Benjamin Hopfer

3D Flow Field Simplification via Streamline Bundles

3D FLOW FIELD SIMPLIFICATION VIA STREAMLINE BUNDLES

BENJAMIN HOPFER

Master's Thesis

Institute of Computer Graphics and Knowledge Visualization
Graz University of Technology

In cooperation with:
VIRTUAL VEHICLE - Kompetenzzentrum
Das virtuelle Fahrzeug Forschungs-GmbH (ViF)

December 2011

Benjamin Hopfer: 3D *Flow Field Simplification via Streamline Bundles,*
Master's Thesis, © December 2011

SUPERVISOR:
Dipl.-Inform. Dr.-Ing. Sven Havemann

LOCATION:
Graz, Austria

ABSTRACT

Results from computational fluid dynamics (CFD) simulations are generally complex and difficult to understand. This work proposes a new method that computes from a given simulation result, e. g., the underhood flow of air around a car engine, a sparse directed graph network with a few hundred nodes. The goal is a graph that preserves the essential properties of the flow in such way that it is suitable for applications ranging from information visualization to flow simulation. The algorithm finds bundles of similar streamline segments, which are then mapped back to the original dataset in order to produce a complete partition. A flow graph is derived from this partition by integration over the CFD cells. By utilizing a custom-built simulation framework, the proposed method is shown to produce meaningful graphs, which can be used within the mentioned application areas.

ZUSAMMENFASSUNG

Ergebnisse von numerischen Flusssimulationen (computational fluid dynamics, CFD) sind im Allgemeinen komplex und schwierig zu verstehen. Diese Arbeit stellt eine neue Methode vor. Ausgehend vom Ergebnis einer Strömungssimulation, beispielsweise von Luft im Motorraum eines Autos, wird ein dünner, gerichteter Graph mit wenigen hundert Knoten erzeugt. Ziel der Arbeit ist ein Graph, welcher die essenziellen Flusseigenschaften in einer Weise abbildet, die Anwendungen von der Visualisierung bis hin zur Simulation erlaubt. Der Algorithmus findet Bündel ähnlicher Stromliniensegmente und ordnet diese dann wieder dem ursprünglichen Datensatz zu, um eine vollständige Partition zu erzeugen. Aus dieser Partition entsteht durch Integration über die CFD-Zellen ein Flussgraph. Durch Einsatz einer speziell implementierten Simulationsumgebung wird gezeigt, dass die vorgeschlagene Methode sinnvolle Graphen erzeugt, welche in den erwähnten Anwendungsgebieten verwendet werden können.

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

— *Donald E. Knuth*

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Sven Havemann for his guidance while performing this work. He was a true *advisor*; his unique style allowed me to work freely and on my own, while I never felt left alone. Somehow Sven always managed to give the right advice at the right time, providing helpful answers to specific questions as well as general guidelines about scientific work. Our insightful meetings always pointed me in the right direction.

My parents always encouraged me in whatever I was pursuing and supported me in every possible way during my studies. I therefore want to use this opportunity to thank them.

Finally, this work started on the initiative of the Virtual Vehicle Competence Center (ViF) in Graz. Specifically, I want to thank Dr. Daniel Langmayr and Dipl.-Ing. Christian Rauch for their support throughout this thesis. Daniel never turned any of my requests down and generously shared his expertise in fluid dynamics. Christian was also of great help and provided important initial pointers to relevant literature.

The author would like to acknowledge the financial support of the "COMET K2 - Competence Centres for Excellent Technologies Programme" of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG) .

I would furthermore like to express thanks to the supporting industrial and scientific project partners of the ViF, namely AUDI AG, Magna Powertrain Engineering Center Steyr GmbH & Co KG, Graz University of Technology Institute for Internal Combustion Engines and Thermodynamics and to the Graz University of Technology.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

BSP	Binary Space Partitioning
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CVT	Centroidal Voronoi Tessellation
DTI	Diffusion Tensor Imaging
GUI	Graphical User Interface
ID	Identifier
LOD	Level of Detail
MRI	Magnetic Resonance Imaging
NP-hard	Non-deterministic Polynomial-time hard
OS	Operating System
RAM	Random Access Memory
SSE	Sum of Squared Errors
UI	User Interface
ViF	Virtual Vehicle Competence Center

1 INTRODUCTION

Begin at the beginning and go on till you come to the end: then stop.
– The King, Alice in Wonderland [4]

This chapter starts with describing the primary motivation for this work, the construction of car cooling systems. The engineering task itself is described as well as two auxiliary tools and their respective design processes, namely resistance graphs and CFD simulations. Then a hybrid approach using both of these tools and its advantages are discussed. After shortly describing the main dataset of this thesis and other applications of this work, the chapter goes on discussing why results from cluster analysis and image segmentation are hard to apply. Finally, the proposed solution is sketched shortly.

1.1 CONSTRUCTION OF CAR COOLING SYSTEMS

Constructing the air side of the cooling system of a car is a difficult task. The cooler and fan have to be designed, sized, and placed under the hood of the car so that all hot components are kept cool. Insufficient design, poor placement or undersizing of the cooler or the fan can lead to overheating of a part. Oversizing the cooler or the fan adds weight and wastes material and underhood space.

All these factors have to be taken into consideration for different *operating points*, i. e., at different speeds and loads. A car driving slowly upwards a steep hill has a high load and must rely on the fan for air intake. A car driving at high speed on a highway also has a high load, but thoughtful construction takes advantage of the speed for air intake. Typical operating points are:

Old cars like Porsche 911 and Volkswagen Beetle cooled their engines directly with head wind.

STATIONARY CAR (0 km/h): If the car stands still with a running engine, e. g., at traffic lights, only the fan provides cool air to the cooler.

DOMINANT FAN INFLUENCE (30 km/h to 65 km/h): At low speeds like these, the fan still dominates the air intake, but the engine load is higher than for the stationary car. Typically two operation points are chosen in this range.

WIND TUNNEL SPEED (ca. 140 km/h): This is the usual speed for aerodynamic measurements in the wind tunnel. The fan has almost no influence on the air intake anymore as the air is pushed under the hood by the high speed.

MAXIMUM SPEED (e. g., 260 km/h): The maximum speed of the car. The fan is "overblown", i. e., it behaves like a resistance for the air pushed under the hood.

Car designers must ensure proper cooling under all of these conditions. In order to achieve that they must understand the air flow under the hood of the car for all defined operating points. The *underhood air flow* is the entire flow of air between the hood and the base plate from all air inlets to all air outlets. If this flow of air is understood, engineers can base their design decisions, like changing vehicle part placement or adding barriers, on that. This work only covers *static flows*, i. e., the constant air flow observed for stationary boundary conditions. Time-dependent boundary conditions (e. g., an accelerating car) leading to *transient flows* are out of scope.

