow based on instantaneous scalar or vector fields. However, given a sufficiently unsteady flow, results tend to be unreliable, making a Lagrangian approach much more favorable.

In visualization, however, the representation is mandated by the underlying simulation. It can therefore become necessary to change the representation by using Equation 3. When looking at Equation 3, one notices that switching from one representation to the other requires integration and/or differentiation. Differentiation, and especially integration are computationally expensive processes. A careful choice of algorithms is therefore required to ensure good performance and therewith usability.

Eulerian visualization methods, such as isosurfaces, mostly work on instantaneous flow fields. For a visualization system, which is in charge of managing the simulation data, this is simple to support. The user selects a particular time value of interest, which the system can store in a global state. All operations are then performed on the data associated with the chosen time value. Even when working on Lagrangian data, velocity information is typically stored as an attribute of the particles. Therefore, the usage of Equation 3, which would result in a more challenging scenario due to the derivative in time, is
not necessary. Lagrangian methods on the other hand are time dependent by nature and as a result a single global state is not sufficient.

1.1 Contributions and layout of this thesis

In this thesis, we improve various aspects of dealing with the Lagrangian representation in the context of visualization. This thesis is grouped into three parts.

In Part i, we describe a method for extracting isosurfaces from SPH data, or in more generic terms, an Eulerian visualization based on Lagrangian simulation data. The main advantage of the presented method is that it works directly on the underlying SPH representation and does not require resampling of the fluid domain while retaining efficiency. In comparison with previous methods, the method provides precision guarantees while being several orders of magnitude faster.

In Part ii, we present our research in the area of Lagrangian Coherent Structures (LCS) and Finite-Time Lyapunov Exponent (FTLE). As their name implies, LCS are Lagrangian features and are regarded as a possible replacement for Vector Field Topology (VFT) separatrices for unsteady flows. We begin by comparing various alternatives for defining and extracting LCS. In addition, we propose a new improved definition, called the C-ridge, which is computationally more robust. We also provide a complete and efficient implementation of LCS computation based on the C-ridge definition, with a focus on Eulerian input data. We wrap up this part by applying these results on a simulation of a revolving door with an air curtain. Our LCS-based analysis revealed a heat leak which would have been very hard to find otherwise.

In Part iii, we present our visualization systems research. The primary goal was to create data-flow algorithms which enable seamless integration of time-dependent visualization methods. The algorithms were built with the visualization system Visdom in mind [2]. Visdom has the ability of handling multiple scenarios concurrently. As a result, the presented algorithms enable the development of modules for the handling of both time-dependent data with multiple scenarios, while minimizing the amount of computations performed.

The contributions in this thesis are spread out over several areas within scientific visualization. The fundamentals of each research area are therefore introduced at the beginning of each part.

1.2 Publications

This thesis is primarily based on the following publications:


During my time as a Ph.D. candidate, I also contributed to the following publications:


Smoothened Particle Hydrodynamics (SPH) is a purely Lagrangian fluid simulation technique. As such, the fluid is represented using Lagrangian particles without any information about their neighborhood. For this reason, creating visualizations of such data is algorithmically and computationally challenging. An easy way around these problems is to resample the data, which makes it possible to use many existing visualization techniques designed for grid layouts. Yet, the SPH representation also offers several unique advantages, making it worthwhile to develop visualization techniques working directly on SPH data. In the following chapters, we present our algorithm for the extraction of isosurfaces directly from SPH data. In the context of this thesis, this represents an Eulerian visualization method applied to Lagrangian data.
SPH is a Lagrangian simulation method, introduced by Gingold and Monaghan [26] and Lucy [48] for astrophysical problems. In its original form, SPH is purely Lagrangian, i.e., the sample points move with the flow. SPH nowadays is an umbrella term for a certain group of simulations using the same framework. Some types of SPH simulations are not purely Lagrangian. In other words, retrieving a pathline starting from a particle position does not always simply amount to following the particle. For Lagrangian visualizations which take advantage of the Lagrangian nature of SPH, differentiating classes of SPH methods becomes important. For Eulerian visualization, the difference is less of an issue. As a matter of fact, the velocity field is typically present as a particle attribute, and therefore Equation 3 does not have to be applied directly in order to extract an Eulerian flow field.

SPH represents fluids using particles. It is therefore ideally suited for simulation cases with open boundaries or free surfaces. Thanks to improvements in the treatment of boundaries, SPH has become also increasingly popular in industrial applications. In the following, we will focus on aspects important to Scientific Visualization as a complete introduction to SPH based simulations would be out of scope of this thesis.

SPH is completely meshless, i.e., there exists no static or dynamic connectivity structure. Therefore, computation of any quantity is done by averaging or smoothing over the set of neighboring particles. The interpolation rule for a quantity \( A \) from SPH data is

\[
A(x) = \sum_{j=1}^{N} A_j W(||x - x_j||, h_j) \Delta V_j
\]  

(4)

where \( A_j \) is the value of \( A \) associated with the \( j \)-th particle, \( x_j \) its position, \( h_j \) its smoothing length, and \( \Delta V_j \) is the volume of this particle. All these quantities are available as attributes of each particle, where the particle volume may be given as the quotient of the mass \( m_j \) and the density \( \rho_j \). \( W \) is a radially symmetric kernel which typically is a low-degree polynomial resembling a Gaussian. The kernels are often designed to fulfill certain physical properties such as conservation of momentum or mass. Formally the summation is done over all \( N \) particles, but practically, the kernels have a finite radius and therefore it is only done over the particles which have the point \( x \) within their kernel support. It is important not to confuse the smoothing length \( h \) with the size of a particle. Even though they are connected such that
a larger particle typically has a larger smoothing length, the computation of the particle size is independent of the smoothing length. The smoothing length is proportional to the volume in which it influences interpolation, and therefore also the simulation dynamics. Retrieving the volume of a fluid can be done by summing over the volumes of each contained particle.

Ideally, the sum of weights

$$
\sum_{j=1}^{N} W(\|x - x_j\|, h_j) \Delta V_j
$$

is equal to one, which means that constant functions are correctly interpolated at arbitrary points. In practice, Equation 5 is fulfilled only at particle positions. Between particles, it is only approximated – and interpolating between particle positions is very important for visualization purposes. Also, close to the boundary, the sum of the weights drops quickly because the number of the particles in the neighborhood drops. Therefore, a more precise reconstruction is obtained by the normalized interpolation, that is the quotient of Equation 4 and the interpolation of unity (Equation 5)

$$
A(x) = \frac{\sum_{j=1}^{N} A_j W(\|x - x_j\|, h_j) \Delta V_j}{\sum_{j=1}^{N} W(\|x - x_j\|, h_j) \Delta V_j}. 
$$

Taking the (spatial) gradient of Equation 4 leads to the simple expression for the gradient of A,

$$
\nabla A(x) = \sum_{j=1}^{N} A_j \nabla W(\|x - x_j\|, h_j) \Delta V_j.
$$

When working with normalized interpolation, the calculation of the gradient changes similarly. It can be obtained by applying the quotient rule of differentiation to the quotient of Equation 4 and Equation 5

$$
\nabla A(x) = \frac{\sum_{j=1}^{N} A_j \nabla W_j(x)}{\sum_{j=1}^{N} W_j(x)} - A(x)\frac{\sum_{j=1}^{N} \nabla W_j(x)}{\sum_{j=1}^{N} W_j(x)} \frac{\sum_{j=1}^{N} (A_j - A(x)) \nabla W_j(x)}{\sum_{j=1}^{N} W_j(x)}.
$$

From a visualization perspective, the above formulas are interesting for several reasons: We can apply the same kernels used for the
simulation, giving a desirable consistency between visualization and simulation. Also, because quantities are smoothed by the interpolation, the resulting fields exhibit very little noise making them easier to deal with. This advantage becomes even more significant when gradients are involved. With the presence of an analytic, and therefore exact formula for gradients, SPH gradient fields often are of high quality with little noise.

SPH however poses also a number of problems to visualization. First, the interpolation formula requires a lookup for finding the neighboring particles. In typical SPH simulations, the number of neighboring particles easily reaches 50–100 particles, making a single interpolation an expensive operation. Therefore, methods often try to minimize the amount of such interpolations, e.g., by caching, in order to get good performance. Second, boundary treatment is still an open problem: Currently, there exists no binary test for correctly deciding whether a point is within the domain or not. Such a test would have to correctly take into account shape and volume of every single particle. However, SPH does not specify the shape of a particle. Therefore, one typically relies on simple heuristics for this problem. In this thesis, we implement a frequently used (see e.g., [59]) test of Equation 5 against a threshold. However this is far from optimal.
Contouring is the process of extracting isocontours from a scalar field. Intuitively, an isocontour in 2D are height lines, obtained by walking along a surface at constant height. In 3D, isocontours become 2D surfaces, referred to as isosurfaces. Mathematically, an isocontour is defined by the implicit equation

$$\phi(x) = c.$$  

(9)

where $\phi$ is a d-dimensional continuous scalar field and $c$ a constant. Contouring is a highly popular and fundamental technique for the visualization of scientific data, often one of the first things applied when investigating new data. By using derived fields such as $\lambda_2$ [36], it can also become a building block for feature extraction.

There exists a huge body of literature on isosurface extraction with specializations for the different types of data discretization. Historically, isosurfaces were obtained by generating contours on a set of planar sections and stitching them together [21]. Direct isosurface extraction from volumetric data started with Lorensen and Cline’s seminal paper [47] and subsequent modifications for topologically consistent isosurfaces [53, 50], well known as the Marching Cubes algorithm.

Marching Cubes operates on hexahedral cells and produces polygonal meshes as its output. It starts by classifying the eight vertices of the cell as above or below the requested isovalue. Using this binary scheme results in a total of 256 different configurations. By constraining mesh vertices on edges between grid vertices, the topology of a single configuration is fixed. Therefore, it is possible to create a table covering all possible cases, making Marching Cubes a very fast method for contouring.

Following its initial publication, Marching Cubes became a very active research topic. Optimized methods based on octrees [90], interval trees [14, 13] or extrema graphs [34] avoid the processing of cells not intersected by the isosurface. Several efficient methods use the span-space [45, 71, 10] for fast and space-efficient retrieval of the set of all intersected cells. In view-dependent isosurface extraction methods [44, 43], visibility culling avoids computation on parts of the surface that are not visible. Isosurface extraction from time-dependent data requires temporal extensions of the acceleration data structures. Recent methods are based on persistent octrees [72, 84] or on a temporally extended span-space [88]. For a more thorough review of the topic, we refer the reader to a survey of Marching Cubes methods by Newman and Yi [52].
While the problem of point-based isosurface extraction from grid-based data has been studied extensively [46, 81, 51], there is relatively little work on point-based isosurface extraction from point-sampled data. Co and Joy presented an isosurface algorithm for point-sampled data [15] where a set of point splats is generated and rendered by a splatting technique [16]. Rosenberg and Birdwell [59] presented an isosurface method for SPH data. It is restricted to the special case of extracting the free surface, which is assumed to be where Equation 5 falls below a given threshold. In our method, we use the same criterion for cutting the isosurface, when it reaches the data boundary. An accurate method for finding free surface particles in SPH data has been proposed by Marrone et al. [49]. By using a level-set method, the free surface can be reconstructed. The level-set method, however, assumes a constant particle size and does not attempt to preserve the fluid volume.

The method by Rosenthal and Linsen [60] uses a similar approach, where the field is reconstructed only on a set of edges between neighbor samples. An angle criterion is used for refining the set of neighbors, preventing the need for a Delaunay triangulation. The isosurface is obtained by using point splatting where the normals of the point splats are obtained by least-squares fitting. As a result, an explicit isosurface geometry is not provided. In two later papers, a level-set technique is used for smoothing the isosurface [63] and the smoothing process is combined with the isosurface extraction into a Partial Differential Equation (PDE) [61]. A follow up [62] improves the performance of their approach by evaluating the level-set function only in a narrow band around the zero-level set. One issue with their approach is that they deal with point-sampled data, not with functionally represented data (kernel-based) data like SPH. By using linear interpolation, the reconstructed field values do not correspond to the true SPH field value. Also, the field value of a given point is dependent on the triangulation and not just on the data itself. With SPH data, the field can be reconstructed precisely at any point in space. It does not make a difference whether the query location is a particle position or not. Therefore, SPH is an implicit representation of the exact isosurface. The goal of an SPH isosurface algorithm must therefore be to generate samples of the exact isosurface. In that perspective, a smoothing process is not desirable for SPH data.
4.1 ALGORITHM DESCRIPTION

In this chapter, we will describe our method of extracting isosurfaces directly from SPH data. Our method is based on the Marching Cubes method, with a number of modifications. As described in Chapter 3, there are a number of optimization techniques avoiding the processing of empty grid cells. In our algorithm, we take this idea one step further such that cells which are not investigated do not consume any memory. This is important because SPH particles can be distributed highly irregularly in space, resulting in a grid of enormous dimensions. Therefore, the Marching Cubes grid, which will be called virtual grid for the remainder of this chapter, is represented implicitly using origin, dimension and voxel size.

The algorithm consists of an initialization phase and two main parts as can be seen in Figure 4.1. During initialization, an acceleration structure is created for efficient SPH interpolation (see Section 4.1.1). In the first step (see Section 4.1.3), seed cells in the virtual grid are selected. The number of seed cell selections mainly depends on the isosurface size and complexity and not on the size of the virtual grid. In the second step (see Section 4.1.4), the processing starts at the seed cells and creates the surface geometry.

4.1.1 Spatial Data Structure

For efficient SPH evaluation a spatial data structure is needed because of the required neighborhood lookup. In our implementation, we chose to use a regular grid where every particle is only stored in the cell where its center lies. The grid spacing is chosen to be $2 \cdot \max \{h_i \mid i = 1..N\}$. As a result, a neighborhood lookup has to consider 27 regular grid cells. The factor 2 results from the SPH kernels which usually have an extent of $2h$. A regular grid performs well as long as the particle size is nearly constant, which is true in current engineering applications. For data with large varying particle sizes however, an adaptive spatial data structure, such as a kd-tree, should be used. This step can be seen as a form of preprocessing and does not have to be repeated when other isosurface levels are requested on the same data.
4.1.2 Virtual Grid

The presented algorithm is based on a virtual grid consisting of uniform cubic cells. This grid, which is not to be confused with the regular grid used for particle search (as explained in Section 4.1.1), is never stored in memory entirely but only represented by means of extent, dimension and cube size. The edge length of a cube is set to $c \cdot \min\{h_i | i = 1..N\}$ where $c = 0.5$ per default. The spatial sampling rate given by the particles in our two example data sets was about 0.55$h$, 0.65$h$ (dam and tsunami) and $h$ (vortex rings). Reasonable values for $c$ are therefore between 0.5 and 1.0. With values higher than 1.0 the topology of the isosurface can start to become simplified. By choosing $c$, a trade-off is made between level-of-detail and computation time.

A major issue when dealing with SPH is that it is very expensive to interpolate a value. When using Marching Cubes it is normal that a value at a given position is requested multiple times – up to eight times per grid vertex. A lot of performance can therefore be gained...
by caching values for later reuse. One benefit of the virtual grid is that there is no direct cost associated with it in terms of memory consumption as the grid is only implicitly available. It is therefore not an option to use a full grid for the cache. Instead we resort to a slice-based approach, similar to the one used in [59]. As we will see, it is sufficient to keep three slices at once in memory as the cubes are calculated more or less in order (see also Section 4.1.4). Since the set of seed cells is not the complete set of intersected cells, it can happen that a grid point which is in a slice already deleted needs to be evaluated. Such cache misses of course affect performance of the method. At the cost of increased memory consumption, one can increase the number of slices which are kept in memory at once. However, in the three examples, and for all time steps used to create the videos, a cache of three slices was sufficient to get a near-optimal cache hit ratio.

4.1.3 Seed Cell Selection

The purpose of this step is to find cells of the virtual grid, with the goal to approximate the set of cells that are effectively intersected by the isosurface. Selecting seed cells that are not intersected by the isosurface is not a problem, but their number should be small in order to avoid unnecessary computation. False negatives, or missed cells that do intersect the isosurface, are also not a problem as long as there is at least one seed cell selected per connected component of the isosurface. For performance reasons it is desirable to have as many (positive) seed cells as possible to improve the cache-hit ratio. This is in contrast to other work [6] where the objective is a minimal set of seed cells.

The basic idea of the seed cell selection is to find pairs of neighboring particles which have data values below and above the isosurface level. For this we compare each particle with all particles within a radius of \( h \). Once such a particle pair is found, we have to make sure that a cell of the virtual grid which contains the isovalue is selected. Assuming continuous data, this has to happen at least once on the straight line connecting the two points because of the intermediate value theorem. Therefore, all cells intersected by the straight line between the two particles are added the list of seed cells. This is done using a Bresenham-style iteration.

A performance issue of the SPH representation is now that the evaluation of smoothed data is quite expensive because it requires finding a large set of neighbors, computing distances to them, and evaluating kernel functions. To make seed cell selection fast, we use the raw particle data, i.e., the data specified at the particle positions. This avoids evaluating kernel functions. SPH kernels satisfy the partition of unity property (Equation 5), which means that smoothed data do
not deviate significantly from the raw particle data. More importantly, by using raw data we do not miss any small connected component: small components are near local minima or maxima of the scalar field. In the case of a maximum, the smoothed scalar field cannot assume values larger than the largest raw data value that contributes to the weighted sum (Equation 4). This means that taking raw instead of smoothed data expands the data range in the vicinity of a local minimum or maximum, which can lead to unnecessary seed cells (false positives), but not to missed isosurface components.

The process of selecting seed cells is very well suited for parallelization, and has been implemented on the GPU using CUDA [55]. The resulting seed cell indices are transferred to the Central Processing Unit (CPU) and are stored in a set container (from the C++ standard template library), which is internally a tree structure. The set container by design cannot store the same key multiple times, so all the duplicate cells are eliminated automatically. The indices are sorted in the tree using a lexicographic ordering of the cube index. This facilitates caching (see Section 4.1.2) where both lexicographic ordering of the virtual grid indices and efficient insertion are required.