1.1.1 *Expert generated resistance graphs*

One tool that aids engineers during the design of car cooling systems is *KULI* [15]. The air flow is modeled by a graph with only a few nodes (typically from 10 to 20), where each vertex represents one part of the space under the hood. Several geometric primitives are available for these vertices, e. g., the fan can be modeled as a cylinder, the cooler as a cuboid and less important geometry as simple points. Edges represent possible air flow and the vertices hold resistance data to compute these air flows.

The terms "resistance graph" and "flow graph" are not actually used within KULI.

To generalize this concept, the term *resistance graph* will be used. It denotes the abstract concept of graphs which hold sufficient resistance information to simulate the air flow. The result of simulating a resistance graph is a graph with complete flow information for each vertex and edge, a *flow graph*.

The main advantage of tools based on resistance graphs is the fast simulation of all operating points. KULI also adds the possibility to simulate the air side and the water side of the cooling system simultaneously. It is therefore a tool for planning the whole cooling system.

The big disadvantage of KULI is the need to define the graph representation of the air side. This task demands expertise and speculation and can result in various possible representations. The quality of these representations depends on the experience of the engineer and cannot be directly ascertained or compared. This leads to a repeatability problem: Different engineers produce different resistance graphs, and it is hard to single out the best representation.

Figure 1.1 explains the role of resistant graph tools like KULI within the design process. It is clearly shown that all information about the flow comes ultimately from the expert creating the graph. Notice, that additional operation points do not add considerable processing time, because each simulation only takes a few seconds.

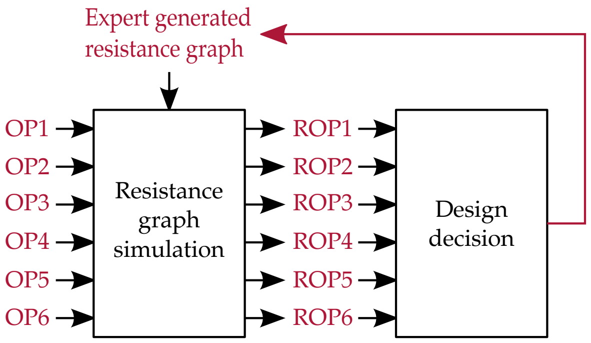

Expert generated resistance graph
OP1
OP2
OP3
OP4
OP5
OP6
Resistance graph simulation
ROP1
ROP2
ROP3
ROP4
ROP5
ROP6
Design decision

Figure 1.1: Design process using resistance graphs. An expert creates the resistance graph. The air flow is simulated using this graph for several operation points (OP1 to OP6), which requires only a few seconds. The resulting flow graphs for the operation points (ROP1 to ROP6) are then used to guide further design decisions. Design decisions usually lead to changes in the resistance graph and the process starts again.

1.1.2 *Computational fluid dynamics*

The world is continuous, but the mind is discrete.
– David Mumford

Another important tool for understanding air flows is a Computational Fluid Dynamics (CFD) simulation. Its main idea is to discretize the continuous problem of flow simulation by dividing the simulation volume into small *cells*, called the *grid* or the *mesh*.[1] For each cell, flow parameters (e. g., velocity, pressure, temperature) are assumed constant. The parameters of each cell are related to the parameters of its neighbor cells by nonlinear equations, namely conservation of mass and conservation of momentum (Navier-Stokes equations). After adding boundary conditions (e. g., outside pressure and car velocity), the flow parameters of each cell are calculated from the system of nonlinear equations by iterative numerical methods, up to a predefined error. This permitted error is the deviation of the calculated boundary condition values to the predefined ones. It can be decreased by increasing the number of iterations of the solver. Figure 1.2 shows the rendered partial result of a CFD simulation of the Toyota dataset [29]. As the term "CFD simulation result" is cumbersome to use, the shorter term *flow field* will be used to talk about data with this structure and not specifically about the result of a CFD simulation.

The Toyota dataset is used throughout this thesis, a detailed explanation can be found in Section 1.2.

The main advantage of CFD simulations is their freely definable quality. The resolution of a simulation can easily be increased by reducing the cell sizes and the precision of the result can easily be increased by lowering the permitted error.

[1] "Mesh" is the term used in Computer Science whereas "grid" or in this case "unstructured grid" is the CFD community term. They are interchangeable with each other and both will be used throughout the thesis.

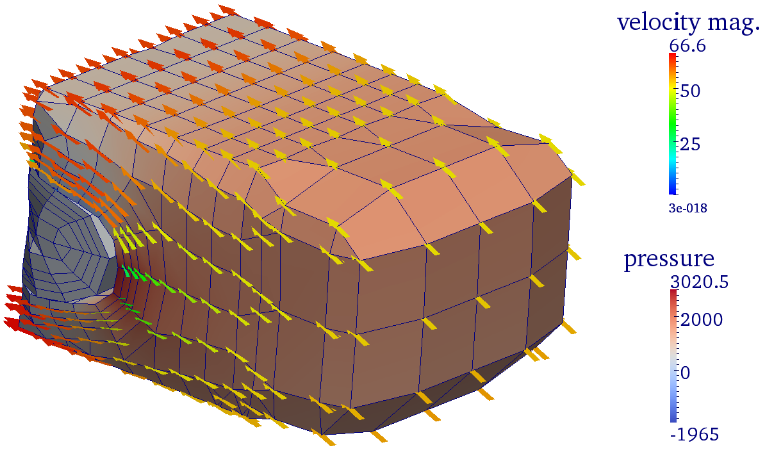


Figure 1.2: 3D-view of a CFD simulation result subset of the Toyota dataset. The subset is located between an upper front air inlet and the cooler. In the Toyota dataset the parameters (pressure, velocity, and temperature) are assigned to the grid points, not the mesh cells. In this visualization blue lines represent the cell borders (i. e., mesh edges), pressure is indicated by surface color, and velocity is indicated by thick arrows which are colored by magnitude.

The second advantage is that the results of a CFD simulation do not rely heavily on user expertise.[2] The simulation result depends only on the 3D-dataset and the boundary conditions.

One disadvantage of CFD simulations is the required computation time for iteratively solving systems of millions of nonlinear equations.[3] Notice that for the car cooling use case, one simulation is required for each operation point, due to the different boundary conditions. A resistance graph based simulation on the other hand is finished in mere seconds on a standard PC and multiple executions for different operation points are therefore not a problem.

Another disadvantage of CFD simulations is the sheer amount of output data produced. The output is a set of flow parameters at every cell. The main challenge after a CFD simulation is therefore understanding the resulting flow field and extracting important information to guide the design of the cooling system.

Methods for visualizing and understanding flow fields are for example false-colored renderings of dataset slices or streamlines. A streamline is a line within a flow field which follows the flow direction at every point, i. e., it is a tangent to the flow direction at every point. Another way to think of streamlines is as the path of a massless particle floating through the field. A streamline is therefore uniquely

[2] Of course user expertise is needed for defining the mesh, the boundary conditions, and other simulation input.