4.1.4 Surface Extraction

Given the set of seed cells, the actual extraction of the surface is performed. From a broad view, this process is similar to the Marching Cubes scheme with modifications described in the following subsections.

4.1.4.1 Marching Cubes Table Lookup

For each seed cell, a standard Marching Cubes step is done using a publicly available lookup table by Bourke [11]. This produces a set of consistently oriented triangles that correctly matches triangles generated in adjacent cells. The Marching Cubes method generates vertices by linearly interpolating data on the edges of the cell. A lookup in a second table is made to retrieve the subset of neighbor cells where the isosurface continues. All these cells are added to a queue of cells provided that they have not been visited before. This simple queue mechanism makes sure that only those cells have to be processed which are either seed cells or which effectively contribute to the isosurface. It was already used by Wyvill et al. [91] in their isosurface method predating Marching Cubes.

4.1.4.2 Correction Step

Since the SPH representation allows for an accurate evaluation at arbitrary points in the domain, we can apply correction steps to the vertices of the triangle mesh. These vertices lie on edges of a cube within the
virtual grid, the two end points of which have a pair of values above and below the isosurface level. According to the intermediate value theorem, there must be a point on the edge where the field exactly equals the level. The idea is now to correct the vertex found by linear interpolation toward this point. This is done by evaluating the directional derivative of the field with respect to the coordinate that varies along the edge. The directional derivative is one of the three components of the gradient (Equation 7) and it can be computed efficiently together with the field value (Equation 4). With this derivative, a number of Newton-Raphson steps are done, making sure that corrected points stay within the extent of the edge. A small number of iterations leads to points lying on the actual SPH isosurface with high precision.

4.1.4.3 Boundary Trimming

In contrast to grid based data, SPH data usually has no concise definition of the boundary. Also, even if a concise definition was available, it would not be aligned with the virtual grid used by our algorithm. Solid boundaries, or walls, are sometimes available as an extra set of “solid particles”, or they are just geometrically specified. Free boundaries have to be detected from the set of (fluid) particles, e.g., by the algorithm of Marrone et al. [49]. As mentioned in Chapter 2, we approximate the free surface by checking the sum of weights (Equation 5) against a threshold. In this work, we use this simple technique for detection of all boundaries. However, our framework also allows for an implementation of Marrone’s method, which basically amounts to computing a derived field on grid nodes near the boundary and taking its zero level set. Boundary detection is required for our method because we have to trim the isosurface mesh when it runs past the free surface. We not only have to make sure that all vertices of the mesh satisfy the isosurface equation, but also that they are within the fluid volume.

There are two thresholds used, the node threshold for the virtual cube vertices and the vertex threshold for the isosurface vertices. Cells of the virtual grid where the sum of weights falls below the node threshold for any of its vertices are discarded. Therefore, this threshold is set to a very low value, such as 0.1 to prevent clipping isosurface parts which still fulfill the vertex threshold. The vertex threshold is used for trimming triangles that have at one or two of its vertices a sum of weights below this threshold. We recommend a value of 0.5 to ensure a faithful interpolation. Trimming is done as is shown in Figure 4.2 and it can result in a quad which has to be split into two triangles. We will refer to the vertices created by the trimming process as boundary vertices as opposed to the interior vertices of the triangle mesh (cf. Figure 4.3).
18 polygonal isosurface extraction from SPH data

Figure 4.2: Different trimming cases. The relation symbols indicate the sum of weights relative to the vertex threshold. If the sum of weights is above or equal to the threshold at all vertices, the triangle is kept (a). Otherwise, if the sum of weights is below or equal at all vertices the triangle is discarded (b). It is trimmed in the three remaining cases (c-e).

Figure 4.3: Comparison of original (top) and trimmed (bottom) wireframe.

4.1.4.4 SPH Surface Normal Evaluation

Normal vectors are required for the rendering of a surface with a local lighting model. Accurate surface normals are therefore essential for a correct perception of the isosurface. A standard method is to estimate the normal at a surface vertex by a weighted average of the normals of the adjacent triangles. However, due to the irregularly shaped triangles in Marching Cubes meshes, this simple approach produces poor normals which show as artifacts in the rendered surface. Better isosurface normals are obtained by using the underlying scalar field and estimating its gradient at the vertices of the isosurface. In the special case of SPH data, high-quality gradients can be obtained directly at arbitrary points by (Equation 7). Since the calculation of the SPH gradient is more expensive than a simple interpolation, these
high-quality normals are not computed in the preview quality mode where mesh vertices are not corrected.

4.2 RESULTS

The algorithm has been tested on three data sets, the tsunami data set from [58], the dam breaking simulation [40], and the vortex rings [78] data set.

There are two parameters in the algorithm which have a large impact on the performance. First, there is the ability to adapt the virtual grid size. It can be increased from the default 0.5h for preview purposes to get better performance. Also, depending on the smoothness of the data, a low value here leads to an unnecessary high triangle count. The second parameter is a scaling factor for the smoothing length $h$ of the SPH data. It has been shown [35] that shrinking the kernel size can give a huge speedup with only a small loss of quality. This will reduce the quality of the SPH interpolation but also make it faster as the number of neighbors is reduced. For preview purposes this loss of quality is very often acceptable.

All timings were performed on a Xeon E5430 with 64 gigabytes of RAM running Linux. The GPU used is a Nvidia GTX 295. Even though the GTX 295 is equipped with two GPU, only one of them was used. Also, the CPU portion of the code only uses a single core. As described in Section 4.1.3, the seed selection process is implemented on the GPU. For this, the SPH data has to be transferred onto the GPU and the resulting seed cells have to be fetched back which is known to be a costly process. Therefore, the timings of the seed selection process include memory transfers from and to the GPU.

4.2.1 Tsunami Data Set

The first test data set is an SPH simulation of the creation of a tsunami [58]. In the simulation, a wedge is sliding downward which is idealizing a landslide, as seen in Figure 4.5. The simulation contains 249 time steps and 1.2 million particles and uses the quadratic kernel in (Equation 10) with a support radius of 2h. The smoothing length is the constant $h = 0.056$:

$$W(r, h) = \begin{cases} \frac{15}{16\pi h^3} \left( \frac{1}{4} q^2 - q + 1 \right) & 0 \leq q \leq 2 \\ 0 & q \geq 2 \end{cases}$$

(10)

where $q = \|r\|/h$. 
Figure 4.4: Pressure isosurface (time step=194, p=3000) in the tsunami data with a virtual grid size of $0.5h = 0.028$ without (a) or with (b) the effect of scaling down the smoothing length by a factor of 0.5. The surface is colored by the sum of the kernels, where green means a value of 1 and red a value of 0.5 or less. Even with the reduced kernel size, the algorithm manages to reproduce the large surface without visible defects and to capture high-frequency details, such as the “splashes” in the front part of the isosurface.

Figure 4.5: Vorticity magnitude isosurface in the tsunami data (time step=150, vorticity=5).

4.2.2 Dam Break Data Set

Our second test data set is an SPH simulation of the SPHERIC dam breaking case [40]. It has 87 time steps, each with roughly 670,000 fluid particles using the popular cubic spline kernel:

$$W(r, h) = \begin{cases} 
1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & 0 \leq q \leq 1 \\
\frac{1}{4}(2 - q)^3 & 1 \leq q \leq 2 \\
0 & q \geq 2
\end{cases}$$

(11)

where $q$ is defined as in Equation 10.

Solid boundaries are modeled with solid particles, while the air contains no particles. In contrast to the tsunami data set where the smoothing length is fixed for every particle, the dam break data has a
variable smoothing length. When varying the kernel size as described in Section 4.2.4, the smoothing radius of every particle is multiplied by a constant factor.

4.2.3 Vortex Rings Data Set

The third data set is a simulation of the collision of two vortex rings at circulation-based Reynolds number equal to 1600 using a vortex method [78]. The initial conditions are set to reproduce the experimental results by [68]. It contains about 10,000,000 particles and uses a cubic kernel as defined in Equation 12 with a radius of $h = 0.0005$. There are no free surfaces in this data.

$$w(q_i, h) = \begin{cases} 
\frac{1}{h} - \frac{5}{2} q_i^2 + \frac{3}{2} q_i^3 & 0 \leq q_i \leq 1 \\
\frac{1}{2} (2 - q_i)^2 (1 - q_i) & 1 \leq q_i \leq 2 \\
0 & q_i \geq 2 
\end{cases} \tag{12}$$

and

$$W(r, h) = w\left(\frac{\nabla x}{h}, h\right) \cdot w\left(\frac{\nabla u}{h}, h\right) \cdot w\left(\frac{\nabla z}{h}, h\right) \tag{13}$$

In the work of Rosenthal et al. [62] they write that it takes about 6 minutes to extract a surface of 47k vertices from a data set of 8 million particles. Comparing this to our performance in Table 4.1, we find that our method is about 80 times faster, extracting about 150k vertices from a data set of 10 million particles.

4.2.4 SPH Kernel Size

In SPH simulations the number of particles overlapping a position within the data is usually very large – on average about 70 for the
Figure 4.7: Transparent rendering of the vorticity isosurface with $\omega = 10$ (a) and $\omega = 0.1$ (b).

<table>
<thead>
<tr>
<th>h scaling factor</th>
<th>1</th>
<th>0.75</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid size</td>
<td>0.5h</td>
<td>h</td>
<td>0.5h</td>
</tr>
<tr>
<td>Grid construction</td>
<td>1.13</td>
<td>1.13</td>
<td>1.4</td>
</tr>
<tr>
<td>Seed cell selection</td>
<td>0.72</td>
<td>0.72</td>
<td>0.55</td>
</tr>
<tr>
<td>Surface extraction</td>
<td>3.48</td>
<td>0.73</td>
<td>2.90</td>
</tr>
<tr>
<td>Total w/o corr.</td>
<td>4.20</td>
<td>1.45</td>
<td>3.45</td>
</tr>
</tbody>
</table>

Table 4.1: Computing time (seconds) for a vorticity isosurface ($\omega = 20$) in the vortex rings data, $h = 0.0005$.

dam break data and over 200 for the tsunami data. This overlap can be reduced by shrinking the kernel size by a constant factor. Doing so can result in an enormous speedup without sacrificing too much accuracy, which for a preview is certainly sufficient. Next to the original kernel size, we selected the scaling factors 0.75 and 0.5 for testing.

In case of the tsunami data set, setting the scaling factor to 0.75 reduces the extraction time by almost 40%, as shown in Table 4.2. At a scaling factor of 0.5 the performance is doubled, and the visual quality did not suffer visibly as seen in Figure 4.4.

With the dam break data the performance impact of kernel size reduction is similar as seen in Table 4.3. In case of a noisy surface as it is the case in the pressure isosurface of the dam break data, a factor of 0.5 can introduce very small isosurface components close to the boundary, where the SPH interpolation quality suffers even more, see Figure 4.8. Otherwise, the surface was extracted with almost no visual errors.

4.2.5 Virtual Grid Size

Depending on the resolution of the simulation and the smoothness of the data, a grid as fine as 0.5h is not needed. In Table 4.4 the grid statistics with two different grid sizes for the tsunami data set is shown. Even though the number of grid cells is shrunk by a factor of
Figure 4.8: Effect of excessive kernel scaling in noisy data sets. In the two close-ups the isosurface is replaced by one generated with scaled (0.5) kernel, revealing artifacts which are not present in the original version.

eight, all other numbers are more or less decreased by a factor of four which is to be expected because of the 2D nature of the surface. The decrease in the number of seed cells is not as dramatic since seed cell selection works better with larger grid cells.

The effectiveness of the candidate selection can be observed in Table 4.4 and Table 4.5. The technique selected about 3% of all possible virtual cells, approximately half of them were used for the isosurface. The ratio could be improved a lot by using linear interpolation instead of the Bresenham algorithm in the seed selection. This way however, we cannot guarantee the evaluation of all positive seed cells.

In terms of runtime, the time needed for the surface extraction process is also cut roughly by a factor of four. When shrinking kernel size, the effect is less dramatic because the percentage of time used for data structure management such as candidate lists, cache handling etc. starts to increase.

### 4.2.6 Effectiveness of the Correction Step

In Table 4.6 and Table 4.7 the distribution of the sizes of correction steps is analyzed. It shows that correction steps are mostly in the order of a few percent of the edge length, but can reach up to a third of the edge length. Enabling the correction step roughly doubles the total time for surface extraction as shown in Table 4.2 and Table 4.3. However, if either a coarser grid is used or the smoothing length is scaled down, this overhead diminishes rapidly. The gain in visual quality by the correction step outweighs the loss by any of these two simplifications (Figure 4.4).
Table 4.2: Computing time (seconds) for a pressure isosurface (time step 149, \( p = 2000 \)) in the tsunami data. The grid construction times raise because of the finer-resolution of the grid when the kernel size becomes smaller. Grid construction time is omitted in the totals as its a preprocessing step.

<table>
<thead>
<tr>
<th>h scaling factor</th>
<th>1</th>
<th>0.75</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid size</td>
<td>0.028</td>
<td>0.056</td>
<td>0.028</td>
</tr>
<tr>
<td>Grid construction</td>
<td>0.08</td>
<td>0.08</td>
<td>0.1</td>
</tr>
<tr>
<td>Seed cell selection</td>
<td>0.7</td>
<td>0.7</td>
<td>0.36</td>
</tr>
<tr>
<td>Surface extraction</td>
<td>2.65</td>
<td>0.5</td>
<td>1.72</td>
</tr>
<tr>
<td>S.e. with correction</td>
<td>3.45</td>
<td>0.7</td>
<td>3.02</td>
</tr>
<tr>
<td>Total w/o corr.</td>
<td>3.35</td>
<td>1.20</td>
<td>2.08</td>
</tr>
<tr>
<td>Total with corr.</td>
<td>4.15</td>
<td>1.40</td>
<td>3.38</td>
</tr>
</tbody>
</table>

Table 4.3: Computing time (seconds) for a \( \lambda_2 \) isosurface (time step 30, \( \lambda_2 = -150 \)) in the dam break data.

<table>
<thead>
<tr>
<th>h scaling factor</th>
<th>1</th>
<th>0.75</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid size</td>
<td>0.006</td>
<td>0.012</td>
<td>0.006</td>
</tr>
<tr>
<td>Grid construction</td>
<td>0.07</td>
<td>0.07</td>
<td>0.1</td>
</tr>
<tr>
<td>Seed cell selection</td>
<td>0.08</td>
<td>0.08</td>
<td>0.05</td>
</tr>
<tr>
<td>Surface extraction</td>
<td>0.73</td>
<td>0.17</td>
<td>0.54</td>
</tr>
<tr>
<td>S.e. with correction</td>
<td>2.13</td>
<td>0.48</td>
<td>1.63</td>
</tr>
<tr>
<td>Total w/o corr.</td>
<td>0.81</td>
<td>0.25</td>
<td>0.59</td>
</tr>
<tr>
<td>Total with corr.</td>
<td>2.21</td>
<td>0.56</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Table 4.4: Statistics of virtual grid cells and cells actually used for isosurfaces of Table 4.2 when no particle shrinking is used.

<table>
<thead>
<tr>
<th>Grid size</th>
<th>0.028</th>
<th>0.056</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual cells</td>
<td>2,401,980</td>
<td>303,408</td>
</tr>
<tr>
<td>Seed cells</td>
<td>73,602</td>
<td>14,917</td>
</tr>
<tr>
<td>Non-intersected cells</td>
<td>47,007</td>
<td>8,012</td>
</tr>
<tr>
<td>Total cells used</td>
<td>29,388</td>
<td>7,183</td>
</tr>
<tr>
<td>Isosurface triangles</td>
<td>56,543</td>
<td>14,535</td>
</tr>
</tbody>
</table>

Table 4.4: Statistics of virtual grid cells and cells actually used for isosurfaces of Table 4.2 when no particle shrinking is used.
## 4.2 Results

### Table 4.5: Statistics of virtual grid cells and cells actually used for isosurfaces of Table 4.3 when no particle shrinking is used.

<table>
<thead>
<tr>
<th>Grid size</th>
<th>0.006</th>
<th>0.012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual cells</td>
<td>6,802,912</td>
<td>855,456</td>
</tr>
<tr>
<td>Seed cells</td>
<td>98,811</td>
<td>41,327</td>
</tr>
<tr>
<td>Non-intersected cells</td>
<td>38,286</td>
<td>11,151</td>
</tr>
<tr>
<td>Total cells used</td>
<td>156,810</td>
<td>36,319</td>
</tr>
<tr>
<td>Isosurface triangles</td>
<td>307,278</td>
<td>73,739</td>
</tr>
</tbody>
</table>

### Table 4.6: Statistics of the correction step for the tsunami isosurfaces of Table 4.2.

<table>
<thead>
<tr>
<th>Grid size</th>
<th>0.028</th>
<th>0.056</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean vector norm</td>
<td>0.000218</td>
<td>0.000807</td>
</tr>
<tr>
<td>Max vector norm</td>
<td>0.009890</td>
<td>0.018360</td>
</tr>
<tr>
<td>St.dev. vector norm</td>
<td>0.000402</td>
<td>0.001253</td>
</tr>
</tbody>
</table>

### Table 4.7: Statistics of the correction step for the “dam break” isosurfaces of Table 4.3.

<table>
<thead>
<tr>
<th>Grid size</th>
<th>0.006</th>
<th>0.012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean vector norm</td>
<td>0.000130</td>
<td>0.000453</td>
</tr>
<tr>
<td>Max vector norm</td>
<td>0.001997</td>
<td>0.005511</td>
</tr>
<tr>
<td>St.dev. vector norm</td>
<td>0.000182</td>
<td>0.000550</td>
</tr>
</tbody>
</table>
In this part we presented an algorithm for isosurface extraction, a fundamental visualization technique, from SPH data. The correction step used during the mesh extraction makes sure that vertices correspond to actual isopoints. Thanks to the usage of the virtual grid and the caching mechanism, the method is not only precise but also efficient. More specifically, compared to the works of Rosenthal et al. our method runs more than 80 times faster than their method on similarly sized data. For preview purposes, the performance can be improved even further by reducing the virtual grid size or by shrinking the kernel size. The method itself is also not limited to SPH data, but can be used for other functionally represented data, provided that the computation of gradients based on kernel gradients yields sufficient accurate results.

The algorithm however also has drawbacks. Due to the reliance on the Marching Cubes algorithm, we also inherit its limitations. In particular, the quality of the produced output mesh is low. Also, Marching Cubes assumes linearity within a single cell. An SPH scalar field, however, is not linear. Even though mesh vertices themselves are guaranteed to be on the isolevel, we are currently unable to guarantee topological correctness. This can become a problem when the scalar field exhibits large variance, for example in turbulent flows. As a result, choosing a proper virtual grid cell size is crucial. Also, free surface detection in SPH is still an open problem. We rely on a heuristic to detect it, but a more rigorous approach, for example based on the method of Marrone et al. [49] would be desirable.
Lagrangian Coherent Structures (LCS) are Lagrangian flow features separating flow areas of coherent behavior. They fall into the category of topological flow visualization. Traditionally, topological flow visualization was performed using Vector Field Topology (VFT). However, VFT fails to work properly with sufficiently unsteady flows. LCS are seen as a possible replacement for VFT separatrices. To locate them, the flow stretching and compaction behavior is measured over a fixed amount of time. Roughly, LCS are then defined as the areas where the stretching/compaction is maximal. A mathematically precise definition of LCS turned out to be challenging and is still an open problem. Current attempts to define them mathematically turned out to be overconstrained.

After introducing VFT and LCS in detail, we will compare various LCS definitions by evaluating the cross-flux. Also, we introduce a new definition, the C-ridge, which is computationally more stable. Then, we present the algorithm for extracting LCS. Even though the definitions for LCS look rather short on paper, there are several issues to account for during extraction. We conclude this part by analyzing the flow around a revolving door with an air curtain, mainly using LCS based on the C-ridge extracted using the presented algorithm.
6.1 CLASSICAL VECTOR FIELD TOPOLOGY

**VFT** is a well-known theory for the analysis of vector fields making use of concepts from the theory of continuous dynamical systems. In contrast to Lagrangian approaches which are inherently time-dependent, **VFT** only considers steady vector fields, that is, vector fields that do not change over time. **VFT** is useful because it achieves a very compact representation of a vector field, while retaining important features. As a result, it is ideally suited to get an overview over very large vector fields in order to locate regions of interest.

**VFT** was introduced into the visualization community by Helman and Hesselink [31]. Their original publication considered 2D-flows only. In a follow-up paper, it was extended to 3D [32]. In the following years, **VFT** became a very active research topic. An excellent tutorial was written by Asimov [5]. Even today, **VFT**-related research is still very active. The state-of-the-art report by Laramee et al. [41] gives a good overview over more recent developments.

It is a well-known fact that vector fields are equivalent to Ordinary Differential Equations (ODEs) [5]. The general idea of **VFT** is to consider the solutions of the initial value problem,

\[
\frac{dx}{dt} = v(x), \quad x(t_0) = x_0
\]  

or in other words, the streamlines of the vector field. It is important to note that **VFT** assumes the streamlines to be unique. This is the case, if the vector field is Lipschitz continuous [27]. In the practical case of discretely-sampled simulation or sensor data, this is fortunately the case.

Normally, a streamline does not return to a point which it has already visited. Such streamlines are called regular. However, there are two different ways a streamline can return to an already visited point. If it visits a point after a fixed amount of time, the streamline is called a closed or periodic orbit. If the vector field is zero at the initial condition, the streamline becomes a point. Such a streamline is called a stationary point, or, if it is an isolated point, a critical point.

Critical points and closed orbits can then be classified further. In case of a closed orbit, nearby streamlines are related to the closed orbit. Therefore, it is classified based on an eigen-analysis of the Poincaré map. A stationary point is classified based on the eigenvalues of its Jacobian.
The topological skeleton is defined as the collection of all critical points and closed orbits and their lower-dimensional invariant manifolds. Such manifolds are defined by the set of points whose streamlines converge to stationary points/periodic orbits in either positive (stable invariant manifold) or negative time (unstable invariant manifold). Note that only saddle points and saddle periodic orbits have lower-dimensional manifolds. In case of source and sinks, the invariant manifolds are not lower-dimensional and in case of centers it degenerates to the point itself. The invariant manifolds are also referred to as separatrices as they separate the flow which is diverted to either side of the saddle.

The main problem with VFT is that it is designed for steady vector fields. However, most real world scenarios are time-dependent. A very famous counter-example to VFT is the double gyre, published by Shadden [70], where the VFT saddle significantly deviates from the true separation point. Another example was published by Wiebel [89] where the stationary point, a sink, significantly lags behind the observed time-dependent attractor.

### 6.2 LCS and FTLE

There have been various attempts to find lines or surfaces that characterize the structures of an unsteady flow field in a similar way as topological skeletons characterize a steady flow. The most influential approach is Haller’s definition [29] of LCS, defined as the ridges in the scalar field of Finite-Time Lyapunov Exponent (FTLE). His initial motivation was to describe the structures formed by passive tracers in three-dimensional fluid flows. It was shown that these structures behave in a similar fashion in unsteady flows than as separatrices do for the steady case.

The foundation of the FTLE scalar field is the so called flow map. The flow map (in mathematics sometimes simply called the flow) of a vector field \( \mathbf{u}(x,t) \) is the map from the point where a massless particle is seeded at time \( t_0 \) to the point where it is located at time \( t \). The flow map is denoted by \( \Phi_{t_0}^t(x) \). Formally, it is described by the following initial value problem

\[
\frac{\partial}{\partial t} \Phi_{t_0}^t(x) = \mathbf{u}(\Phi_{t_0}^t(x), t), \quad \Phi_{t_0}^t(x) = x. \tag{15}
\]

Computing the gradient of the flow map \( \mathbf{F} = \nabla \Phi_{t_0}^t(x) \) leads to the (right) Cauchy-Green deformation tensor \( \mathbf{C} = \mathbf{F}^\top \mathbf{F} \). Intuitively, this gradient describes how infinitesimally close particle traces relate to each other. By multiplying the transposed flow map gradient with itself, rotation is cancelled out. This can be seen when looking at the polar decomposition [74] of the flow map gradient \( \mathbf{F} = \mathbf{R} \mathbf{U} \), where \( \mathbf{R} \) describes the rotation and \( \mathbf{U} \) stretching. Therefore,

\[
\mathbf{C} = \mathbf{F}^\top \mathbf{F} = (\mathbf{R} \mathbf{U})^\top (\mathbf{R} \mathbf{U}) = \mathbf{U}^\top \mathbf{U} \tag{16}
\]
With this, the finite-time Lyapunov exponent (FTLE) can be defined as

\[
\sigma(x, t_0, t) = \frac{1}{|t - t_0|} \ln \left( \frac{\sqrt{\lambda_{\text{max}}(C)}}{\lambda_{\text{max}}(C)} \right)
\]  

(17)

where \(\lambda_{\text{max}}(C)\) denotes the largest eigenvalue of \(C\).

The FTLE therefore describes the rate of separation of a pair of particles, where the maximum is taken over all spatial orientations of the pair. The FTLE can also be computed for a flow map going backward in time. In this case it describes attraction rather than repulsion.

Defining LCS from the FTLE field is an interesting topic. In the initial publication, Haller presents two approaches: an analytic and a geometric approach. The analytic approach is based on the extremal properties an LCS has to fulfill. However, it is unsuitable for the computational detection of LCS as every possible material line and surface would have to be investigated. Therefore, a second approach is presented which is based on the flow map gradient. This geometric definition is picked up by Shadden et al. who defines them as the ridges of the FTLE field \([70]\). In a later paper, Haller defines LCS directly on the basis of the Cauchy-Green tensor \([30]\). To complicate matters further, it was noted by Norgard as well as Peikert (see Section 7.6, as well as \([54, 56]\)) that the ridge definition used by Shadden as well as the Weak LCS definition from Haller \([30, \text{Theorem 7}]\), are over-constrained and are only usable when certain conditions are relaxed.
Haller introduced LCS as the ridges of the FTLE field. There have been numerical experiments [70], indicating that even though there is flux across LCS, it is negligible. However, this coincidence cannot be formulated as a mathematical statement, because the FTLE contains a parameter, the integration time, and also because there are multiple definitions for a ridge. Such a statement would for example hold in the limit of infinite integration time if a steady flow was assumed. Then, the Lyapunov exponent would be constant per trajectory, and the ridges, under any reasonable definition, would be material lines. In scientific visualization, we have to deal with finite time domains, and here it turns out that FTLE ridges can deviate significantly from material lines or surfaces.

Ideally, an LCS has both the properties of being a material line (or surface) and of being maximally attracting or repelling. In practice, a compromise has to be made and in the case of Haller’s two LCS definitions this means that only one property is met exactly in each definition. This observation can now serve as a basis for measuring the quality of various proposed LCS definitions on given test data. An obvious error metric for comparing different types of FTLE ridges, FTLE watersheds and related features is the flux through these lines or surfaces. The flux per unit length can be computed as the normal component of the velocity vector minus the velocity with which the feature moves in the normal direction. The latter can be estimated by tracking the feature over a small time interval.

Besides comparing different types of FTLE ridges with respect to this error metric, a second goal is to explore other FTLE-related concepts. The definition of FTLE contains a normalization and a natural logarithm. These two operations are inherited from the (infinite-time) Lyapunov exponent, where they are needed to make it independent of the starting time of a trajectory and, under the conditions of the Oseledec theorem, also independent of the trajectory. In the finite-time case these independences do not hold. Therefore, neither the logarithm nor the normalization (which both are monotone functions) are needed. This leads us to the study of the eigenvalues and eigenvectors of this tensor as a basis for an alternative definition of LCS lines and surfaces.
7.1 Watersheds, Height Ridges, and Section-Based Ridges

Even though ridges in a 2D height field are easily recognized by the human eye, there is not a unique ridge definition. Generally, a ridge is a lower-dimensional manifold in a scalar field that generalizes the notion of a local maximum.

A natural definition of a ridge is the **watershed**. For a 2D scalar field the watershed is obtained by integrating the gradient of the field, starting at a saddle point. Intuitively, this means to walk uphill from a pass to a peak, always taking the direction of the steepest slope. In topological terms, watersheds are part of the topological skeleton of the gradient field. This way, the extension to 3D scalar fields is straightforward: They are the unstable 2D manifolds of saddle points. Sahner et al. [66] used such 2D watersheds for their concept of strain skeletons in 3D velocity fields.

**Height ridges** are a generalization of local maxima. A (maximum convexity) height ridge of co-dimension one is given by the points which have a zero first derivative and a negative second derivative if derivatives are taken in the direction of the minor eigenvector of the Hessian [20]. Here, the minor eigenvector refers to the eigenvector associated with the smallest signed eigenvalue. Often this set of points is further reduced by additional constraints in order to remove false positives [42, 57].

**Section-based ridges** are obtained by finding local maxima in a set of parallel \(d - 1\)-dimensional sections and connecting them [37]. Obviously, the orientation chosen for the sections influences the result, and reasonably good results can be expected only if the ridge is roughly orthogonal to the sections. Section-based ridges have the advantage that they do not require second derivatives.

In 3D velocity fields, it can be observed that the **FTLE** field often has surface-like maxima, defining structures which are approximately orthogonal to the major eigenvector. Therefore, a height ridge of co-dimension one can be used to represent such structures [64].

7.2 Tensor Field Lines

**LCS** are obtained by applying any of the above ridge definitions to the **FTLE** field. Since the **FTLE** field only depends on the eigenvalues of \(C\), an alternative is to include also the eigenvectors of \(C\). The background of this is: The Cauchy-Green tensor \(C\) is a positive symmetric tensor, and \(U = C^{1/2}\) is the right stretch tensor in the polar decomposition \(F = RU\), while \(R\) is a rotation matrix. The eigenvectors \(N_i\) of \(U\) (being also the eigenvectors of \(C\)) are orthogonal and contain the directions of maximal and minimal stretch. The eigenvectors \(n_i\) of the left stretch tensor \(V\) (with \(F = VR\)) are the vectors \(N_i\) after rotation, \(n_i = RN_i\) (see Figure 7.1).
A tensor field line is obtained by integrating along the minor eigenvector of $C$. The seed point for this integration can be a local FTLE maximum or another point which is assumed to lie on an LCS. Integration is stopped when a degenerate point is approached where the eigenvector direction becomes undefined. The resulting structures have the property that they are aligned with the direction of maximal stretching. Tensor field lines are included in this study as an alternative to ridges.

### 7.3 C-ridges

Section-based ridges are biased by the sectioning direction. However, this can be largely reduced by using sections that are taken orthogonal to the predicted ridge direction. In our case of ridges of an FTLE field $\sigma(x)$, such an orthogonal direction exists, namely the major eigenvector $N_1$ of $C$, as was explained in Section 7.2. The ridge obtained this way can be expressed by

$$N_1 \cdot \nabla \sigma = 0 \quad \text{and} \quad (H_\sigma \cdot N_1) \cdot N_1 < 0 \quad (18)$$

where $H_\sigma$ denotes the Hessian of $\sigma$. It differs from the height ridge only in the use of the eigenvector $N_1$ which for a height ridge must be replaced by the minor eigenvector of $H_\sigma$. This new type of ridge, restricted to points where $\sigma > 0$, fulfills two of the conditions of Haller’s recent concept of weak LCS (cf. [30], Definition 7), while the other two (material surface, the surface itself is orthogonal to $N_1$) can hold in general only in the limit $t \to \infty$. For brevity of notation, we will use the name C-ridges for these ridges.

The C-ridge can be extended to a surface in 3D by taking a local FTLE maximum as the seed point and a tangential plane orthogonal...
to the major eigenvector. In Figure 7.2 we illustrate the integration approach for the C-ridge. At the seed point the ridge normal vector is given by the major eigenvector $N_1$ of $C$. We choose a vector $v_{00}$ normal to $N_1$ and rotate this vector around $N_1$ in discrete angular steps, giving vectors $v_{0i}$. We denote the $j$th integration step in the $i$th direction as $v_{ij}$ and the resulting point as $q_{ij}$. At $q_{ij}$ a correction step is done in the direction of the local eigenvector $N_1$, maximizing the FTLE value $\sigma$. The result is denoted by $p_{ij}$. The next step $v_{ij+1}$ is then computed as the projection of the previous stepping direction $v_{ij}$ onto the normal plane of $N_1$ at the point $p_{ij}$. Integration is stopped when $\sigma$ drops below a given threshold.

It is worth pointing out that both height ridges and C-ridges can be detected locally, i.e., they would not require a seed point. However, local maxima of FTLE always lie on a ridge, and by taking (a subset of) the maxima as the seed points, a subset of “important” ridges is obtained. This approach is also computationally more efficient, even though it means that duplicates must be checked for. Furthermore, it allows ridges to be tracked over time by tracking the local maximum, which is easily done by following the steepest ascent.

As a side note, the same procedure without the correction step would yield the 3D analogue to the tensor field line, which has been used for visualization [79] too. However, it is well known [69] that a surface exactly orthogonal to the major eigenvector does not exist in general.

### 7.4 Error Metric for the Material Line/Surface Property

Material lines and surfaces have the property that there is zero flux crossing the line or surface. A straightforward error metric is therefore based on the flux. Computing the flux requires relative velocities because the line or surface is moving. Formally, for the case of a line $\gamma$ in 2D, this is

$$\int_{\gamma} \left( u(x(s,t),t) - c(s,t) \right) \cdot n(s,t) \, ds$$

where $x(s,t)$ is the position of the curve point with parameter value $s$ at time $t$, $u(\cdot, t)$ the velocity field, $c(s,t)$ is the point’s velocity as given by its tracking, and $n(s,t)$ is the curve normal.

For a local measure, the flux is taken per unit length (or unit area). Its computation requires computing LCS for a sequence of time steps. LCS curves (or surfaces) are tracked over time and the speed orthogonal to the LCS is derived from its motion. This speed is compared to the local velocity field, again projected onto the orthogonal space of the LCS. The difference is the local flux, i.e., the flux per unit length or area. Since the local flux is commensurate with a velocity, it can be
normalized by dividing it by the local velocity magnitude. Normalized local fluxes were used by Shadden et al. [70]. This error metric can be used for line-type structures in 2D domains and for line-type and surface-type structures in 3D domains.

The tracking of an LCS over time is done by tracking its seed point. For the majority of the above feature definitions the seed point is a local FTLE maximum, which is simply tracked by following the gradient (steepest ascent method). The remaining feature types, watersheds and tensor field separatrices, are seeded at saddle points of the FTLE field and degenerate points of the tensor field $C$, respectively. These other seed points could be tracked by a critical point tracking method [76, 25]. For our study, we did a simple tracking by proximity.

However there is no one-to-one mapping between these seeds and the FTLE maxima where ridges are seeded. At the initial time step, the best matching watershed and tensor field separatrix have to be found manually. After each tracking step it is verified that the watershed still extends to the FTLE maximum and the separatrix does this sufficiently close. If not, this means that a “bifurcation” event has happened. Then the tracking is stopped, and the procedure is re-initialized.

7.5 Results

In this section we evaluate the differently defined LCS on series of time steps of two 2D vector fields. We also use a 3D data set from [64] for a qualitative comparison of height ridges and the proposed C-ridges.