[3] I was told to be careful when stating CFD simulation times, as they can vary greatly. Usually simulation of the Toyota dataset takes from a few hours up to one night on a state-of-the-art workstation.

defined by a *seed point* within a given flow field. The according streamline is created by a *stream tracer* which is tracing the flow field starting at the seed point by employing integration methods. More information on streamline generation can be found in Section 3.4.2.

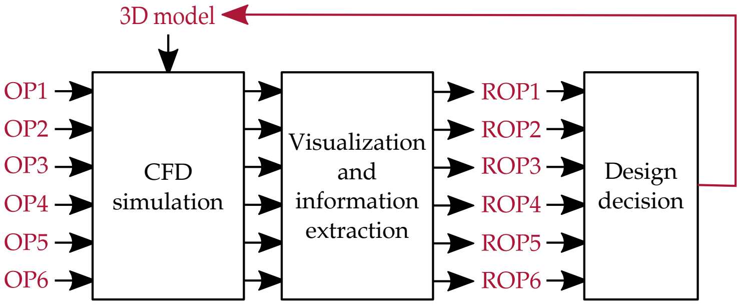


Figure 1.3: Design process using CFD simulations. CFD simulations are applied to the 3D-model for each operation point (OP1 to OP6), which takes a few hours per simulation on a standard computer. The results are then visualized and interpreted leading to one result per operation point (ROP1 to ROP6). These results guide further design decisions, which change the 3D-model and require another simulation iteration.

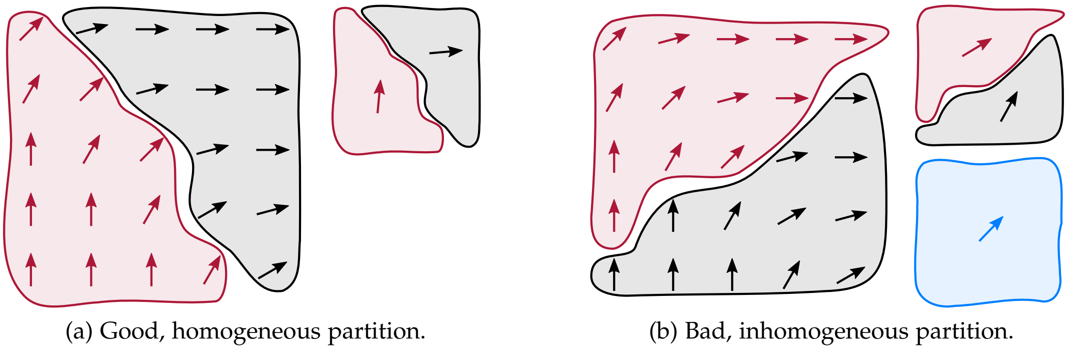
Figure 1.3 shows the design process when using CFD simulations. Since CFD simulation takes significantly longer than resistance graph simulation, adding operation points increases computation time significantly. Remember that the design process is iterative, i. e., design decisions usually lead to a different 3D-model and require new simulations.

1.1.3 *CFD-derived resistance graphs*

The primary goal of this thesis was to bridge the gap between resistance graph representations and the flow fields which result from CFD simulations. Ideally, the result is an algorithm that takes a flow field as input and creates a resistance graph with a few dozen vertices as output.

The global approach for solving this problem is straightforward: In a first step, derive a flow graph from the flow field. From the flow graph, the resistance graph can be generated in the second step. To derive the flow graph the flow field needs to be divided into disjoint regions, i. e., into a partition of the 3D-mesh. The regions of this partition become the vertices of the graph. Two regions sharing a common boundary surface are connected by an edge, because there is possible flow between them. Additional information like the mean velocity and mean pressure of vertices or the exact flow over edges are attached to the graph elements.

Flow graphs can be generated for any partition with connected regions, but resistance graph simulations are based on the assumption that there is exactly one flow condition at each vertex. Therefore *homogeneous* partitions with respect to the pressure and the velocity field are favorable for computing meaningful resistance graphs. The important observation here is that CFD simulation results are highly redundant, i. e., close cells usually have similar flow parameters and can therefore be grouped without losing much information. Stated differently, if the partition consists of homogeneous regions, the lost information due to collapsing cells to one vertex is smaller than for arbitrary regions. Figure 1.4 shows a simple vector field partitioned in both a homogeneous and a inhomogeneous region.


(a) Good, homogeneous partition.

(b) Bad, inhomogeneous partition.

Figure 1.4: A simple 2D vector field partitioned into two good (left) and two poor (right) regions, from a homogeneity standpoint. The smaller top right copies represent the information reduction after conversion to the flow graph: Each region is represented by only one average velocity. Clearly, the good partition retains more information about the flow field. The poor partition however, is almost as descriptive as using only one region for the whole field (small blue figure).

The derivation of resistance graphs from flow graphs is not part of this work, as it depends on the underlying simulation algorithm.[4] It is assumed however that the resistance graph simulation is reversible, i. e., that an algorithm computing a flow graph from a resistance graph can be reversed.

One benefit of such an automated mapping from flow fields to resistance graphs would be deeper understanding of resistance graphs. Instead of relying on experts to create correct resistance graphs, research could be done on ways or rules to create sound graphs using the repeatable results of automatic resistance graph generation.

Another hope of this thesis is that the resistance graph computed from a flow field may produce sufficiently correct simulation results for nearby operating points. This would reduce the number of required CFD simulations without losing considerable accuracy. Figure 1.5 explains the hybrid design process if this hope is justified. Also notice that if the operation point range is completely covered

[4] The research group at ViF assumes that KULI is not capable of handling vertex counts in the hundreds. Therefore the derivation of a KULI resistance graph from a flow graph was not performed.

by CFD-derived resistance graphs, additional operation points can be added without significantly increasing the computation time.

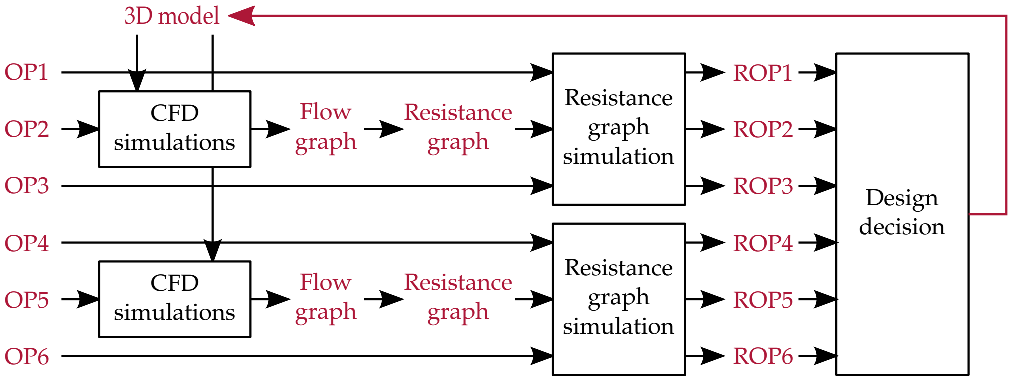


Figure 1.5: Design process using a few CFD simulations and converting them to resistance graphs. CFD simulations are applied to the 3D-model for some operation points (OP2 and OP5), which usually takes a few hours per simulation on a standard workstation. The resulting flow fields of these simulations are then partitioned and converted to flow graphs which again can be converted to a resistance graphs. If the resistance graphs are also valid for similar operation points (e. g., OP1 and OP3 are similar to OP2, OP4, and OP6 are similar to OP5), these similar operation points can be directly simulated using the resistance graphs and computation time can be decreased significantly. The process is an iterative optimization process, i. e., after design changes are made, simulations have to be redone.