7.5.1 “Double Gyre” Example

The double gyre is an analytic unsteady 2D vector field which was used by Shadden et al. [70] to demonstrate that saddles of vector field topology can deviate from the point of actual saddle behavior. The field is defined by

\[
\begin{align*}
    u(x, y, t) &= -\pi A \sin(\pi f(x, t)) \cos(\pi y) \\
    v(x, y, t) &= \pi A \cos(\pi f(x, t)) \sin(\pi y) \frac{df(x, t)}{dx}
\end{align*}
\]  

(20)

where

\[
f(x, t) = \varepsilon \sin(\omega t)x^2 + \left(1 - 2\varepsilon \sin(\omega t)\right)x
\]  

(21)

and $A = 0.1$, $\omega = \pi/10$, $\varepsilon = 0.25$. First, we use long-time FTLE to reproduce the findings from Shadden et al. [70]. Then, different ridge types are compared using short-time FTLE. This makes sense because the difference in flux is expected to be more pronounced when short integration times are used.
Results for Long-Time FTLE

As a first test, we measured the flux through a height ridge of an FTLE of the “double gyre” computed with integration time $T = 30$, and compared this with the findings of Shadden et al. [70]. Their Fig. 8 can be roughly confirmed with our method of computing height ridges of FTLE. Also the flux through this ridge is consistent with their values. However, our height ridge computed for the same parameter values ($\lambda = 0.1, \omega = 2\pi/10, \epsilon = 0.1$) passes the point $x = 1.019, y = ...
That means that the ridges disagree by about 0.0003. We computed the FTLE field using the C version of Netlib’s RKF45 with relative error setting of $10^{-9}$ (and consistently also with $10^{-8}$), and with grid resolutions of $2 \cdot 10^{-7}$ and $2 \cdot 10^{-6}$ (see Figure 7.3b). We computed a flux less than $10^{-5}$ (five times less than computed by Shadden et al.) with either of the two resolutions. The normalized flux, as defined in Section 7.4 is less than 0.0002.

7.5.1.2 Results for Short-Time FTLE

The goal of [70] was to investigate the sharp FTLE ridges that result from flow maps taken over a long integration time $T$. It is mentioned that FTLE ridges are less suitable for short times $T$. However, in the practice of data visualization the time domain is bounded by the available data sets, and very often it is too short to develop sharp ridges. If ridges are “soft”, the different ridge definitions lead to the extraction of significantly different curves. It is one of the main purposes of this study to compare these definitions and the resulting ridge curves in terms of their geometry and their cross flux which quantifies their deviation from being material lines.

In Figure 7.4, the main ridge of a short-time ($T = 3$) FTLE of the “double gyre” is shown. The height ridge (red), section-based ridge (orange), C-ridge (green) and tensor field line (blue) are all seeded at the local FTLE maximum near the x-axis. The FTLE watersheds (black) are seeded at a saddle point. Since there are several watersheds converging to the same local maximum, the best one in terms of cross-flux was taken for the comparison. The separatrix of the tensor field C (magenta) is seeded at a degenerate point. These do not converge to FTLE maxima as they are not directly related to the FTLE field. For the comparison we took the one separatrix that approaches the chosen FTLE maximum most closely.

The set of 5x5 particles in Figure 7.4 shows that all curves have some particles crossing them in the time interval (1.0, 2.0). Toward the bottom of the image, all feature lines except the tensor field separatrix (magenta) coincide with the two FTLE watersheds (black). The height ridge (red) originating at the FTLE maximum below the bottom of the image curves to the right at $t=1.0$ and later connects with the branch reaching to the left. In either stage the long branch of the height ridge follows closely one of the two watersheds. Also the C-ridge (green) closely follows it. Some of the curves are crossed by fewer particles, such as the y-section ridge (orange), the tensor field separatrix, and in particular the tensor field line (blue) which is consistent with the flux plot in Figure 7.5b for the same time interval.

The evaluation result using our error metric is shown in Figure 7.5 where normalized fluxes are plotted for the various types of ridge curves. In Figure 7.5a fluxes are measured at the lower end of the ridge. Since this ridge is quite pronounced here, most curves nearly
Figure 7.4: Close-up of the “double gyre” with advected particles and the tracked feature lines: height ridges (HR), FTLE watersheds (WS), y-section ridge (YR), C-ridge (CR), tensor field line (TL), tensor field separatrix (TS).

coincide. The exception is the tensor field separatrix which does not run on top of the ridge. In Figure 7.5b where the ridge starts to fall off, it turns out that height ridges, watersheds and C-ridges behave similarly throughout the time domain \( t = 0, \cdots, 10 \).

7.5.2 *Square Cylinder Example*

This vector field is a transient 3D simulation of the flow around a square cylinder (or cuboid) done with Ansys’ finite-volume solver CFX. The cuboid vertically extends only to one half of the computational domain, therefore the fluid (water) can flow over the top and instead of a classical von Kármán vortex street a chain of more three-
Figure 7.5: Fluxes per unit length measured for the various feature lines over the full periodic time range $t = 0, \cdots , 10$ of the “double gyre” field. FTLE ridges: height ridges (HR), $y = \text{const}$ section ridges (YR), C-ridges (CR), watersheds (WS). Field lines of C: seeded at FTLE max (TL), at degenerate point (TS). Integration time used for computing C and FTLE was $T = 3$.

dimensionally shaped vortices appears as can be seen in Figure 7.6a, where vortices are visualized by a $\lambda_2$ isosurface.

We use the full 3D version of the data set for a qualitative analysis. We implemented the tensor field surface as discussed in Section 7.3. Figure 7.6b shows that it is consistent with the FTLE color-mapped on the slice. For the quantitative analysis we use this 2D slice on the half height of the cuboid. The reason is that not all of the feature lines have a well-defined 3D analogue. We cropped the slice to a rectangle of physical size 0.18 by 0.06. Only ridges passing through the seed point as indicated in Figure 7.7 are shown. Since the features extend mostly along the x-axis, we also added ridges based on constant x sections to the comparison. The results are plotted in Figure 7.8, where we selected time steps $t = 0.13$ and $t = 0.14$ and manually picked as the seed point one of the FTLE maxima.

Similar observations to the first example can be made here. Through all structures there is non-negligible flux. The three types of ridges give very similar results with the C-ridge being slightly but consistently better than the other two. The tensor field lines which geometrically deviates from the ridges has similar repulsion strength, but the cross-flux differs for some of the time steps quite significantly, e.g. at
Vortices (isosurfaces of $\lambda_2$).

C-ridge surface and FTLE in a slice at half height of the cylinder.

Figure 7.6: Flow around a square cylinder.

t = 0.14 there is almost no flux. The time steps $t = 0.13$ and $t = 0.14$ are shown in Figure 7.7.

Summarizing the evaluation of both examples, we observe that both tensor field lines and FTLE watersheds can drift away at some point from an FTLE ridge. This is because these methods are based on integration and lack a local correction toward a maximum. With the proposed error metric, tensor field lines cannot be qualified as better or worse than FTLE ridges. We included ridges based on axis-aligned sections only for the purpose of comparison. Their simple definition and the fact that the FTLE structures happen to be roughly oriented in axes directions makes them useful as a reference.

Section-based ridges with sections taken perpendicular to the minor eigenvector of $C$ are, however, a valid alternative. Not only can C-ridges be computed with just first derivatives of FTLE (which is sufficient for iteratively finding local maxima) but also the directional...
information contained in $C$ is exploited, which is missing in the FTLE field and therefore in its height ridges. In all cases where watersheds and height ridges are in good accordance, the $C$-ridge was found to be also consistent with them.

### 7.5.3 Francis Turbine Example

This vector field is an unsteady CFD simulation of an entire Francis turbine. High FTLE values can be found at the lower end of the draft tube where it branches in two parts. This region of interest was already used by Sadlo et al. [64] to compute height ridges with an optimized method. Here, we compare height ridges and $C$-ridges in terms of mesh quality. By using the following brute-force method, we make sure that mesh quality is not influenced by any optimization technique.

On a regular grid, the flow map $\Phi$ is computed and from this the gradient $F$, the Cauchy-Green tensor $C$, the FTLE $\sigma$ as well as its gradient and Hessian $H_\sigma$. All derivatives are computed by convolving with derivatives of Gaussian, using $\sigma = 2$. Depending on the choice of height ridges or $C$-ridges, the major eigenvector of $H_\sigma$ or $C$ is computed on all grid nodes. Then, in a loop over the cells, the March-
Figure 7.8: Error of feature lines at selected time steps in the square cylinder example. HR: height ridge, WS: watershed, XR: x = const section ridge, CR: C-ridge, TL: tensor field line, TS: tensor field separatrix.

The Ridges algorithm [23] is applied. This consists of using principal component analysis to convert the eigenvectors into a consistently oriented vector field $\mathbf{e}$ and then computing the zero-level isosurface of $\mathbf{e} \cdot \nabla \sigma$. The Asymptotic Decider [53] method is used for the disambiguation. Finally, all triangles are trimmed at the isoline $\sigma = \sigma_{\text{min}}$ for removing parts below a threshold and at the isoline $(\mathbf{H}_\sigma \mathbf{e}) \cdot \mathbf{e} = 0$ to remove non-ridge parts (“connector surfaces” [18]).

Figure 7.9 shows that while the two types of FTLE ridges generally coincide, the height ridge is much noisier. The standard deviation of the Gaussian kernel is a parameter that affects the smoothness of the ridges. This effect of the scale on ridges can be studied in a scale-space setting [38, 22]. In our study, several fixed values were used (for Figure 7.9 twice the flow map resolution), and the quality difference between the two types of ridges was found to be consistent. It can therefore be attributed to the fact that second derivatives of $\sigma$ are needed in the C-ridge only for the trimming of triangles, but not for the ridge extraction itself.

By computing the flux through various types of ridges of FTLE fields, we confirmed the findings made in [70] and we demonstrated that for shorter integration times, the normalized flux can be 0.1 or more.

We introduced the C-ridge and showed that it is a good alternative to FTLE height ridges, because it proved to be slightly better quite
consistently. The main advantage of the C-ridge is that it only uses eigenvectors of C, i.e. information that is basically available when the FTLE field is computed, and first derivatives of FTLE for the maximum search. Height ridges require second derivatives which is numerically problematic since the FTLE field already required a numerical differentiation.

7.6 APPENDIX: OVERDETERMINED RIDGE DEFINITIONS

In [70] two alternative ridge definitions are used, both of which are overdetermined and can therefore not be strictly fulfilled in general. In definition 2.2 of the second-derivative ridge, condition SR1 says that the curve must everywhere be tangent to the gradient of the FTLE field, i.e., it must be a slope line of $\sigma$. Together with an initial condition this fully specifies the curve, e.g., a watershed of $\sigma$ if a saddle point is taken as the seed point. Condition SR2 also prescribes the tangent direction of the ridge curve at each of its points, this time as the eigenvector of the Hessian of $\sigma$ corresponding to the larger (signed) eigenvalue. An additional constraint is that the other eigenvalue must be negative. Again, condition SR2 specifies the curve up to a seed point. For practical ridges this means that at least one of these conditions holds only approximately. Condition SR2 can be transformed into the standard definition of the (maximum convexity) height ridge [20] by replacing the tangent direction of the ridge by $\nabla \sigma$. Height ridges, if they are sufficiently “sharp”, fulfill SR1 in an approximate sense. A relaxed version of SR1 can be used to post-filter the set of computed height ridges, e.g. by allowing for a certain maximum angle between the ridge and $\nabla \sigma$ [57]. Definition 2.1 of the curvature ridge is overdetermined in the same way as definition 2.2, therefore both definitions must necessarily be relaxed to become practically usable as, for example, watersheds, height ridges or surface creases.
In this chapter, we describe the ridge extraction pipeline, based on the C-ridge as described in the last chapter. The same approach can also be used to extract height ridges, or any other locally defined ridge. An overview of the pipeline is shown in Figure 8.1.

First, the flow map, either in forward or backward direction, is computed. Based on the flow map, its gradient is calculated by a convolution with a Gaussian. From the flow map gradient, the Cauchy-Green tensor as well as the FTLE field are computed. Since the ridge extraction requires both the FTLE field and the Cauchy-Green tensor, these two operations are computed separately. As ridge extraction depends on the gradient of the scalar field, another Gaussian gradient is computed on the FTLE field. Finally, the ridge extraction is started, having as inputs the eigenvectors of the Cauchy-Green tensor and the gradient of the FTLE field. Because of filtering, the FTLE field is also forwarded to the ridge module.

8.1 Computation of the Flow Map

Visualization by LCS starts with the computation of the flow map. The flow map is sampled on a dense regular grid. There are two reasons for this choice: first, the flow map is required in a very high resolution, very often much finer than the underlying simulation data. Second, sampling the flow map on a dense regular grid makes further processing easier as is seen for example in the gradient computation. Note that the actual computation is carried out directly on the underlying simulation data, therefore no re-sampling takes place. For the numerical integration, a fourth-order Runge-Kutta method with automatic step size control is used, and velocity data are linearly interpolated between time steps. The computation is parallelized on all processors/cores of the Central Processing Unit (CPU) to speed up the process.

For point location in an unstructured grid, the cell tree [24] is used. The cell tree is an adaptive cell location scheme. It is similar to kd-trees, but instead of using a hard boundary to distinguish between the two child nodes, the child nodes can overlap, resulting in a soft boundary. This way, cells do not have to be duplicated in child nodes, saving memory and improving performance significantly.

Another issue are in- and outflow boundaries. When long integration times are used, integration often reaches such boundaries. We solve this problem by integrating pathlines across open boundaries.
The velocity vector is kept constant from the point where the boundary was crossed. Even though this is not a physical extension of the flow field, it is the best approximation that can be done without extending the simulation. Note that this simple scheme is sufficient for our case, but for more complex outflow boundaries, the method by Tang et al. [75] might be more appropriate.

### 8.2 Computation of FTLE

Given a flow map sampled on a dense regular grid, there are two stages of numerical differentiation involved in the process of first computing an FTLE field and then its ridge surfaces. The first differentiation is needed in the calculation of the Cauchy-Green tensor $C$, which requires the flow map gradient. In the second stage, the ridge computation, derivatives of the FTLE field are required. Here, second derivatives can be avoided if $C$-ridges instead of height ridges are used. Nevertheless, the whole process is still highly sensitive to numerical precision and noise. The simplest scheme for estimation of derivatives would be finite differences. However, finite differences do not represent a ground truth, and furthermore, they do not provide good estimates for points between grid nodes. The various numerical methods (based on finite differences, polynomial fitting, or other) all include data from a neighborhood, the size of which determines the amount of smoothing. It is important to have enough smoothing to cancel numerical noise, but not to the point where features start to move. Our solution is to use Gaussian derivatives, which are obtained by convolving the data with derivatives of a Gaussian. This way, the flow map gradient as well as the smoothed FTLE field, its first and, optionally, second derivatives are reconstructed. The standard deviation of the Gaussian – the scale in scale-space terminology – is chosen to be
as small as possible, but large enough to avoid discretization artifacts, typically 1.5 to 2.0 times the grid resolution of the flow map. If the kernels are cut off at 4.0 standard deviations, stencils of $15^3$ voxels have to be used. This makes convolution an expensive operation that adds significantly to the overall computing time even if separability of the Gaussian kernel is respected.

A much faster way is to make use of the convolution theorem and compute the convolution in frequency space. The Gaussian kernel is separable and therefore the gradient components can be calculated independently. Fast Fourier Transform (FFT) and its inverse are available in many different implementations. For our implementation the CUFFT library is used, which is released as part of CUDA [55]. The 3D Gaussian kernel with equal standard deviation across all dimensions centered around the origin is

$$g(x) = \frac{1}{\sqrt{(2\pi)^3 s^3}} e^{-\|x\|^2/(2s^2)}$$

where $s$ denotes the standard deviation (to avoid confusion with the symbol $\sigma$ used for the FTLE). Its gradient is then

$$\nabla g(x) = -\frac{x}{s^2} \frac{1}{\sqrt{(2\pi)^3 s^3}} e^{-\|x\|^2/(2s^2)}$$

The Fourier transform of the components of a gradient field $\nabla f(x)$ can therefore be computed by a component-wise multiplication of $\text{FFT}(f(x))$ and $\text{FFT}(\nabla g(x))$.

The Discrete Fourier Transform assumes that the input data is periodic, i.e., the convolution kernel wraps around the data borders. This leads to periodic artifacts at the boundary. To solve this problem, the input data is enlarged in each dimension by the size of the kernel. Frequently, the data is extended using zero-padding. However this would lead to high gradient values at the border which in turn would result in spurious ridges. Therefore, the data is extended using constant propagation, i.e., they have the same value as the closest boundary value. This way, the effect from periodic convolution is suppressed and usable results are achieved at the domain boundaries.

While the convolution-based scheme is generally applicable, its FFT optimization requires that the entire sampling grid is within the domain. When the underlying data are unstructured grids, the flow map grid can overlap “thin” interior boundaries, such as a wall separating two separate zones. Such discontinuities in the gradient estimation lead to artifacts, which can however be effectively eliminated within the ridge extraction step, as is described in the next section.
8.3 Ridge Extraction

For the computation of a C-ridge surface, there exist different approaches. One is to use a seed point at a local FTLE maximum and compute the radial extension of the surface from this point, see Section 7.2. Using the major eigenvector $\mathbf{N}_1$ of $C$ as a prediction for the ridge normal, the ridge surface is extended at its current front. The newly added vertices undergo a correction step in the direction of the local eigenvector $\mathbf{N}_1$, maximizing the FTLE value $\sigma$. Integration is stopped at vertices where $\sigma$ drops below a given threshold.

Such a seed-based method is especially useful for extracting a single ridge and tracking it over time. The alternative approach, which is our choice, is based on the Marching Ridges algorithm [23]. It can generate both height ridges and C-ridges.

Before the Marching Ridges procedure starts, the tensor $C$ and the FTLE value $\sigma$ are computed on each grid node, as is described in Figure 8.2. Also, depending on the chosen ridge type, one or two derivatives of $\sigma$ are computed. Finally, an eigenvector $\mathbf{e}$ is computed per grid node, which is $\mathbf{N}_1$ in the case of a C-ridge and the minor eigenvector of $\mathbf{H}_\sigma$ in the case of a height ridge.

Now the Marching Ridges algorithm starts looping over all grid cells. A principal components analysis (PCA) is performed on the eigenvectors (and their negatives) at the eight nodes of the hexahedral cell. The first principal component is used to consistently orient the eight eigenvectors of the tensor. These are then used to compute the scalar product $\mathbf{e} \cdot \nabla \mathbf{f}$ corresponding to the first part of Equation 18. From these scalar values at the eight nodes, the zero isosurface is computed with the standard Marching Cubes [47] method.

The described algorithm is in most parts the same for C-ridges and height ridges. A major difference is that height ridges require second derivatives of $\sigma$ already in the computation of ridge vertices, not just in the subsequent trimming of the ridge surface. This is the reason why height ridges are more expensive to compute and typically give noisier results than C-ridges.

8.4 Ridge Filtering

As with most feature extraction algorithms, some post-processing has to be applied to the extracted (or raw) features. There are several important filtering steps applied in order to make the output meaningful.

Valley Line Detection The algorithm presented results in a set of triangles the vertices of which satisfy the first part of Equation 18, but which may be valley points instead of ridge points. That means that some vertices, together with parts of the triangles, have to be
Triangles RidgeSurface(FlowMap Φ, bool ridgeMode)
// Input: flow map sampled on a regular grid
// Output: non-oriented triangles

// 1. Compute flow map gradient
Φ ← FFT(Φ)
foreach grid node do Ŧ ← FrequencyDomainGradient(Φ)
  F ← InverseFFT(Ŭ)
// 2. Compute C and FTLE (σ)
foreach grid node do C ← Fᵀ · F, σ ← FTLE(C)
// 3. Compute FTLE gradient and Hessian if required
σ ← FFT(σ)
foreach grid node do
  ∇σ ← FrequencyDomainGradient(σ)
  if (ridgeMode) then Ḥσ ← FrequencyDomainHessian(σ)
∇σ ← IFFT(∇σ)
if (ridgeMode) then Hσ ← IFFT(Ĥσ)
// 4. Calculate eigenvectors
foreach grid node do
  if (ridgeMode) then e ← MinorEigenvector(Hσ)
  else e ← MajorEigenvector(C)
// 5. Extract Ridge
foreach grid cell do
  N ← set of eight adjacent nodes
  E ← set of eight eigenvectors e at nodes in N
  p ← FirstVector(PrincipalComponentAnalysis(E))
  // 5.1 Orient eigenvectors consistently using PCA
  foreach e in E
    if e · p < 0 then e ← −e
    s ← e · ∇σ
  (V, T) ← MarchingCubesVerticesAndTriangles(s, level = 0)
// 5.2 Classify vertex (ridge or valley)
foreach vertex v in V do
  Hσ ← spatialDomainHessian(Φ, v)
  (σ, e) ← interpolationOnEdge(σ, e, v)
  d ← (Hσe) · e  // negative for ridge, positive for valley
// 5.3 Output and Trimming
  GenerateOutputTrianglesAndTrim

Figure 8.2: Ridge surface algorithm. FTLE is defined by Equation 17. IFFT is an abbreviation for Inverse FFT.
trimmed. In order to decide this for a vertex at position \( \mathbf{x} \), the first and second derivatives of \( \sigma \) are reconstructed at \( \mathbf{x} \) and the second part of Equation 18 is evaluated. Here is the only place where second derivatives of \( \sigma \) are needed in the case of C-ridges. We avoid them by using two test points \( \mathbf{x} \pm \mathbf{h} \mathbf{e} \) roughly orthogonal to the ridge where \( \mathbf{h} \) is the grid resolution and \( \mathbf{e} \) is the eigenvector reconstructed at \( \mathbf{x} \). A local maximum can be checked by reconstructing \( \sigma \) at these test points and comparing it with the value at \( \mathbf{x} \). This way, we can compute a scalar which is an approximation of the second derivative of \( \sigma \). This scalar can then be assigned to every vertex in the extracted mesh. The final trimming is then done using the method described in Section 4.1.4.3 from the Smoothed Particle Hydrodynamics (SPH) isosurface extraction algorithm.

**Filtering based on FTLE value** Ridges can also be recorded where the FTLE value is very low. In such areas, the amount of separation (or contraction) is low and the LCS is regarded as noise. Again, the method described in Section 4.1.4.3 from the SPH isosurface extraction algorithm is used to trim the raw features. A user-defined threshold \( \sigma_{\text{min}} \) is used to define the point of trimming.

In case there are triangles, it can happen that a triangle needs to be trimmed because of the FTLE value as well as because the ridge becomes a valley. To deal with this issue, the cutaway point on a triangle edge is computed for both criteria, and the one closer to the valid vertex is kept. This way, an arbitrary amount of filtering criteria can be supported this way.

**Angle-based filtering** As already noted, C-ridges can be regarded as a variant of the weak LCS concept from [30]. However, the weak LCS definition is over-constrained and therefore not directly usable (see Section 7.6). The solution is to relax conditions, such that its definition becomes practicable. We chose to relax condition 2, where the eigenvector of the Cauchy-Green tensor must be orthogonal to the tangential space of the LCS. It is relaxed, such that it is the angle between the LCS normal and the eigenvector of the Cauchy-Green tensor must be below a user-defined threshold. This is also motivated from [57] where it was shown that in case of height ridges, the gradient can significantly deviate from the surface tangent. An example showing the effectiveness of this trimming can be seen in Figure 8.3.

**Non-orientability of ridge surfaces** A problem arises in the process of rendering ridge surfaces. A triangle with vertices \( v_1, v_2, v_3 \) has a positive face, on which the vertices appear in counterclockwise order, and a negative face, on which they appear in clockwise order. Since ridges are based on eigenvectors (of either the Hessian or the Cauchy-Green tensor), the resulting triangle mesh will not
have consistently oriented triangles. A possible solution to this problem is traversing the mesh using breadth-first search and flipping triangles that are not consistent. A problem occurs when the mesh is not orientable, e.g., has the topology of a Möbius strip. The border between differently oriented triangles can, however, be easily detected by the described breadth-first search. The mesh is then cut at this border, which results in a consistently oriented mesh. The vertex normals for border vertices can be correctly computed by taking all incident triangles and their orientation into account. The resulting mesh can then be rendered without artifacts using two-sided lighting.

**Simulation Geometry** One problem of the described approach used here is the use of a dense regular grid, without considering the underlying simulation geometry. With unstructured grids or SPH simulations, there usually exists boundary geometry. With unstructured grids in particular, the dense regular grid can overlap thin walls used to separate the flow in two separate compartments. With the use of Gaussian gradients, these boundaries are ignored and therefore gradient values are computed using completely unrelated flows. The discontinuity in the data results in high gradient values, which the ridge extraction algorithm wrongly treats as LCS. To solve this problem, we introduce a secondary array allowing us to mark individual grid cells as invalid. All cells intersecting with the building geometry are marked as invalid. The ridge extraction algorithm then respects this information by only considering cells which are not marked as invalid. Although this solution is not mathematically rigorous, experience has shown that it very effectively eliminates spurious ridges.
8.5 Performance Analysis

For the performance analysis, the data from the application case in Chapter 9 was used. The algorithm presented here was implemented in the Visdom visualization framework [86]. All performance measures were carried out on a 2x Quad Core Xeon E5430 machine with 64GB of RAM running Linux. The GPU used is an NVIDIA GTX 580.

8.5.1 Flow Map Computation

The computation of the flow map is by far the most expensive part for any FTLE-based visualization. Therefore, the computed flow map was saved to disk to facilitate further repeated processing. However, as a result of this, the integration time is fixed for the entire analysis. The integration time was chosen to be three seconds which is slightly more than one period of the simulation. Since one period consists of the door only rotating by 90 degrees, periodic artifacts are not expected. The computation of the flow map is carried out directly on the unstructured grid (without any resampling). The flow map is sampled on a regular grid with dimensions $140 \times 120 \times 80$. These numbers were chosen to prevent flow map computation in uninteresting regions while retaining equal spacing in all three dimensions. Also, it should be pointed out that no adaptive computation of the flow map [65] was used.

The computation of one single flow map takes about 4 minutes, resulting in 8 minutes of computation per time step when also accounting for the backwards flow map. This resulted in a total of 37 hours of computation for all of the 280 time steps. Because the flow map for the entire data is precomputed, this rather large amount of time does not impact the data analysis process by engineers.

8.5.2 Ridge Extraction

C-Ridges are extracted using the FTLE gradient and the major eigenvector of the Cauchy-Green tensor, as seen in Figure 8.1. The performance numbers for every step in the pipeline can be seen in Table 8.1. The difference in performance between FTLE gradient and the flow map gradient is explained by the fact that the flow map gradient requires the differentiation of three components while the FTLE field is simply a scalar.

The total extraction time from loading of the flow map to the ridge geometry is in the area of six seconds (when counting both forward and backwards ridges), which is more than fast enough for practical application.
<table>
<thead>
<tr>
<th>Step</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow map gradient</td>
<td>224</td>
</tr>
<tr>
<td>Cauchy-Green tensor</td>
<td>461</td>
</tr>
<tr>
<td>FTLE computation</td>
<td>653</td>
</tr>
<tr>
<td>FTLE gradient</td>
<td>119</td>
</tr>
<tr>
<td>Ridge extraction</td>
<td>1324</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2781</strong></td>
</tr>
</tbody>
</table>

Table 8.1: Performance numbers for the steps of the visualization pipeline.

8.5.3 **FFT-Based Gradient vs. Explicit Convolution**

We compare the performance of our FFT-based gradient implementation to our initial convolution-based implementation. The FFT-based version is implemented on the GPU using CUFFT while the convolution-based version is single-threaded CPU code. The FFT based version is of course expected to be faster as it has an asymptotically better runtime than direct convolution ($O(n \log n)$ vs. $O(n^2)$). Even though we are aware that comparing such vastly different approaches is questionable, the output of both algorithms is the same and, more importantly, they took a similar amount of time to implement.

With variance set to 2, the FFT-based implementation took 224ms as seen in Table 8.1. In comparison, the code implementing direct convolution took 6 minutes and 42 seconds. This enormous boost in performance is what makes on-the-fly extraction of ridges possible without any preprocessing other than the flow map. Storing the flow map on disk is therefore a good compromise between computation time required to extract LCS as well as minimizing disk space.
9.1 Introduction

Figure 9.1: (a) A revolving door, as is typically used in building entrances. (b) The simulation of the air flow near such a door reveals how large swaths of cold air enter the building, incurring energy losses. Image (c) shows how the addition of an air curtain blowing warm air blocks cold air, preventing it from cooling down the interior. This, however, complicates the flow structure greatly so that neither temperature plots nor contours, path- or streamlines really help in understanding the flow behavior. The solution to this problem are LCS (d), giving structural information about this complicated time-dependent flow.

In this chapter, we apply the results from the previous chapters to analyze the simulated air flow near a revolving door equipped with an air curtain. Energy efficiency in building technology has become a major topic when dealing with today’s climate challenges. Several types of simulations provide the possibility of analyzing and improving the energy efficiency of a building prior to construction, including Computational Fluid Dynamics (CFD) for simulating the indoor air flow. In many countries, standards exist for so-called passive houses or, more generally, low-energy-consumption buildings. Some authorities subsidize private buildings if they meet such a standard, while
for public buildings, a standard can be declared mandatory. One important factor for energy efficiency is building climate control and, in particular, efforts to minimize energy loss through air exchange.

In building climate control, preventing air draft is one of the foremost issues when trying to reduce heat leakage. It occurs when there are two openings to the outside within a building. The resulting draft not only causes warm air to leave the building but also cold air to enter the building. Specially shielded windows and doors are commonly used and prevent most of the leakage when closed. Air exchange systems are then put in place, which supply the interior with fresh air constantly and at much lower energy losses than if windows were opened to get fresh air into the interior.

In large public buildings such as shopping malls, this task becomes more difficult as people constantly move in and out. In such cases revolving doors are frequently put in place as they prevent direct air exchange with the outside. They also have the advantage of not letting in snow, rain or dust.

Another common technique to reduce draft is air curtains placed inside the doors, usually blowing warm air from the side or from the top. Air curtains can significantly reduce the flow of incoming cold air as well as of the outgoing warm air. This is also important for the health of people working in proximity of these doors.

Both of these approaches, taken alone, have certain disadvantages. Air curtains, used with sliding doors, do not prevent the direct contact of the warm air inside with the cold air outside. In large buildings where these doors open frequently, enormous energy losses still occur. Also, the placement of the curtains is difficult. A small change in the environment can result in pressure gradients bending the air curtain. That in turn will stop the air barrier from working, and cold air can enter the building. With revolving doors alone, the problem is that cold air is mixed with warm air inside the doorway and released into the building periodically. Even though draft is prevented, heating is required to compensate for the heat loss due to mixing.

In combination, the two technologies of revolving doors and air curtains have been shown in CFD simulations to achieve significantly better energy saving results [7]. The use of air curtains is especially interesting in buildings adhering to a low-energy-consumption standard, even though it seems to contradict intuition.

In this chapter, we tackle the problem of analyzing the simulated air flow near a revolving door equipped with an air curtain. The air flow to be analyzed and visualized has a complicated time-dependent structure, because it is the result of both convection and the pump-like effect of the revolving door. A visualization of the temperature alone, by means of slices or isosurfaces, shows the temperature distribution over time. This is one of the end results the engineer is most interested in, besides energy loss and velocity of the draft inside the
Figure 9.2: Volume flow rate of cold air as a function of the temperature difference, from [7].

building. However, the cause of this temperature distribution is not explained by its visualization. Instead, the visualization needed to reveal structures in the velocity field. Vector field topology is a candidate, but it is known to not give much insight in flows which are sufficiently unsteady [70]. Even though the air flow caused by a revolving door in combination with an air curtain is nearly steady in some areas, it is highly unsteady around the air curtain, which is the region of primary interest. As a result, vector field topology is not an ideal candidate for the visualization of such simulations and LCS should be used instead. We will show how LCS add to a better understanding of flow patterns which are relevant as they affect the efficiency of the air curtain. For a brief overview of the application case, see Figure 9.1.

9.2 THE APPLICATION CASE

9.2.1 Revolving Doors and Air Curtains

The first patent filed on revolving doors was in 1888. Revolving doors were primarily designed to prevent the “entrance of wind, snow, rain or dust” and to improve efficiency because of the “reduced possibility of collision” [17]. Today, they are mainly used to minimize heating effort by preventing air drafts [8]. In a building which is not airtight, a revolving door reduces the draft by roughly a factor of three when compared to a sliding door, and still a factor of two if the building is airtight, see Figure 9.2. The prevention of strong air drafts also increases comfort for people working in close proximity such as receptionists. However, revolving doors do not prevent cold air from
entering the building, they only prevent air drafts. This is confirmed by a simulation of a revolving door with two wings as seen in Figure 9.1(a). Close to the floor, large amounts of cold air is cooling the inside; heating is needed to compensate this heat loss. A possible solution to this problem is the usage of air curtains. Air curtains are a common measure to create a barrier between the cold air on the outside and the warm air on the inside. They blow warm air typically at rates of \(3\text{–}7\,\text{m/s}\) from the side or from the top, but, depending on the environment, they can also be made stronger. When combined with sliding doors, air curtains have the air outlet and intake arranged in such a way that a roll of warm air or a pair of counter-rotating rolls is generated. This blocks cold outside air and thus reduces draft through the open door, which, in turn, reduces energy loss by approximately 50 percent [67]. For the combination with a revolving door, a different design has been chosen by our application partner (Air Flow Consulting AG): the air curtain blows warm air from the floor, such that it follows the natural convection direction. This leads to a more stable curtain at lower energy consumption. Figure 9.1 shows an overview.

9.2.2 Simulation Data

The simulation of a revolving door with an air curtain was carried out by our industry partners using ANSYS CFX solver [4]. It spans 20 seconds after which a quasi-periodic state is reached, in other words, the flow pattern repeats itself after one period of rotation of the door wings. The temporal resolution of the saved data is 0.1 seconds. The underlying mixed unstructured grid contains approximately 1.65 million cells. The simulation consists of three main parts; the interior, the exterior and the door. The interior and exterior are modeled like a very large room, large enough such that air can flow freely from the door into the in- or outside. The door is 2.5 meters in height and the width of a single door wing is 1.24 meters. The initial goal of the simulation is to test the energy savings when an air curtain is used in combination with a revolving door instead of a sliding door. The simulation does not take into account people going through the door. The revolving door itself has four compartments, which is a very common design for low-traffic doors. The air curtain is idealized by an air outlet with constant velocity and temperature. The air outlet (of the device) is modeled as an inflow boundary of the simulation domain.

9.3 Complementary Visualization

Along with the LCS, several basic visualization techniques complement the analysis of the air flow. Slicing planes or isosurfaces of the temperature help to understand the temperature distribution. The
temperature certainly is the most fundamental field to look at for the engineers, however, it is important to realize that temperature alone does not provide the information contained in LCS. Because the flow is not caused by convection alone, but actively influenced by the door and the air curtain, temperature contours and LCS are not necessarily aligned. This can be seen best inside the sectors of the revolving door, where the temperatures form layers that are roughly horizontal, while LCS are not aligned with these layers. In other words, a picture of the temperature distribution does not explain what caused it. Therefore, the velocity field, which is a product of convection and the door’s pumping effect, has to be studied. LCS can give valuable insight into the structure of the flow. However, complementary information is usually needed because LCS do not provide important information such as the current flow direction. Therefore, we also use pathlines and particle advection. Moreover, these techniques are also used to analyze the flow behavior at places where no ridges are present. To give the viewer a feeling for the surroundings, a reduced version of the door geometry is also rendered transparently.

9.4 RESULTS

In this section, we discuss our analysis of the air flow using LCS. We start by inspecting the temperature distribution and discuss its limitations. Then, we extract the basic topological structure of the airflow in its quasi-periodic state. We show how the air curtain fails to fully shield off the interior by exposing a small heat leak. Two alternative scenarios are then studied, which attempt to close this leak.

9.4.1 Temperature Distribution

The temperature distribution is naturally the first thing the engineer will look at when simulating such a scenario. For that, we chose to sample the temperature on a plane, as is shown in Figure 9.3. As can be clearly seen, the cold air at the bottom of the door is held off effectively by the air curtain. This is an important first insight, confirming the initial idea of combining air curtains and revolving doors. Also the significant amount of warm air escaping the doorway is visible. An air curtain blowing cold air from the top would help here, however it is not clear whether the gain in efficiency would outweigh the effort of another curtain. Also, according to our domain experts, legal requirements can prohibit such a design.