1.2 THE TOYOTA DATASET

The CFD simulation results used for this thesis were provided by the research group at Virtual Vehicle Competence Center (ViF) and are based on the Toyota Prius dataset [29]. CFD simulation results for three operation points were provided: At 30 km/h, at 190 km/h, and at $v_{\max}$ (262 km/h). The CFD result at $v_{\max}$ is shown in Figure 1.6. The figure also shows a problem with cells outside the hood boundaries, which has to be overcome in preprocessing.

The dataset uses the outside hull of the Prius, but compared to a modern car, the inside looks empty. It turned out that this did not invalidate the applicability to real world problems.

The 3D unstructured grid is divided into approximately 9.1 million cells which are based on approximately 8.9 million points. It features four basic cell types, with hexahedra filling most of the space and other cell types filling in the seams and details. The four cell types are shown in Figure 1.7.

As explained in Section 1.1.2 the CFD simulation computes flow parameters at each cell. At the end of the simulation however, flow parameters are assigned to the grid points – the corners of the cells – for the Toyota dataset. Therefore, one can also think of the simulation output as dense point cloud, with fixed flow parameters at every point. From this point of view, the cells only add connectivity information.

Each point of the point cloud carries at least seven values and is therefore at least 7-dimensional. The seven dimensions are made up

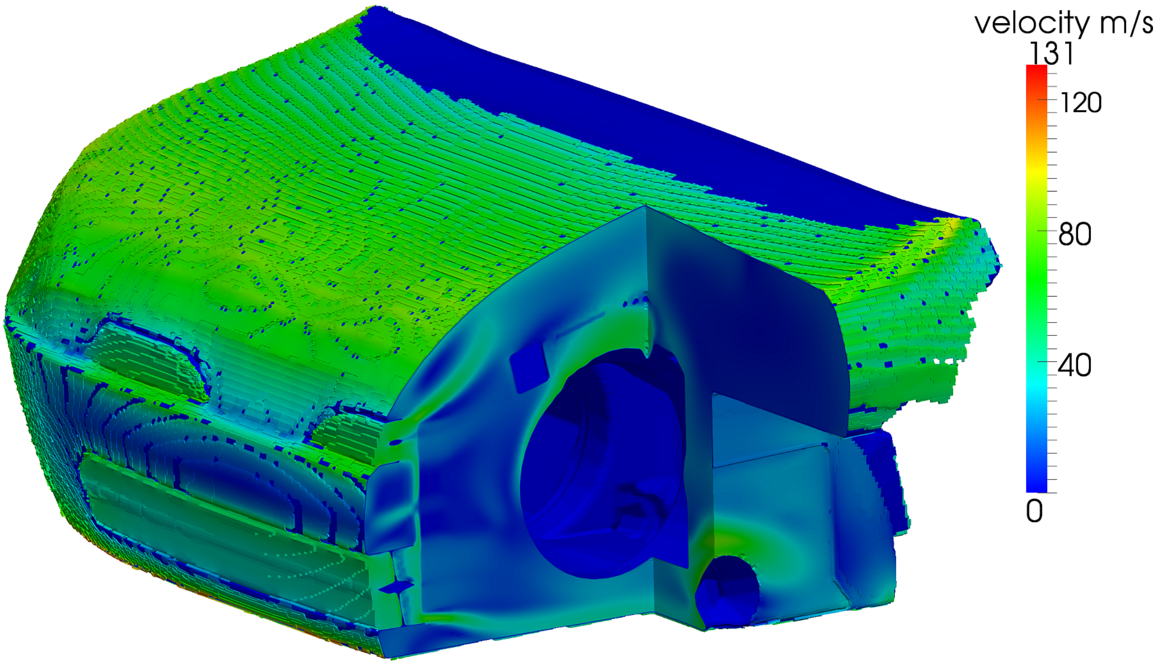
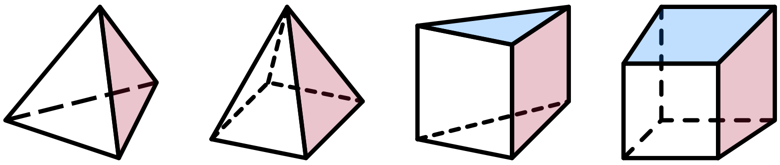


Figure 1.6: The Toyota dataset used for most demonstrations in this thesis at v_{max}. The car hood is shown from the front with the three main inlet openings visible. The cut-out was added for illustration purposes. The colors represent velocity magnitude.



(a) Tetrahedron) (b) Pyramid (c) Wedge (d) Hexahedron

Figure 1.7: Cell types used within the Toyota dataset's 3D-mesh.

by three values for its Euclidean position, three values for the velocity vector, and one value for the pressure.

The problems of the dataset and required preprocessing steps are described in the theory and implementation chapters (Section 3.3 and Section 4.2).

1.3 UNDERSTANDING 3D FLOW FIELDS

Computers are useless. They can only give you answers.
– Pablo Picasso

After being introduced to the specific car cooling problem in the previous sections, one might think that the solution to the problem is also very specific. This is not the case. As explained, the key to the solution is an abstract subproblem:

> *Given:* A flow field, i. e., a dense mesh of probably millions of elements with assigned velocities and pressures.

> Find a sparse flow graph that preserves as much information of the flow field as possible, i. e., a graph where vertices represent homogeneous regions and edges represent flow between these regions.

The purpose of computation is insight, not numbers.
– Richard Hamming

Having a solution to this problem not only aids in designing car cooling systems. It also helps in *understanding* flow fields by reducing redundancy and stressing important features. One straightforward application of using flow graphs to understand flow fields is *interactive flow exploration*. Section 5.3 shows one example of how flow graphs could be used to build a graph based flow exploration tool. Notice, however, that a flow graph is a good tool for visualizing and understanding the coarse features like volume and mass flow rates or pressure distributions, but it is not suited for analyzing small local features like vortices. A region containing mainly small curls compared to the flow graph resolution is a dead end for flow graphs: Incoming and outgoing volume flow rates are almost zero. Circular flows that are larger than the flow graph resolution on the other hand, can be represented by circles in the flow graph.