The visualization of the temperature distribution gave a first helpful overview of the simulation. However, it is also clear that there is little information gained on the air flow structure. For example, the origin of the warm air exiting the building can only be guessed. The air curtain as well as the interior are both potential sources.
Figure 9.3: The temperature slice at $t = 12.35$ shows that the air curtain manages to hold off the cold air. But it is also evident that no information about the flow structure is gained this way.

9.4.2 General Overview using LCS-Based Visualization

A rendering of the C-ridges at the beginning of one period is depicted in Figure 9.4. Attracting LCS are colored in red, while repelling LCS are colored in blue. The rendering uses adaptive transparency and a dense flow visualization [12] to make it easier to understand the structure of the LCS. We focus on the area around the air curtain and ignore the ridges originating from warm air escaping to the exterior. There are three main structures here: the large attracting LCS originating directly from the air curtain marked (1), the periodically occurring “shovels” (repelling LCS) moving upwards (2), and the bubbles at the floor (3). For illustration purposes and because interpreting LCS is not easy, we look at cross sections of the ridges as seen in Figure 9.5 and Figure 9.6. Of central importance for the air curtain is the blue repelling ridge labeled (1) in Figure 9.5. Air on the left of that ridge remains within the door, prevented from entering the interior. On the other side, the air is pulled up by the air curtain, warmed up and, depending on the time within the period, it is brought back into the door or released into the interior of the building. The temperature of the air curtain manages to heat up this air so that no temperature drops are visible. Going back to the 3D visualization in Figure 9.4, we quickly see that this separation not only works in the middle of the entrance, but across the entire opening.

The repelling LCS marked (2) in Figure 9.5 tends to break apart within a period, as seen in Figure 9.6. It separates air from the air curtain which is released into the door from air released into the interior of the building. If it would not break apart, particles left of this ridge would always end up within the door area, which has been shown not to be the case in the particle animation.
9.4 results

Figure 9.4: General overview of the ridge structure. Red and blue surfaces depict attracting and repelling LCS, respectively. Ridges were trimmed at an FTLE threshold of 1.5. The large attracting structure marked (1) originates from the air curtain. The repelling structures (2) originate from circulating structures in ascending air. The repelling structure (3) can be seen as a simulation artifact originating from the idealized air intake.

9.4.3 Heat Leak

Even though the ridges indicate a near-optimal separation, the provided visualizations suggest that more improvements can be made to the design in order to further optimize energy saving. In Figure 9.7 a close-up of the ridges near the door as well as four pathlines are shown. The absence of ridges in the area between the curtain ridges and the revolving door is notable. This indicates that no shielding takes place there and that the flow behavior should be investigated in more detail. We therefore seeded a few pathlines in the interior close to this area. These pathlines move directly into one of the sectors of the revolving door. This is caused by the pressure gradient resulting from the rotating wings. As a result, warm air from the interior gets pulled out, causing heat leakage.

It might seem surprising that we were most interested in areas with an unexpected lack of LCS. LCS, or the lack thereof, really proved their usefulness here as they made it possible to find and localize such a heat leak efficiently. Even though we used pathlines to confirm our finding, using only pathlines would have forced the engineer to search the entire domain.
Figure 9.5: Cross section of the LCS showing the flow structure in the symmetry plane of the door. From repelling (blue) and attracting (red) LCS the motion of the flow relative to the moving LCS can be inferred. 1) The four arrows on the left show the flow barrier created by the air curtain. 2) The two arrows on the right show circulation within the ascending warm air.

Figure 9.6: Sliced view of the LCS over the time span of the door rotating 90 degrees (one period). Pairs of repelling (blue) and attracting (red) LCS form flow barriers over the full height, which separate air flows in the door and in the interior. The LCS marked * are within the air curtain, indicating that parts of the warm air enter the door area shielding the inside and warming up cold air coming from the outside. The LCS marked B splits into two parts between 30 and 60 degrees.

9.4.4 Fixing the Heat Leak

Our results were met with great enthusiasm by our industry partners. As a result, our domain experts reran the same simulation with two small modifications in an attempt to fix the described heat leak. The changes are highlighted in Figure 9.8. In the first case, the air curtain
Figure 9.7: (a) The air curtain fails to seal off the interior completely. Warm air behind the door can exit through the door, as indicated by an area of missing LCS. Pathlines confirm this behavior. The reason why some pathlines cross the attracting LCS is that pathlines depict an entire time interval, during which the LCS move. Here, LCS are shown at the time of pathline seeding. (b) The top view shows the door in its position at the seeding time. Particles enter the door sector at the suction side of the wing marked with “S”. Images are taken at time step $t = 15.1$.

is extended at the left side in order to better shield off the leak. In the second case, a small flap is added to the top of the door entrance with the idea of blocking air entering the door due to the pumping effect.

The results can be seen in Figure 9.9 for the first case, in Figure 9.10 for the second case. In the first case, the LCS nicely shield the entire door area. In both cases, the size of the leak is minimized greatly, meaning that the amount of air sucked into the doorway is much smaller.

Our LCS-based visualizations led to two design modifications, both of which eliminate the largest heat leak. Whether the modified designs also lead to energy saving, is not yet clear. A calculation of the total volume flux, integrated over a whole period, shows only a small effect within the band of fluctuations. While a refined analysis will be needed, it can be stated that our visualizations not only helped understanding the air flow but can actively influence design decisions for the building of energy-efficient doors.
Figure 9.8: The additional simulations performed in order to minimize the heat leak. The highlighted areas denote the modifications done to the original case. *Left:* The air curtain is extended on the left side. *Right:* A small flap depicted in blue is added in order to avoid the leak.

Figure 9.9: Results for the first modification case (longer air curtain). *Left:* the LCS cover the entire door, leaving only a small gap as opposed to the original case as seen in Figure 9.7 (a). *Right:* pathlines seeded as in the original case (Figure 9.7) are diverted and stay in the interior.

Figure 9.10: Results for the second modification case. *Left:* The repelling LCS (blue) closely follows the door border. There is a hole in the red attracting ridge, however it does not extend to the entrance. The pathlines are also an indication of this.
In this part, we presented our research centered around LCS. First, we presented two error metrics for numerically computed LCS in unsteady velocity fields. By computing the flux through various types of ridges of FTLE fields, we confirmed the findings made in [70] and we demonstrated that for shorter integration times the normalized flux can be 10% or more. We introduced the C-ridge and showed that it is a good alternative to FTLE height ridges, because it proved to be slightly better quite consistently. The main advantage of the C-ridge is that it only uses eigenvectors of C, i.e., information that is basically available when the FTLE field is computed, and first derivatives of FTLE for the maximum search. Height ridges require second derivatives which is numerically problematic since the FTLE field already required a numerical differentiation.

Based on the introduced C-ridge, we presented a full pipeline for the efficient extraction of LCS. By using FFT, we were able to speed up gradient computation enormously. Also, we explained the process of filtering ridges in great detail. As with any feature extraction algorithm, filtering plays a vital role in producing meaningful output.

Finally, the ridge extraction pipeline was applied to real-world data, the analysis of a revolving door combined with an air curtain. The extracted LCS ridges helped in getting a deep understanding of the flow structure around the air curtain, which would not have been possible by looking at temperature plots alone. Foremost, the results obtained led to the conclusion that the air curtain is capable of holding off cold air from entering the building over a large fraction of the area of the door. The LCS ridges revealed heat leaks above the air curtain which we then verified interactively using pathlines. These results were used to further improve the design, significantly reducing the size of the mentioned heat leak.

The most immediate limitations is that LCS-based visualizations tend to be cluttered. We solved this by using an illustrative visualization technique with adaptive transparency and by looking at cross sections. Still, the resulting visualizations take time to understand, and interactive viewing is advantageous. On the ridge extraction side, the time required for the flow map computation can be a prohibiting factor.
A visualization system is the software on which visualization algorithms are built. Traditional visualization systems load simulation data from disk which is then processed to produce graphical output. Scientific visualization is used in numerous application areas. Therefore, visualization systems have to be highly modular in order to suit the varying requirements of its users. To achieve such modularity, visualization systems mostly use the data flow design pattern which treat algorithms as a black box consuming some input in order to produce output. The algorithms can then be connected together to form a data dependency graph which can then be processed.

For Lagrangian methods, simple data flows however do not suffice. The reason is, that in typical Lagrangian visualization, such as Lagrangian Coherent Structures (LCS), the dependencies of an algorithm are not specified by the data flow alone, but have an additional time parameter. As a result, the handling of time-dependent data in data flow systems is a challenging problem. By solving the problem of specifying temporal dependencies, it suddenly becomes very easy to integrate simulation components into the data flow architecture. The merge of simulation and visualization is useful for several reasons. From a user perspective, it is no longer necessary to learn two separate tools. Also, the requirement of common data formats vanishes. More importantly, the merge opens up new possibilities: it is possible to examine or even alter the simulation state even while the simulation is still running. By altering the outcome of a simulation, a new simulation branch is created, which coexists with is origin. This makes it very easy to compare different outcomes and study the effect of different parameters.

In the following chapters, we demonstrate a system which is capable of achieving the goals mentioned above.
11.1 DATA FLOW SYSTEMS

The data flow design pattern is a well-known and widely-used design pattern in software engineering. It is used, when computations depend on data or computations which are not known in advance or determined by users. In compiler design for example, data flows are used to describe the transformations applied to data as the program is running. Another well known-example are spreadsheets with formulas. When the content of one cell changes, all cells with formulas containing the value of the changed cell will have to update their content.

Data flows are represented using Directed Acyclic Graphs (DAGs). Nodes represent computations whereas the arcs represent the dependencies. In order to execute a dataflow, nodes are processed in topological order. Data flows only mandate a certain data format which nodes use to communicate with each other. Other than that, data flows do not impose any restriction on what they do with the data making them highly modular. This makes the data flow design pattern attractive to visualization systems.

The usage of data flows for visualization was first published in 1989 by Upson et al. \[77\] for the visualization system AVS \[3\]. In visualization systems, the data flow pattern is used to fully separate computations \[28\]. A node in a data flow roughly represents a visualization algorithm. This way, the algorithm becomes independent of how the data is loaded from disk or what preprocessing has to be applied. Following its initial publication \[77\], data flows have become the de-facto standard approach for visualization systems and are used for example in the Iris Explorer \[83\], the Visualization Data Explorer \[33\] and Visualization Toolkit (VTK) \[82\].

11.2 TEMPORAL PROCESSING IN DATA FLOWS

The handling of time-dependent data in data flow systems is a surprisingly difficult problem. Many systems solve it by having a global state denoting the time value the system is using at this moment, or by selecting a time value in file readers which determine the data to be processed. This, however, has a number of implications: it becomes for example impossible to have two output windows, each displaying a different time step, which is useful for comparison. But more importantly, it makes it impossible to implement time-dependent visualiza-
tion algorithms such as pathlines, as computing them requires not just one, but a large number of input time steps. Adding simulation capabilities to data flow systems is problematic for the same reasons: If a certain simulation state is requested, the system needs to make sure that the previous states are already computed. This dependency chain can then be continued recursively.

The fundamental problem is therefore, that the data flow graph insufficiently represents the dependency graph for such problems. VTK [82], including VisIt [19] and Paraview [39], are based on a demand-driven pipeline as opposed to the data-driven model where the data only flows downstream. In the demand-driven approach, users can request data from the sink nodes (nodes, on which no other node in the data flow depends). This information is then communicated to nodes that are further upstream through an additional data flow process. Biddiscombe et al. [9] use this approach to support nodes which perform complex temporal processing. In this system, nodes can request – and consequently access – multiple data items of different time steps from nodes that are further upstream. To realize this, the authors suggest a combination of downstream and upstream processing. However, their solution is incompletely specified. The way a node with two input nodes is dealt with, is left to the node programmer. Also, integrating simulation capabilities in their system is not possible as nodes can only specify dependencies on their input nodes, but not on themselves. Because simulation capabilities and time-dependent calculations have many commonalities, it seems logical to bring the two concepts together and solve the problem in a more fundamental way.

11.3 Simulation and Visualization

A recent trend is the integration of simulation and visualization tools. There are several motivations for this. First, the data can be analyzed using the same tool as it was created with, making the process more intuitive for users. As a result, there is no need to agree on common specific formats in order to exchange the data. Also, visualization can make use of intermediate data which is often thrown away to save disk space. Lastly, it makes it more immediate to use the same hardware for both visualization and simulation.

The merge, however, opens up new possibilities too. It becomes possible to react to the simulation state even while the simulation is still running. Parameters can be altered, changing the outcome of the simulation, as done for example in Visdom [2]. Changing parameters equals to creating an alternative scenario which can then be used for comparing the outcomes. In Visdom, the tool used to achieve this are the World Lines [86].
11.4 **WORLD LINES**

The World Lines [1, 86, 87] are an interactive view for exploring alternate scenarios. They were developed for a system with an integrated simulation component. Using World Lines, it becomes possible to change simulation parameters on the fly, creating alternate outcomes. Alternate scenarios are represented using tracks sharing a common time line. They are created by changing some parameter, changing the outcome of the simulation. Such an event is visualized by a branch in the selected track. The child track represents the outcome with the changed parameter applied while the original does not see the parameter change. Figure 11.1 highlights the most important concepts behind the World Lines view.

The simulation outcome can be reported by statistical or graphical means using separate linked views or alternatively also by inline views in the World Lines themselves. With this information, the user can use this information to create further alternate scenarios, closing the feedback loop. Because the World Lines represent the entire history of decisions done by the user, it becomes possible to come back to an older state at any time in order to test new alternate scenarios. The World Lines were presented with a flooding scenario in mind, where several protection measures are tested in order to prevent or minimize the damage in a city in the case of flooding. The approach however is not limited to flooding but can be used with any kind of simulation.
12.1 Introduction

The handling of time-dependent data in data flow systems is a surprisingly difficult subject. For example, computing the path of a massless particle in a flow requires all input data spanning the requested time interval. This additional dependency seems unnatural for data flow systems because data flows model computation order through connections only. In AVS [3] for example, the time value is a global state, thus making it impossible to compute such particle traces. When adding simulation capabilities to data flow systems, a similar problem arises. If a certain simulation state is requested, the system needs to make sure that the previous states are already computed. Again, a simple data flow connection insufficiently represents the additional temporal dependencies. The situation gets even more complicated if decision support systems are used to analyze alternative scenarios. The results of a module can depend not only on multiple time values, but also on multiple alternate parameter values.

In this chapter, we show how we solve all of the above problems using generic data flow algorithms. The user can introduce various simulation, analysis and visualization functionalities as simple data flow modules, with the arising dependencies resolved behind the scenes. For example, if asked to produce pathlines, the system can automatically generate the data required by using an integrated simulation on the fly. The same algorithms allow World Lines [1, 86, 87] (Figure 12.1b) to be used in classical visualization scenarios by loading pre-simulated data.

We exploit the data flow modularity to extend a levee breach scenario (Figure 12.1c) with analysis and visualization nodes to investigate the material transport through the water using pathlines (Figure 12.1d). This setup is used as an example throughout this chapter in order to explain the presented concepts.

12.1.1 Problem Description

During the execution of a data flow system, each node calculates and outputs one or more data items. A data item can be requested for every frame, a coordinate consisting of a time value and a track which identifies a parallel world. The space of all frames is a multiverse and can be represented by a two-dimensional grid (Figure 12.2a), as we will do in the figures of this chapter.
Figure 12.1: Investigation of material transport through water in (c) a levee breach scenario. (b) World Lines navigate the underlying (a) data flow nodes across time and multiple simulation runs (visualized as tracks) to calculate (d) pathlines that describe the transport phenomena in alternative scenarios. Figures in this chapter, unless otherwise noted, created by Jürgen Waser, who contributed the user interface for this work [85].

Figure 12.2: Definitions. A frame is defined by a time value and a track. (a) The multiverse is the space of all frames. (b) The capabilities of a node is a subset of the multiverse and consists of all frames the node can produce data for. (c) The jobs of a node are a subset of the capabilities the node has to generate data for.
The system lacks a visualization to report what the view nodes can generate. (Q2) The user needs control mechanisms to increase the number of data items a node can produce, e.g., to generate an additional data item in the levees node that models a levee breach. (Q3) Intuitive navigation concepts are missing to let users select what they want to see. The data flow network corresponds to the setup in Figure 12.1 which has the task to generate pathlines.

Depending on its type, settings and position within the network, a node is capable of producing data for a particular subset of the multiverse. We refer to this subset as the capabilities of a node (Figure 12.2b).

For example, the terrain node in Figure 12.3 generates only one data item, which represents the static, geometric boundary conditions of the simulation. The time-dependent simulation and pathlines nodes are capable of generating data for several frames. The set of frames these nodes can produce depends not only on the node itself, but also on the frames for which input data is present. Thus, we require a data flow algorithm to compute what the nodes can produce, and a visualization to present these capabilities to the user (Q1).

The investigation of alternative scenarios requires the ability to manually extend the capabilities of a node (Figure 12.3, Q2). By default, the levees node in Figure 12.3 calculates one geometric data item that models the intact levees of the city. To manually enforce a levee breach, the node needs to be able to calculate an additional data item. To be able to explore alternatives as needed, a generic system for capabilities extension is required.
With the evaluation and visualization of capabilities at hand, we further require interactive navigation concepts that let users select what they want to see (Figure 12.3, Q3). For this purpose, a mechanism is needed that automatically assigns jobs to the nodes. A job is a subset of the capabilities which a node has to compute data for (Figure 12.2c). In existing systems, the user was responsible for assigning jobs to each of the involved nodes manually.

12.1.2 Proposed Solution

As described in the last subsection, every leaf node is capable of producing a set of frames. For each of these frames, the dependency tree can be evaluated, which can be seen as a data flow itself. A direct implementation of this approach enables the use of the simple well known data flow algorithms to implement arbitrarily complex multiverse nodes. However, the resulting data flows would become very complex because each node in the original node would have to be duplicated for each frame the original node is capable of producing data for. Also, changing parameters such as the integration time of a pathline node results in a data flow change. Even though such a direct implementation is impractical, it is useful to keep it in mind in the course of this chapter.

The solution is to use the above idea of an embedded data flow without making it explicit for the user of the system. Two additional data flow passes are used to evaluate the node capabilities and to enable job assignment. The user controls the system via World Lines, which operate on top of the data flow. As illustrated in Figure 12.4, the multiverse can be mapped onto the visual entities representing frames that World Lines provide. To show the user what can be calculated and visualized, the framework first determines the capabilities of all the view nodes. In this computation, the data flow is traversed in topological order from the source nodes to the view nodes. This downstreaming of capabilities is illustrated through grey arrows on the data flow connections in Figure 12.4. After the capabilities of all view nodes in the network are calculated, their union is highlighted in World Lines.

To allow the user to study alternative scenarios, the node capabilities can be extended by providing new settings. The new settings can be introduced at the level of an entire track or individual frames. For example, in Figure 12.4 both the levees and the terrain are defined by a single settings object valid across the whole multiverse. Should the user wish to experiment by breaching a levee, this can be done by branching a new track off the parent track (Figure 12.5, S2). The user specifies a new set of settings describing the breach, valid across the newly created track. The levee node can then produce two distinct data items, one where the levees are intact, and another which
models a breach. As can be seen in Figure 12.5, the extension of the capabilities triggers another downstreaming pass, causing the World Lines to update and show the newly available capabilities.

Once the capabilities are shown, the user can perform interactive job assignment via multiple cursors (Figure 12.6, S3). For each view node, the World Lines view provides an individual cursor. The position of the cursor determines which frame or frames a view node is to visualize. These frames are the view nodes’ jobs. The jobs of all other nodes are determined by a data flow upstream process which guarantees that each node in the network generates the required information. In this process, the data flow is traversed in topological order from the view nodes to the source nodes. This upstreaming of jobs is illustrated through yellow arrows on the data flow connections in Figure 12.6. Once the upstreaming is complete, a standard data flow pass can be done to calculate all the necessary data items, and display the results of the calculation.

To demonstrate our solution, we consistently use the levee breach scenario throughout the chapter. This example is later joined by two more case studies – a more complex real-world simulation scenario and a study of vortices in pre-simulated dam-break data loaded from files.
Figure 12.5: Proposed solution, part two. (S2) The capabilities of a node can be extended through the specification of track settings. Interactively, this is accomplished via branching in World Lines. Here, the user branches with a parameter of the levees node to model a breach. The extended capabilities are downstreamed the data flow to update the World Lines view with the newly created track.

12.2 OVERVIEW OF THE NAVIGATION CYCLE

In this section, we provide an overview of the most important components in the multiverse navigation cycle. The chart in Figure 12.7 depicts an operational step-by-step work flow of the data flow tasks (green boxes) and World Lines tasks (blue boxes). The terms used in this chapter are summarized in Table 12.1.

**Downstreaming capabilities** The multiverse navigation always begins with this data flow process which evaluates the set of frames of all view nodes which can be computed. The capabilities of a node are calculated as an intersection of the capabilities of its input nodes. This intersection is further modified by nodes which perform temporal processing. For example, a simulation node appends frames to show how far the simulation can proceed.

**Capabilities visualization** The capabilities of all view nodes are transmitted to the World Lines view. World Lines employ a framewise representation to visualize the received capabilities.

**Scope-based settings** In this optional step, users configure the settings of nodes. If the user creates a new branch or a setting influences
Figure 12.6: Proposed solution, part three. (S3) The user tells the view nodes what to calculate through interaction with multiple cursors. We utilize a data flow upstream process (yellow arrows on connections) which assigns jobs to other nodes automatically.

Table 12.1: Terms used in this chapter.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>data flow module with input and output ports</td>
</tr>
<tr>
<td>input nodes</td>
<td>all nodes directly connected to the input ports</td>
</tr>
<tr>
<td>output nodes</td>
<td>all nodes directly connected to the output ports</td>
</tr>
<tr>
<td>settings object</td>
<td>collection of node parameters</td>
</tr>
<tr>
<td>track</td>
<td>user interface element representing an alternative scenario</td>
</tr>
<tr>
<td>frame</td>
<td>coordinate consisting of a time step and a track, also the user interface representation</td>
</tr>
<tr>
<td>scope</td>
<td>data structure representing either a frame (per-frame scope) or a track (per-track scope)</td>
</tr>
<tr>
<td>scope set</td>
<td>collection of scopes</td>
</tr>
<tr>
<td>capabilities</td>
<td>scope set representing data a node is able to produce</td>
</tr>
<tr>
<td>jobs</td>
<td>scope set representing data a node is required to produce</td>
</tr>
<tr>
<td>cache</td>
<td>scope set representing data a node has produced</td>
</tr>
<tr>
<td>settings set</td>
<td>scope set representing all settings objects of a node</td>
</tr>
</tbody>
</table>

A node’s capabilities, the downstreaming of capabilities is triggered again.
**Job assignment** Each view node is associated with a cursor in the World Lines view. If a cursor moves, one or more frames from the related view node have to be computed.

**Upstreaming jobs** The user request is communicated from the view node upstream to all connected nodes in the data flow.

**Downstreaming data** The final step concerns the standard data flow execution where each node performs its allocated jobs to compute all requested data. The navigation cycle starts anew if the user assigns additional jobs using cursors or modifies node settings.

### 12.3 Scopes

Before explaining the data flow algorithms in more detail, it is necessary to introduce the concepts of *scopes* and *scope sets*. As mentioned before, data flow connections lack any sort of temporal or parameter-related information. To remedy this, almost all the objects in our system have additional information attached to them in the form of scope-based structures. Since the design of these structures has a great effect on the performance and abilities of our algorithms, they will be explained here in detail.

#### 12.3.1 Scope and Scope Set

Almost all the objects in our system can be said to occupy some subset of the multiverse or originate in it. Were it not for the fact that some
changes can be applied to entire tracks only, we could use frames as the building blocks of our system. Instead, we define a scope as the basic unit in the system, and we define it dually: A scope can be either per-frame or per-track. Per-frame scopes are defined by a time value and a track and functionally identical to a frame. On the other hand, a per-track scope encompasses all the frames belonging to the same track. This definition allows us to treat all sorts of changes the user can introduce to the system the same way in the algorithms that follow. Examples of scopes are shown in Figure 12.8a.

Scope sets are defined as a collection of unique scopes. They can contain both per-track and per-frame scopes, as can be seen in Figure 12.8c which illustrates how we represent scope sets for the explanation of our algorithms. Scope sets are used to represent how object values are distributed across the multiverse. The values are not stored in scope sets – they only serve as a repository of scopes which can be used as a key to retrieve the actual object values from hash tables (Figure 12.8b). For example, the scopes can describe capabilities, but also data items which a node has computed.
The implementation of scope sets is important to the performance of our algorithms, making it necessary to stop considering a scope set as a black box. To show the example of the internal organization of a set, we will use the example scope from Figure 12.8c. The insides of this particular set is shown in Figure 12.8d. The time values of all the per-frame scopes belonging to the same track are stored in a single ordered set. This set, along with a Boolean for the per-track scope, is stored in a singly associative container. The container associates track identifiers with the track-specific data. To see why sets are organized as they are, it is time to define the basic operation that can be performed on scope sets.

12.3.2 Hierarchical Lookup

The reason why it is possible to have multiple data or setting values associated with a single node stems from the hierarchical nature of World Lines. To explore different simulation or visualization parameters, the user can create new tracks by branching from existing ones. However, it would be impractical if unaffected nodes would perform unnecessary calculations recreating existing data. For example in the flooding scenario, it is not necessary to recalculate the terrain for every parameter modification. Instead, the terrain should be fetched from the scope of the root track, no matter what scope is currently being handled by the data flow. This is why every node has a hierarchy of different values it can produce represented by a scope set.

When producing new data items from input nodes whose scopes are not identical to the processing node, a hierarchical lookup must be performed to retrieve a matching scope. Figure 12.9 provides an example for such a lookup chain. At the beginning, the lookup algorithm searches for the given scope directly (1). If nothing is found, the lookup continues in the scope of the track associated with the given scope (2). From there, the algorithm searches in the parent track, starting at the scope which has the same time value as the given one (3).
Algorithm 1 ScopeSet::hierarchicalLookup

input: Scope scope
output: Scope resultScope

1: track ← scopeSet.getTrack(scope.trackId)
2: while scopeSet.exists(track) do
   3: if scope.type=per-frame and track.hasTimeValue(scope.timeValue) then
      4: resultScope.trackId ← scope.trackId
      5: resultScope.type ← per-frame
      6: resultScope.timeValue ← scope.timeValue
      7: return resultScope
   8: else if track.containsPerTrackScope then
      9: resultScope.trackId ← scope.trackId
     10: resultScope.type ← per-track
     11: return resultScope
   12: end if
   13: track ← track.parent
  14: end while
15: throw("Scope not found")

If this scope is not present either, the scope of the parent track is inspected (4). This recursive search goes up the track hierarchy until a matching scope is found (5,6). The implementation of the lookup procedure is presented in Algorithm 1. The runtime of the lookup is of order $O(n \cdot \log m)$ where $n$ is the number of tracks and $m$ the number of time values associated with a track. It should be noted that this is a worst case analysis. In common scenarios, the factor $n$ will behave more like $\log n$ because the World Lines tree usually has logarithmic depth.

Our choice for the lookup chain is motivated by the need for flexible parameter experimentation. In Section 12.6 we will demonstrate how to exploit this feature for the animation of parameters which are valid across many tracks. In the following sections, we explain how to use scope sets to evaluate capabilities, to assign jobs and to cache computed data.

12.4 Capabilities

The capabilities of a node consist of all scopes for which the node is able to produce data. While the capabilities of all nodes have to be calculated, the user is interested in the capabilities of the view nodes only – the end results that can be produced.
12.4.1 Downstreaming Capabilities

At the beginning of the multiverse navigation cycle, nothing is known about the results the nodes may produce. Source nodes can determine their capabilities based on the input data or settings used, but for all other nodes, a downstream process must be performed. To calculate a node’s capabilities, the capabilities of all of the input nodes need to be already calculated. Once prepared, they are combined with the following condition: Any scope that cannot be matched within the other input nodes’ capabilities has to be discarded. An unmatched scope signifies that not all input nodes can provide data necessary to calculate the output at this scope and thus cannot be part of the node capability.

In mathematical terms, the operation can be expressed as the scope-set intersection

$$\text{cap}(\text{node}) = \bigcap_{n \in \text{inputs}(\text{node})} \text{cap}(n).$$

(24)

The intersection operation is performed pairwise, being both commutative and associative. Figure 12.10 illustrates an example to explain the intersection of two scope sets denoted as cap(a) and cap(b). The related track hierarchy comprises three tracks, both track 1 and track 2 are a direct branch from track 0. As shown in Figure 12.10, it is implemented as a two-way lookup. We attempt to match scopes of cap(a) in cap(b), and then scopes of cap(b) in cap(a). Every scope matched by a hierarchical lookup (not the found matching scope) is added to the resulting scope set (see also Algorithm 2, lines 4-6). The only exception to this rule are the per-track scopes. Should the lookup of a per-track scope fail, the scope is decomposed into per-frame scopes which are matched individually. Any of the matched per-frame scopes are added to the result as well. An example of this can be seen in Figure 12.10, where the lookups of the two track-global scopes in cap(a) fail in cap(b). However, the decomposition of the per-track scopes results in three successful lookups each, which are added to the result of the intersection. Due to the unlimited number
12.4 Capabilities

Algorithm 2 ScopeSet Intersection $\cap$

\textbf{input:} ScopeSet $\text{set}[0]$, ScopeSet $\text{set}[1]$

\textbf{output:} ScopeSet result

1: result $\leftarrow$ emptyScopeSet
2: for $i = 0$ to 1 do
3:     other $\leftarrow (i+1) \mod 2$
4:     for each scope $\in \text{set}[i]$ do
5:         if $\text{set}[other].\text{hierarchicalLookup(scope)}$ then
6:             result.insert(scope)
7:         end if
8:     if scope.type = per-track then
9:         track $\leftarrow \text{set}[i].\text{getTrack(scope.trackId)}$
10:        while $\text{set}[other].\text{exists(track.id)}$ do
11:           otherTrack $\leftarrow \text{set}[other].\text{getTrack(track.id)}$
12:              for each otherScope $\in$ otherTrack do
13:                 if otherScope.type = per-frame then
14:                     scopeToInsert $\leftarrow$ otherScope
15:                     scopeToInsert.trackId $\leftarrow$ scope.trackId
16:                     result.insertScope(scopeToInsert)
17:                 end if
18:             end for
19:         track $\leftarrow$ track.parent
20:     end if
21: end for
22: end for
23: return result

of possible per-frame scopes, this view of the decomposition cannot be effectively implemented in practice. Our optimization is shown in Algorithm 2, line 8-21.

The runtime of the complete intersection is in $O(2 \cdot n^2 \cdot m \cdot \log m + 2 \cdot n^2 \cdot m) = O(n^2 \cdot m \cdot \log m)$ where $n$ is the number of tracks and $m$ the number of time values in the larger of the two input scope sets. The first term accounts for the hierarchical lookups and the second term covers the special case.

12.4.2 Visualization of Capabilities

Upon completion of the capabilities downstreaming process, we need to report the results to the user. World Lines are ideally suited to visualize the per-frame scopes present inside the results. Figure 12.11a and Figure 12.11b show screenshots of World Lines displaying the capabilities of view nodes. For further details on how capabilities are visualized, we refer to the work of Waser [85].
Figure 12.11: Frame-wise representation of tracks to visualize capabilities. (b) Capabilities of node ‘View B’, which is connected to the pathlines node, contain less frames than the capabilities of (a) ‘View A’. (c) Visualization of adaptive time steps loaded from external files.

12.5 JOBS

The jobs of a node are defined as all the scopes for which the node has to compute data. The user determines these for the view nodes using cursors as seen in Figure 12.12. In the work of Waser [85], this process is explained in more detail. Once the jobs of all view nodes are set, the necessary jobs of all other nodes are automatically computed in the correct order.

12.5.1 Upstreaming Jobs

After the jobs have been assigned to the view nodes, the jobs of all other nodes are automatically evaluated via an upstream process (Figure 12.6). For each of these nodes, the result depends on the jobs of all output nodes. In principle, a node has to compute data for any scope that is requested by any of its output nodes. In mathematical terms, this operation can be expressed as the scope set union

$$\text{jobs}(\text{node}) = \bigcup_{n \in \text{outputs}(\text{node})} \text{jobs}(n).$$

As its name implies, the scope set union simply merges all the scopes of its scope set operands together, avoiding duplicate entries (see also Algorithm 3). The union produces a set of jobs we call output jobs, but which are not necessarily equivalent to the set of jobs
the node must perform. Nodes with global scopes such as the terrain node only need to produce one data item regardless of which scopes this data item is used to calculate. The real scope set of jobs the node must perform is calculated by performing a hierarchical lookup of every output job in the node’s capabilities, and inserting the matching capabilities scope into this scope set. In this way, the system is guaranteed to produce the minimum of calculations necessary, leaving no room for redundancy. The runtime of the scope set union amounts to

\begin{algorithm}
\caption{ScopeSet Union $\cup$}
\begin{algorithmic}[1]
\STATE input: ScopeSet set[0], ScopeSet set[1]
\STATE output: ScopeSet result
\STATE result $\leftarrow$ emptyScopeSet
\FOR {each scope $\in$ set[0], set[1]}
\STATE result.insert(scope)
\ENDFOR
\RETURN result
\end{algorithmic}
\end{algorithm}

$O(n \cdot m)$, where $n$ denotes the number of tracks and $m$ the number of time values in the largest scope set. The ordered internal nature of the scope sets allows this operation to be performed quickly.

12.5.2 Downstreaming Data

As soon as we know the jobs of all nodes, the requested data can finally be produced. In this step, the data flow is traversed in a downstream process to execute all nodes. During execution, a node loops over its assigned jobs to produce data for each scope inside. If the node has to fetch data from an input node, another hierarchical lookup

Figure 12.12: Navigation using two cursors (a)(b). The World Line which belongs to the active cursor (b) is colored in blue. The cursors can be configured and customized.
is performed to automatically retrieve the correct data items. This way, data at per-track scope can be fetched from input nodes even if the node is processing a per-frame scope. This is a common use case in the flooding scenario, where the simulation node has to fetch the static terrain and the track-specific levees no matter what simulation step is currently being handled (see also Figure 12.5).

We employ another scope set, the cache (Table 12.1), to remember all scopes for which the node has generated data. The actual computed data is cached inside a hashtable. This approach allows for arbitrary caching strategies which are useful for the task at hand. In our current implementation, all data is kept due to frequent navigation in time and across tracks. In order to prevent excessive memory use, the system is able to swap out computed data to the hard disk. When a node fetches data from its inputs, the data is pinned. Pinned data is loaded from the disk if necessary, and cannot be swapped out as long as it remains pinned.

12.6 Scope-based settings

To study alternative scenarios with World Lines, we require the ability to manually extend the capabilities of a node. In the use case of the levee scenario, we want to model different types of levee breaches. Each breach is modeled by branching off a new track. Consequently, additional settings objects are constructed and assigned to the related levees node. The settings objects are internally treated much as another input to the node. Like the data items produced by nodes, each of these receives a scope, and is stored in a hash table with the scope as a key. When a data value is calculated, the appropriate settings object is also retrieved via a hierarchical lookup.

The present settings affect the capabilities calculation. The scope set of the settings, the settings set, can cause additional capabilities scopes to be generated. Including the settings set into the capabilities calculation is simple:

\[
\text{cap}(\text{node}) = \left( \bigcap_{n \in \text{inputs}(\text{node})} \text{cap}(n) \right) \cap s(\text{node}). \tag{26}
\]

When a new track is created using the World Lines, the settings set is updated, causing the desired capabilities extension.