1.4 RELATED RESEARCH TOPICS

Several research topics are related to partitioning flow fields into homogeneous regions. Two related topics come immediately to mind: *cluster analysis* from statistics and *image segmentation* from image processing. The following subsections shortly explain these research topics and discuss the applicability of their results to flow field partitioning.

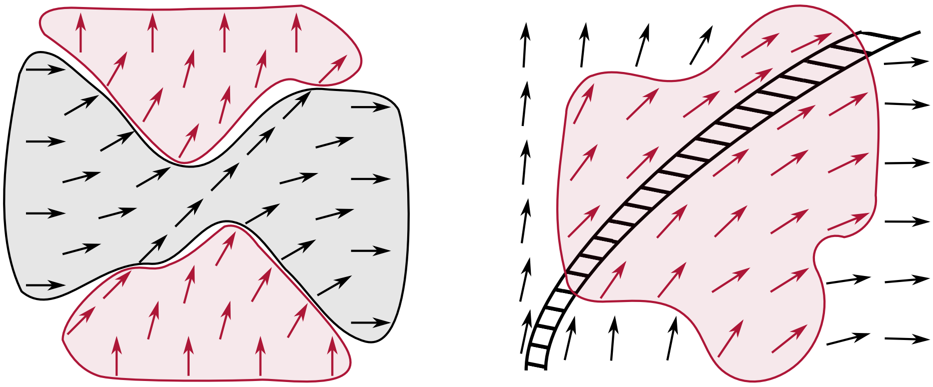
1.4.1 *Applicability of cluster analysis*

There is no exact, commonly agreed on definition for *cluster*. Everitt even states that "... it turns out that such formal definition is not only difficult but may even be misplaced." [11]. An informal definition might be that cluster analysis tries to *group similar objects* together into clusters, while trying to *separate dissimilar objects*. One important property of all clustering algorithms is that the grouped objects are statistical n-dimensional tuples or vectors. Each tuple is usually represented by a point in n-dimensional space and "similarity" of points is usually defined by a *distance function* between these points. Therefore algorithms from cluster analysis can easily be applied to the 7-dimensional point cloud described in Section 1.2.

The problem with this approach is that almost all advanced clustering algorithms work *solely* on point clouds (e. g., k-means, kernel based methods, density based methods). This implies that they cannot take connectivity information into account. After clustering the point cloud to point clusters, the clusters need to be mapped to the cell grid. Some of the resulting regions in the cell grid might not be connected or some point cluster may even be completely scattered on the cell grid. See Figure 1.8a for a simple example of this problem. To produce the required *compact* clusters w. r. t. spatial position, the distance measure must therefore put strong emphasis on the spatial distances in 3D-space between the points. This however limits the algorithm's potential to cluster according to flow parameters. The main problem for utilizing clustering algorithms is therefore to derive a reasonable distance function that accounts both for spatial compactness and flow homogeneity.

Another problem for using cluster algorithms is existing geometry which separates the dataset, i. e., areas where constructional elements divide the CFD volume. As the CFD volume only covers the fluid, there are no cells within these constructional elements and no clustering should occur through these parts. As cluster algorithms only use point clouds, small model parts (like thin plates), can easily be "jumped over", leading to clusters as shown in Figure 1.8b.

In addition not all cluster algorithms qualify for the problem because of computational limits (i. e., about 8 million 7-dimensional points).



(a) Non-connected cluster.

(b) Clustering over mesh boundaries.

Figure 1.8: Problems with using cluster analysis approaches to partition a flow field. The left image shows one non-connected cluster (red), which results from too strong emphasis on flow parameters and too weak emphasis on Euclidean distance. The right image shows a cluster (red) going over constructional parts (hatched), because clustering algorithms work on point clouds and do not utilize connectivity information.

The simple problems in Figure 1.8 can be overcome by splitting the problematic clusters, but more complex cases exist for both of these problems. Despite the mentioned problems, point cluster based approaches are employed to partition flow fields. Some approaches that utilize k-means on the mesh points are discussed in Section 2.1.2.

1.4.2 *Applicability of image segmentation*

Image segmentation is the wide field of extracting edges and homogeneous regions from digital images in order to aid image processing and improve image understanding. Although methods of cluster analysis were successfully employed in image segmentation (most prominently k-means), many image segmentation algorithms utilize the inherent neighbor information of pixels in images.

Naturally, image segmentation methods which extract homogeneous regions from 2D images could be applicable to 3D CFD data partitioning too. The pixels of an image correspond to the cells and the color information corresponds to the attached values (i. e., velocity and pressure).

The first problem for applying image segmentation algorithms arises from the unstructured grid in CFD data. Pixels in images have a uniform structure and neighborhood as well as fixed size. This is not the case for unstructured 3D-grids. More problems arise from the additional dimensions of CFD data. The following prominent image segmentation methods can theoretically be adapted to unstructured 3D-data. They are shortly described and their applicability is discussed.

Edge based image segmentation methods try to find the abrupt intensity changes between different regions and use these *edges* to identify the regions. In 3D this means finding complexly shaped surfaces that separate regions, which is geometrically difficult but possible. However, the fact that there are no identifiable abrupt changes in a typical flow field renders these methods useless for flow field partitioning.

Other approaches that lead to great results in region based image segmentation are watershed algorithms. They interpret gray-level images as topological surface and incrementally increase or decrease water levels. Then merging "water ponds" define region borders. The approach is not applicable to CFD data firstly because the resulting topology would be at least four-dimensional and secondly because there is no simple method to incorporate all flow information into the single fourth dimension.

Energy based methods, like the influential Mumford-Shah energy functional [23], minimize a cost function that contains terms for regions, edges, and intensity. This cost function could be extended to work in 3D and it would contain terms for regions, surfaces separating regions and flow information like velocity and pressure. The problem is that minimizing the complex cost function for large CFD datasets is expected to be computationally infeasible.

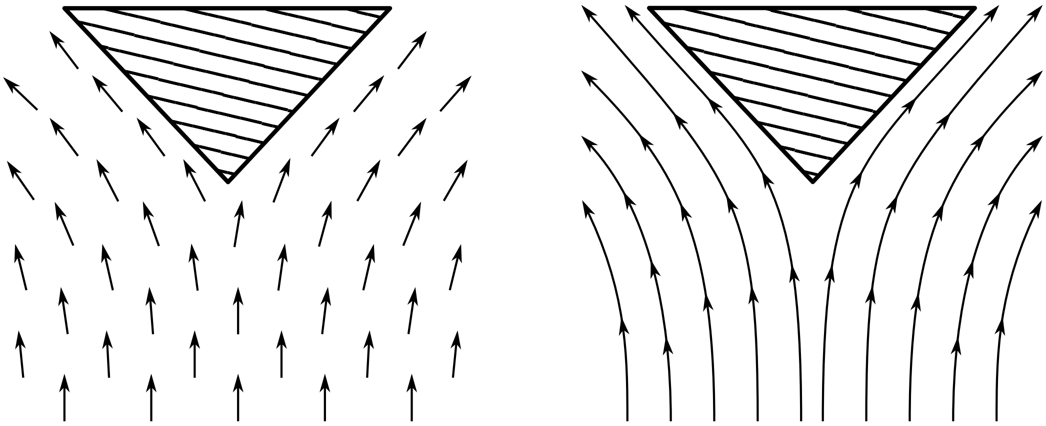
Obvious examples of applicable simple image segmentation algorithms are region merging, region growing, and region splitting. They are usually too greedy at a local level to yield good image segmentations and similar results are expected for flow fields. A variety of region growing ("rapid flooding") is used within McKenzie et al.'s flow field partitioning method [21]. A region splitting approach for vector fields was introduced by Heckel et al. [14]. Both approaches are described in Chapter 2.

1.5 PROPOSED SOLUTION

Alice: *Would you tell me, please, which way I ought to go from here?*
The Cat: *That depends a good deal on where you want to get to.*
Alice: *I don't much care where.*
The Cat: *Then it doesn't much matter which way you go.*
Alice: *. . . so long as I get somewhere.*
The Cat: *Oh, you're sure to do that, if only you walk long enough.*
– Alice in Wonderland [4]

The previous section has shown that some ideas from cluster analysis possibly lead to good flow field partitions, but their application is not straightforward and difficulties need to be overcome. The main challenge for these approaches is to find a suitable distance function, but a function meeting the requirements is not easy to find.

The proposed solution is completely different from the described standard approaches. It circumvents the problem of finding a distance function in an elegant way, by utilizing *streamlines*. A short introduction on streamlines was already given in the context of CFD simulation result interpretation in Section 1.1.2. More information can be found in Section 3.4.2. The important property of streamlines is that they incorporate both spatial information and velocity information in the same geometric primitive (curves in 3D). Figure 1.9 shows a simple example of a flow field and its dense streamline representation. In the streamline representation the clustering task becomes obvious: *Find bundles of streamline segments which are parallel.*



(a) Flow field representation. (b) Dense streamline representation.

Figure 1.9: A simple flow field with a separating construction element (hatched triangle). The flow field representation, with velocities at grid points, is presented on the left. The dense streamline representation, with curves conveying information about both location and flow direction, is presented on the right. The task of clustering connected cells with similar flow directions in the flow field can be translated to the task of finding bundles of parallel streamline segments in the streamline representation.

Figure 1.10 shows two intuitive partitions for the streamline representation of Figure 1.9. Short bundles of many streamlines, as shown in Figure 1.10a, are advantageous for deriving resistance graphs.

Clustering streamline segments also allows finding bent bundles, which are interesting in visualization scenarios (see Figure 1.10b).

Building on this idea the following four-step-solution is proposed:

1. *Dense streamline generation*: Generate dense streamlines, covering the whole flow field, using standard methods.

2. *Streamline bundling*: Find bundles of streamline segments which are close and parallel.

3. *Region mapping*: Map back the streamline bundles to CFD cells to produce a partition of the flow field.

4. *Flow graph generation*: Generate the flow graph from the flow field partition by integrating cell values over regions and over interfaces between regions.

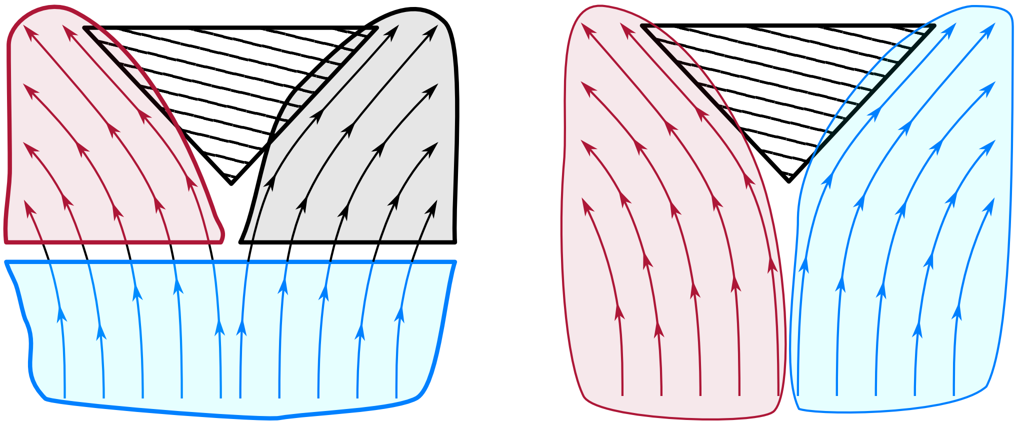


Figure 1.10: Natural bundlings of the simple streamline example. For resistance graph generation, parallel bundles of streamlines are advantageous (left). For visualization purposes on the other hand, bent bundles like the one shown at the right can be more interesting.

2 RELATED WORK

Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size.
– John of Salisbury

In addition to the initial, general considerations given in Section 1.4 this section treats works from two related research fields.

The first is *visualization of vector fields*. Some works of this field utilize clustering to reduce the information contained within large vector fields. Therefore, the overlap with this thesis is in the *computation result*, i. e., finding clusters in vector fields.

The second is *medical image processing*, in particular work on interpreting and visualizing scans of fibrous organic structures. Some of the approaches involve *streamline clustering*. The association with this thesis is therefore in the *method*, i. e., finding and bundling similar streamlines.

This chapter will now discuss important work from both of these fields, starting with *vector field visualization*.

2.1 RELATED WORK FROM VECTOR FIELD VISUALIZATION

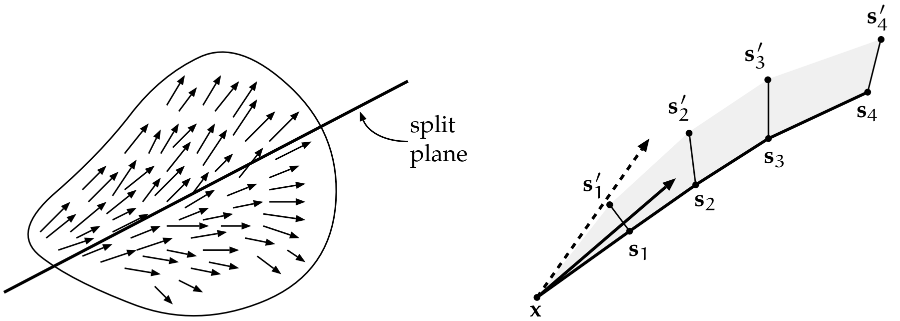
Visualizing vector fields for human interpretation is difficult for two main reasons:

1. It usually involves large amounts of partially redundant data.
2. It is initially unclear which aspects of the data are important to the user.

The former mainly makes the task more difficult, where the latter motivated the development of many different strategies for vector field visualization. Some of these strategies contain vector field clustering as a preprocessing step, others use it directly for visualization. This section describes the most promising candidates for flow graph generation in the context of this work.