Scope-based settings can be exploited to achieve interesting effects. With the proposed solution, settings can not only be assigned to whole tracks but to individual frames too. For example, in Figure 12.13, we experiment with time-varying camera positions. Two frames of the root track define explicit camera locations along the river. As a consequence of the lookup mechanism, these locations transfer into child tracks unless explicitly overridden. When navigating to the frame at
The nodes we have used to demonstrate the algorithms so far are simple nodes (Figure 12.14a). These nodes process values from only one scope at a time, and both the nodes and the node creators are oblivious to the actual scopes involved. However, with the presented framework, any kind of time-varying or track-dependent node can be implemented. As an example, we consider a time interpolation node which is capable of producing new data for additional frames through temporal interpolation of its input data. The evaluation of capabilities according to Equation 24 is no longer sufficient.

The left scope set of Figure 12.14b illustrates the capabilities intersection the algorithms described so far would produce. The time interpolation node is capable of producing more data. The node notes this
by extending the capabilities set to the one shown on the right-hand side of the node. When one of the extended capabilities is selected as an output job, the node has to determine the input jobs necessary for interpolation manually rather than rely on lookups. The time interpolation node is a typical example of a scope set modifier which alters its capabilities and jobs.

Scope set modifiers get a chance to alter their output capabilities and input jobs after they have been calculated using the normal algorithms. Although any type of scope set modifier can be implemented in our framework, we have identified certain categories of these that we will describe in more detail. All nodes in a category share a certain type of behavior. This allows us to provide a base implementation which, much like a simple node, minimizes the quantity of scope manipulation that the user must perform. An explanation of how various categories work can be seen in Figure 12.14. Some of these also rely on special types of cursors to ease the manipulation of navigation-related node parameters, such as integration time for integration-based nodes. For details, we refer the reader to the work of Waser [85].

### 12.7.1 Numerical Integration

Nodes performing numerical integration require a range of input data to generate their output. The aforementioned pathlines module is a typical example for this node category. In Figure 12.14c, the integration range is set to one time step in both forward and backward direction. Thus, the first scope of the root track and the last scopes of the two tracks are removed from the input capabilities to form the actual, modified capabilities. If jobs are assigned, the node inserts further scopes to request input scopes that cover the complete integration range.

### 12.7.2 Simulation

Nodes of the simulation category can produce an unlimited amount of per-frame data. The data is produced from initial and boundary conditions supplied through input nodes. In Figure 12.6, the simulation node receives the boundary geometries from the terrain and the levees nodes. These dependencies often remain static within a track, i.e., a specific simulation run. The simulation node modifies the capabilities, outputting only per-frame scopes based on track start times and track durations. If jobs are assigned to the node, causality needs to be accounted for. All the frames causally preceding the desired frame have to be simulated before it, and are thus inserted into both the input and the output jobs. The input jobs in Figure 12.14d are
Figure 12.14: Scope set modifiers are nodes which manipulate their computed capabilities (grey) and assigned jobs (yellow). A distinction between input (left) and output (right) capabilities and jobs has to be made. The top, left image depicts the World Lines structure involved. (a) In the standard case, input and output capabilities and jobs are equal. (b-e) Scope set modifiers insert or remove scopes to have access to all required data.

represented by yellow borders since they are not a subset of the input capabilities, and the new output jobs are represented by blue dots.

### 12.7.3 Aggregation

Aggregation nodes are a special type of nodes which combine the output of multiple frames into one result. The set of frames chosen by the user is called the brush of the aggregation nodes. Aggregation nodes can receive a brush in order to compute an analytical or visual representative of all data items marked by the brush.

Internally, the brush is converted into a scope set which replaces the original input jobs (see left of Figure 12.14e). This way, the node has access to all input-data items required to compute the aggregation result.
12.8 Case Study

To highlight the fact that our system is modular and useful for various problem types, we prepared two quite different case studies. Both of these use the data flow algorithms to produce results that would have been significantly harder to obtain without the support our system provides.

12.8.1 Vortex Analysis of Pre-simulated Data

In this case study, we are given the data produced by a fluid simulation of a breaking dam consisting of 88 adaptive timesteps. The goal of the study is to analyze the vortices present in the fluid. Several visualization tools are present at our disposal: slices, pathlines and isosurfaces of $\lambda_2$ (a measure for vortex strength [36]). While we can combine these to create single images, with World Lines we can use the branching functionality to record the user’s work flow during visual analysis. In this respect, the contribution is similar to the VisTrails approach [73] which utilizes a graph-based representation of the analysis steps. The basic purpose of the VisTrails history tree is to return the system state to previous states which can be regarded as a sophisticated version of undo/redo functionality. There are no flexible, time-dependent navigation tools, nor is it possible to synchronously compare the results obtained at different steps of the recorded work flow. These limitations are not given if using World Lines for the analysis of time-dependent data.

To begin with, we create two additional tracks in the World Lines view that comprise different isovalue settings. The two isovalues are chosen so that they highlight areas of different vortex strength. Figure 12.15c and Figure 12.15d visualize the isosurfaces (red) related to the two $\lambda_2$ values, giving a good idea of the vortex structures contained. In addition to the isosurfaces, we add the pathlines to help understand the flow surrounding the vortices better. The time span they involve is controlled and shown by the integration cursor. To provide a visual context, we render the free surface of the simulation data in a transparent blue color. The free surface is computed by the isosurface algorithm presented in Chapter 4 where the scalar field used equals to the sum of the particle weights (Equation 5). Areas where the fluid reaches a solid wall are ignored. For an overview of the area, we show a slice image of $\lambda_2$ in Figure 12.15b. This slice is also embedded in Figure 12.15a.

When navigating in time, we find that the vortex structures move significantly in vertical direction. To catch the features within the slice image, the vertical offset of the slicing plane needs to be adjusted. To remember the optimal positioning for each time value, we assign scope-based settings to the related frames of the root track (see key-
Figure 12.15: Vortex analysis in a pre-simulated scenario of a breaking dam (screenshot). (a) The overview renders the free surface of the water while embedding (b) a slice of $\lambda^2$. Green colors highlight areas of high vortex strength, red colors depict low values. (c,d) Two views compare isosurfaces for different $\lambda^2$ values which are calculated for track 1 and track 2, respectively. Since vortices move vertically over time, we adjust the slice position through key-frames.
frames in World Lines of Figure 12.15). Hereby, the slicing plane automatically assumes the optimal offset when navigating in time and across tracks.

The ability to record the exploration path using scope-based settings provides the basis for a thorough, comparative analysis of different parameters. In combination with the navigation through multiple cursors, we were able to quickly identify relevant vortices in the data. Moreover, results were easily reported to other users by navigating the recorded work flow. By examining the work flow, the user can see where and why the decision to use a different visualization parameter was made.

12.8.2 Movement of Swept-Away Vehicles

In the second case study, we decided to examine the problem of vehicle-caused danger during a flood. Michael Greiner from GeoConsult provided us with the data necessary to create a scenario based
on a real town’s flood protection system, seen in Figure 12.16. A river prone to flooding flows through the town, and mobile walls are used to stop the water from spilling into the town.

However, these mobile walls are not perfect defenses, and should they fail or break, a torrent of water may spill into the town. The danger becomes much greater if heavy mobile objects such as cars are not removed, as the water may carry them at high speeds, causing much damage. The goal of the study is to try and determine which cars would be the most dangerous if not removed.

Given a fixed distribution of cars, we experiment with an unknown breach location to see the dangers posed by car movement. From the data flow point of view, the data is provided via a simulation node which produces data on demand. To visualize how the cars move, we employ a node which displays the movement paths of the cars as a line, colored according to distance moved. This node is an integration node, requiring the simulation node to produce both future and past frames of the visualized frame. It then connects the individual car positions computed by the simulation using a B-spline. In that sense, the node does not perform any numerical integration, but from a data flow perspective, it acts like an integration node. We employ three views in parallel, showing three breach locations, which can be seen in Figure 12.17.

As the user navigates across different time values, the same view produces all three images, causing three integrations to happen. In turn, these cause the simulation node to produce any data that is missing. All of these actions are invisible to the user, who moves a grouped cursor and sees the images update.

12.9 Evaluation

The evaluation of the algorithms described in this chapter proved to be troublesome because of their nature – if they work as they are intended to, the user is not aware of their presence. The only manifestation of their use is the ease with which additional functionality is integrated. Therefore, we evaluate whether the various features our algorithms enable would be useful to domain experts, whether they would consider using them, and whether the more advanced interface elements are intuitive.

Our first interviewed expert is a renowned hydrologist performing research related to flood simulation. The discussion revealed that to him, modularity can be seen as the most important feature. Many institutes studying flood simulations have their own simulation engines, and would prefer to use a system capable of integrating them easily. Our scope set modifier implementation for simulations allows this to be done with a minimum amount of effort. Another feature that caught his eye are the pathlines, which he would like to use to
visualize and analyze transport of material such as driftwood or leaking oil.

We also engaged a second expert, another hydrologist, specializing in flood modelling. The second study’s multiple and integration cursors struck him as useful for the purpose of examining and comparing the data. The presentation aspect of the system shown in the first study also proved interesting, as a way of animating and showing the evolution of multiple attributes of interest, such as soil moisture. Both the pathlines and the car lines bear resemblance to his current field of study: the transportation of sediments along riverbeds. An echo of the first expert’s comments could be heard when asked if he would consider using the system: yes, but under the condition that a specific simulation engine could be used.

Aside from the two in-depth conversations summarized here, the system and coincidentally the algorithms have been presented to a number of experts and potential users. They all liked both the navigation enabled by World Lines and the advanced functionality brought by the scope set modifier nodes, but also wanted to be sure that their simulations and use cases could be adapted to be used alongside these. Given that it is the algorithms of this chapter that make this possible by allowing the time- and multiverse-dependent nodes to be
interchangeable and modular, we consider it a proof for the relevance of our system.

12.10 PERFORMANCE

In the following subsections, we try to quantify the overhead of our approach as well as the memory it saves.

12.10.1 System Overhead

There are two sources of overhead within the presented system – the capabilities and the jobs calculation. As Table 12.2 shows, the capabilities update is the more expensive of the two. However, capabilities change rarely and an interactive response is not necessary for a streamlined user experience. Given that the data flow of the swept-away vehicles simulation is very complex (55 nodes, 10 tracks, 600 timesteps), the measured time needed is a very positive result.

The jobs calculation is performed with almost every action the user makes. It is thus crucial that its cost is insignificant compared to the time necessary to execute a job. The timings in Table 12.2 confirm that this is the case. The measurements shown there are performed when the user moves the cursor ahead one frame, with the previous frame already computed. After a cold start of the system, the jobs calculation can take significantly longer (up to 3 ms in case of the Moving Vehicles case). This is because every node needs jobs to be assigned to it, and in case of a simulation node, the number of jobs can be quite large. However, even in this case, the jobs calculation is a minimal overhead, comparing favorably to the execution time. With more precise simulations using a better resolution and more time, the system overhead becomes even less significant.

12.10.2 Memory Savings

One of the biggest advantages of the system is that additional computations can be avoided thanks to our hierarchical approach. As we are
not aware of another system with similar capabilities, we compare it to a naive system computing all the data for each requested frame.

Table 12.3 lists the memory size of the individual data objects for a single frame. Assuming the three tracks from the vortex analysis case, the naive system would require three times the total amount, or 247.86MB. Thanks to the hierarchy, only the isosurface has to be recomputed for the different tracks. Therefore, with the presented approach, only 85.22MB are used, which is a reduction to one third of the memory used by the naive system.

In the Movement-of-Swept-Away-Vehicles case study, the memory savings are marginal as the memory consumption is dominated by the simulation data. This data differs from track to track, and only small amounts of memory can be saved using our approach. This example highlights that both system and memory overhead are mainly dependent on the use case and the user interaction. Our system is designed to improve performance wherever it is possible.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Input Data</th>
<th>$\lambda_2$</th>
<th>Free Surface</th>
<th>Path Lines</th>
<th>Isosurface</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (MB)</td>
<td>76</td>
<td>2.5</td>
<td>2.8</td>
<td>0.02</td>
<td>1.3</td>
<td>82.62</td>
</tr>
</tbody>
</table>

Table 12.3: Memory usage for the vortex analysis for a single frame
In this part, we presented a data flow system for a multiverse setting. The data flow pipeline is known as the most generic architecture for modular visualization environments. Indeed, many modern systems are built upon the data flow approach [82, 80, 19]. Even though all of these systems have some capabilities of working with time-dependent data, none provide a generic formalism for the automatic management of time-varying data. A strategy to handle alternative scenarios as induced by multiple simulation runs is also largely missing. As a result, either programmers or users have to make sure that nodes can access data from all time steps required to compute the requested results. In this part, we have shown how to extend the data flow model with algorithms which enable elaborate data processing across many time steps and alternative outcomes. By combining the new data flow model with the interactive exploration power of World Lines, the complexity of temporal and multiversal processing is hidden from the user. Intuitive navigation concepts let users easily choose what they want to see, and internally, the work load is distributed to the nodes automatically.

All of the discussed features have been implemented in Visdom [2]. The framework supports simulation nodes, different kinds of temporal integration nodes such as pathlines or streak surfaces and many forms of data aggregation. Extending Visdom with time-dependent nodes requires little effort from the programmer, because the management code resides in the core of the system. With the presented concepts, World Lines can now be used to explore parameter spaces of any node, not only those of simulation nodes. The ability to cache and visualize computed results for different parameters in form of interactive tracks and frames enables a flexible comparative analysis of parameter spaces.

Even though the current implementation is single threaded, the presented concepts do not prohibit parallelization. To the contrary, the explicit computation of jobs and their dependencies allows for parallelization based on jobs and not just on nodes. This can have a huge impact on computation time as nodes can already start computing results when their input nodes are still running. In a decision-support system, this gives the user the ability to analyze simulation data while the simulation itself is still running and producing results.

Another limitation of our current implementation is a lack of a caching strategy. Even though we are able to control the memory usage by swapping out data to the hard drive, we do not swap data
based on frequency of use, nor delete data that could be easily recomputed. Introducing a cache manager would be simple, as the current design supports data eviction, and regenerates data automatically when needed. As future work, we therefore plan to study various caching and eviction strategies.

The scalability of the scope set data structure is another issue. Every track created by the user is explicitly represented, including every single time step. In most cases, the time steps will be the same as for its parent track. If there are only a handful number of tracks, this is not an issue. However, the solution could very well be used for ensemble simulation runs, where the number of tracks becomes large. With large number of tracks, any operation involving capabilities will slow down significantly. Achieving such scalability can be achieved by changing the representation of the scope set datastructure. No change to the data flow algorithms is required, as long the new representation supports the required operations, such as intersection, union and hierarchical look up.

Implementing the visualization methods presented in this system requires a robust system with the necessary capabilities. Our contributions from Part iii aim precisely in this direction, allowing complex components such as integrations, interpolations or even simulations to be introduced in the same way as the simplest filter node. It was not impossible to add these before, but the amount of work necessary to do so has been lowered significantly. Easier experimentation, in conjunction with the presentation abilities, can contribute to the interesting work currently taking place in the area of ensemble simulations and uncertainty visualization. There exist many different application areas where these techniques could be used – traffic simulations, computer fluid dynamics, and climate research being just some examples. We hope that our approach will help the spread and reuse of advanced visualization techniques.
The Lagrangian flow representation is highly useful in many contexts. The representation of a fluid using particles is probably the most immediate application, particularly useful for representing fluids with open boundaries. But also for visualization, the Lagrangian representation can be very beneficial. In some cases, the Lagrangian nature is a core property as it is the case with LCS. In other cases, the computation can be significantly simplified, for example in the case of pathline-based visualization of a Lagrangian flow field.

In this thesis, we have identified three major areas, where working with the Lagrangian flow representation is challenging. In Part i, we dealt with the problem of applying Eulerian visualization methods to Lagrangian simulation data, in particular Smoothed Particle Hydrodynamics (SPH) data. In particular, we presented a method for extracting isosurfaces from SPH data without an intermediate resampling step. A main contribution is the usage of a correction step to ensure that all isosurface vertices lie on the proper isosurface. At the same time, the method manages to perform very well, being several orders of magnitude faster than competing methods.

In Part ii we dealt with LCS, which are Lagrangian topological features in unsteady flow fields. There exist multiple competing ways of defining LCS. We therefore started with an evaluation of all the possible LCS definitions. Because LCS are described as a special type of material surface/line, the evaluation was based on the flux across the LCS. Also included in the evaluation is our new LCS definition, the C-ridge. It is based directly on the Cauchy-Green tensor and does not require second derivatives. Therefore, it is numerically more stable to compute. Even though it is difficult to make a recommendation based on the evaluation, the C-ridge performed very consistently. We were also able to demonstrate its computational advantages in a 3D comparison with height ridges. We continued by presenting our method for extracting LCS (based on the C-ridge) from gridded 3D data. It is primarily based on the Marching Ridges algorithm, but contains a number of important modifications for performance and ease of use. In particular, we listed several filtering techniques for reducing noise, false positives as well as simplifying rendering by making the geometry orientable. The use of Fast Fourier Transform (FFT) speeding up the computation of Gaussian gradients was also discussed. Finally, we applied LCS to analyze the air flow around a revolving door which is combined with an air curtain. Using the LCS-based visualizations, we were able to reveal a heat leak which would have been very hard to
spot using more traditional visualizations. Based on these findings, the simulation was rerun with different modifications which minimized the impact of the heat leak.

In Part iii, we presented a data-flow system for dealing with multiverse data, that is, multiple time-dependent scenarios. First, we solved the problem of reporting the space, the user is able to navigate. For this, we introduced a novel data structure, the scope set, which is capable of representing the hierarchical structure of the World Lines very efficiently. This data structure is used as the building block for jobs calculation, which is the process of communicating the request of the user to the nodes. Thanks to the hierarchical representation, no result is computed twice, thus saving both memory and computation time. By generalizing the concept of node settings and by including them into the capability calculation, World Line tracks can be created very easily. For the node developer, the main benefit is that complex time- or scenario-dependent nodes can be implemented with little effort.