2.1.1 *Vector field hierarchies*

Heckel et al. introduce a method for vector field clustering using a region splitting approach [14]. Starting with the whole dataset, they create a Binary Space Partitioning (BSP)-tree by recursively splitting the vector field using planes (Figure 2.1a). Their algorithm uses the grid points only, without taking connectivity information into account. The problem of irregular or disconnected clusters – which is often an issue when neglecting connectivity information (see Figure 1.8) – is mitigated, because the resulting regions are obviously always convex and connected. The hierarchical BSP-tree representation has many advantages in visualization, especially for adaptive Level of Detail (LOD) calculations and visualizations.



(a) 2D-example of a cluster and one desirable splitting plane.

(b) The streamline based error measure for point $\mathbf{x}$ is the summation of the individual point pair distances.

Figure 2.1: Visualization of the main ideas of Heckel et al. (Images taken from Heckel et al. [14]. The font of the labels was adapted.)

The flow of each region at each level is described by one average position and vector. These are created by averaging over all original positions and vectors of the grid points within that region. The vector field at each level is continuously interpolated from these points using Hardy's multiquadric method [13].

In order to find splits, they define a visualization driven error measure which is based on streamlines. To find the approximation error of a given point $\mathbf{x}$, a streamline is traced starting from this point in both the original and the simplified field. They define the error of this particular point by the deviation of the two streamlines, which are described by their points $\mathbf{s}_i$ and $\mathbf{s}'_i$ (see Figure 2.1b):

$$\epsilon(\mathbf{x}) = \sum_{i=1}^{n} \left\|\mathbf{s}_i - \mathbf{s}'_i\right\|. \tag{2.1}$$

The error of a whole cluster $\mathcal{C}$ is then defined as maximum of the individual errors of its points:

$$\epsilon(\mathcal{C}) = \max_{\mathbf{x}\in\mathcal{C}} \epsilon(\mathbf{x}). \tag{2.2}$$

The position and orientation of the splitting plane P for a cluster $\mathcal{C}$ is derived as best fitting plane to all points of the cluster, where points with low error are weighted higher. The algorithm is as follows:

1. Compute the weights of each point inversely proportional to their errors:

$$w(\mathbf{x}_i) = \frac{1}{W\epsilon(\mathbf{x}_i)}, \quad \text{were} \quad W = \sum_{x_j\in\mathcal{C}} \epsilon(\mathbf{x}_j)^{-1},\ \epsilon(\mathbf{x}_j) \neq 0. \tag{2.3}$$

2. The splitting plane goes through the weighted average point $\bar{\mathbf{x}}$ of the cluster:

$$\bar{\mathbf{x}} = \frac{1}{k} \sum_{\mathbf{x}_i \in \mathcal{C}} w_i(\mathbf{x}_i) \cdot \mathbf{x}_i. \tag{2.4}$$

3. The orientation of the splitting plane is (quoting) *"the best-fit plane, in the least squares sense, to the set of weighted points*

$$\{w_i(\mathbf{x}_i) \cdot \mathbf{x}_i \mid \mathbf{x}_i \in \mathcal{C}\}." \tag{2.5}$$

The algorithm for building the BSP-tree is now simply to iteratively split the worst current cluster (i. e., the one with highest $\epsilon(\mathcal{C})$) with the uniquely defined plane P for this cluster until a global error threshold or a given number of clusters is reached.

Heckel et al.'s method has several problems in the application area of this work. The main problem is that it cannot guarantee connected clusters for complex dataset boundaries as described in Figure 1.8b, i. e., it potentially clusters over mesh boundaries. In addition the described visualization driven, streamline based error measure might not lead to a good partition from a physical point of view. This could be circumvented by using a "physical" error measure. However, finding good error or distance measures is problematic as described in Section 1.4.1. Lastly, the algorithms decisions (i. e., splits) are both *local* and *greedy*, and are not expected to lead to globally optimal results.

Overall, the approach solves the problem of scattered clusters and finding a distance measure elegantly. In addition it allows fine control over the number of clusters and the associated approximation error. It therefore applies well to the vector field visualization problem, but is not a first candidate for deriving flow graphs.

2.1.2 k-*means clustering*

A non-local and frequently utilized standard clustering algorithm is k-*means*. It is attributed to Lloyd, who introduced the idea in a signal processing paper in 1957 (published 1982) [17]. The algorithm is hence often referred to as *Lloyd's algorithm*. The term k-*means* and a description from a clustering perspective can be found in [18].

The basic algorithm works as follows: Given n points or vectors $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$, each with m dimensions, a distance function between two points $d(\mathbf{p}_i, \mathbf{p}_j)$ and the number of desired clusters u,

1. select u starting points (*seeds*) randomly from $\mathcal{P}$ as initial cluster centers $\mathcal{C}^0 = \mathbf{c}_0^0, \ldots, \mathbf{c}_{u-1}^0$ (*initialization*),

2. assign each point $\mathbf{p}_i$ in $\mathcal{P}$ to its closest cluster center $\mathbf{c}_j^k$ in $\mathcal{C}^k$, according to distance function $d(\mathbf{p}_i, \mathbf{c}_j^k)$ (*assignment step*),

3. compute the new cluster centers $\mathbf{c}_j^{k+1}$ in $\mathcal{C}^{k+1}$ as the mean of all points assigned to $\mathbf{c}_j^k$ (*update step*),
4. if any stopping criterion is met, exit, otherwise go to step 2 with $\mathcal{C}^{k+1}$ (*iteration*).

The usual stopping criterion is the minimal total change of the cluster centers, e. g., $\sum_{j=0}^{m-1} \|\mathbf{c}_j^k - \mathbf{c}_j^{k+1}\|$. Another possible criterion is the maximum number of iterations k_{max}.

The algorithm aims to find the optimal k-partition of the points $\mathcal{P}$, i. e., the partition which minimizes $\sum_{i=0}^{n-1} d(\mathbf{p}_i, \mathbf{c}(\mathbf{p}_i))$, where $\mathbf{c}(\mathbf{p}_i)$ is the cluster center $\mathbf{p}_i$ is assigned to.

Finding the globally optimal partition for the Euclidean distance function is Non-deterministic Polynomial-time hard (NP-hard) for all non-trivial configurations ($u \geqslant 2$ or $m > 2$) [1, 20]. The k-means algorithm is therefore only a heuristic and may lead to local minima, depending on the initial seeds. For small instances of the problem, this is usually mitigated by repeating the algorithm several times with different initial seeds.

The problem sizes of underhood flow vector fields do not allow repeated application of the algorithm on current computers. In addition k-means suffers of the typical problem of purely point based algorithms when applied to vector field clustering: scattered or even disconnected clusters.

2.1.3 *Centroidal Voronoi tessellation*

One sophisticated point based k-means algorithm for vector field visualization and segmentation was devised by Du and Wang [9]. They introduce Centroidal Voronoi Tessellation (CVT) and show that their k-means algorithm produces such tessellations.

A tessellation (term from geometry) for continuous spaces is what a partition (term from set theory) is for sets. For our cause they are the same.

A tessellation of a set $\Omega \subseteq \mathbb{R}^n$ consists of regions $\{V_i\}_{i=1}^k$ which are gap free and non-overlapping.

A *Voronoi tessellation* or *Voronoi diagram* of Ω consists of the regions $\{\widehat{V_i}\}_{i=1}^k$. It is defined by a set of points $\{\mathbf{z}_i\}_{i=1}^k \in \mathbb{R}^n$ which are termed *generators* and a distance function d_x. The Voronoi region for each point $\mathbf{z}_i$ is then defined as the union of all points that are closer to $\mathbf{z}_i$ than to any other point in $\{\mathbf{z}_i\}_{i=1}^k$ according to the distance function d_x (possibly one-sided):

$$\widehat{V_i} = \left\{ \mathbf{x} \in \Omega \mid d_x(\mathbf{x}, \mathbf{z}_i) < d_x(\mathbf{x}, \mathbf{z}_j) \quad \text{for} \quad j = 1, \cdots, k,\ j \neq i \right\}. \quad (2.6)$$

A *Centroidal Voronoi Tessellation* is a Voronoi tessellation where the generators $\{\mathbf{z}_i\}_{i=1}^k$ are also the *mass centroids* of their clusters. The mass centroids or mass centers $\mathbf{z}^*$ are the minimizers of the energy defined by the sum of squared distances between all points and their respective generators:

$$E(\mathbf{z}, V) = \int_V d_x^2(\mathbf{x}, \mathbf{y})\, dx. \tag{2.7}$$

Du and Wang also define a *total energy* as the sum of the energies of all regions. After initialization with n randomly selected points, their algorithm tries to optimize this total energy by alternating updates of the cluster centers and the cluster regions (k-means).

For distance computations they define a one-sided distance measure d_p from a cluster point $p = (\mathbf{x}_p, \mathbf{v}_p)$ with position $\mathbf{x}_p$ and velocity $\mathbf{v}_p$ to a center point $m = (\mathbf{x}_m, \mathbf{v}_m)$ with position $\mathbf{x}_m$ and velocity $\mathbf{v}_m$:[1]

Measure what is measurable, and make measurable what is not so.
– Galileo Galilei

$$d_p(p, m) = \sqrt{\|\mathbf{v}_p\|^2 - \|\mathbf{v}_p\|\mathbf{v}_p \cdot \mathbf{v}_m + w\|\mathbf{v}_p\|^2\|\mathbf{x}_p - \mathbf{x}_m\|^2}. \tag{2.8}$$

The paper features an appendix explaining the reasoning behind this choice. Basically their distance measure contains three parts:

1. the magnitude of the vector ($\|\mathbf{v}_p\|^2$),
2. a term for angular similarity ($\|\mathbf{v}_p\|\mathbf{v}_p \cdot \mathbf{v}_m$, where $\cdot$ is the scalar product), and
3. a weighted term for spatial proximity ($w\|\mathbf{v}_p\|^2\|\mathbf{x}_p - \mathbf{x}_m\|^2$).

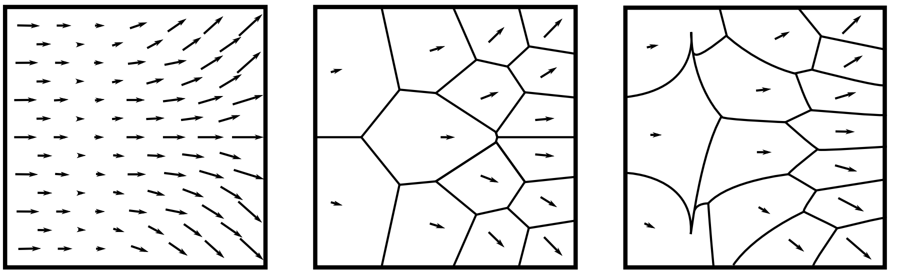
The weight parameter w encodes the relative importance of spatial proximity. Low values of w emphasize flow similarity whereas high values emphasize closeness of points. The authors suggest choosing $w \approx \frac{1}{L^2}$, where L is the spatial diameter of the dataset. Figure 2.2 shows a simple 2D vector field and its tessellation with two different spatial weights.[2]

The authors include several refinements for their method, including update rules for non-uniformly distributed sample points. The paper closes with visualization applications which are of little interest in the context of this thesis.

One problem with this approach is the required computational effort for huge datasets. In the standard implementation, each of n points needs to be compared to each of u cluster centers for each of t iterations. The required computational effort is therefore $O(n \cdot u \cdot t)$. Another problem is again the possible occurrence of irregular or disconnected clusters of both types depicted in Figure 1.8.

[1] Some symbols adjusted to increase clarity.

[2] The velocity magnitudes are not given, but, judging from the weights, they are similar to the spatial magnitudes.



(a) 2D input vector field.

(b) CVT clustering with $w = 0.5$.

(c) CVT clustering with $w = 0.1$.

Figure 2.2: Example for a CVT of a 2D vector field. The vector field is shown at the left, with magnitude proportional to arrow length. In addition two clusterings with $w = 0.5$ and $w = 0.1$ are shown in the middle and on the left respectively. (Original image in Cohen-Steiner et al. [6]. The vector field (left) was simplified. The clusterings (middle, right) are depicted without color.)

2.1.4 *Variational clustering*

One promising approach to solve the problem of disconnected clusters is the paper on *Variational clustering* by McKenzie et al. [21]. Their work builds on renowned work from Cohen-Steiner et al. who introduced *variational shape approximation* for mesh simplification [6]. Therefore Cohen-Steiner et al.'s work will be presented briefly now, followed by McKenzie et al.'s work.

2.1.4.1 *Variational shape approximation*

Cohen-Steiner et al. cast shape approximation as *variational partitioning* problem [6]. Their ultimate goal is to approximate a 3D surface mesh by a predetermined number of flat surfaces which are represented by *proxies*. Starting with an $\mathcal{L}^{\infty,\in}$ distortion metric, they introduce a global distortion measure to describe the deviation of the original shape from the proxies.

Based on this measure, an algorithm similar to k-means is applied, i. e., the alternating application of an assignment step and an update step. Their problem (clustering surface tiles) is similar to the vector field clustering problem, as it also contains connectivity information and it is also undesirable to have disconnected clusters.

Their solution for keeping clusters connected is very elegant. Instead of defining an error measure that contains a strong spatial component, they use a *flooding approach*. By utilizing region growing from seed tiles according to a distortion measure they ensure connected regions. For each resulting region of surface tiles they compute the exact mean (the *proxy*). The most similar tiles to these proxies are then the seed tiles for the next flooding iteration.[3] Figure 2.3 illustrates the approach.

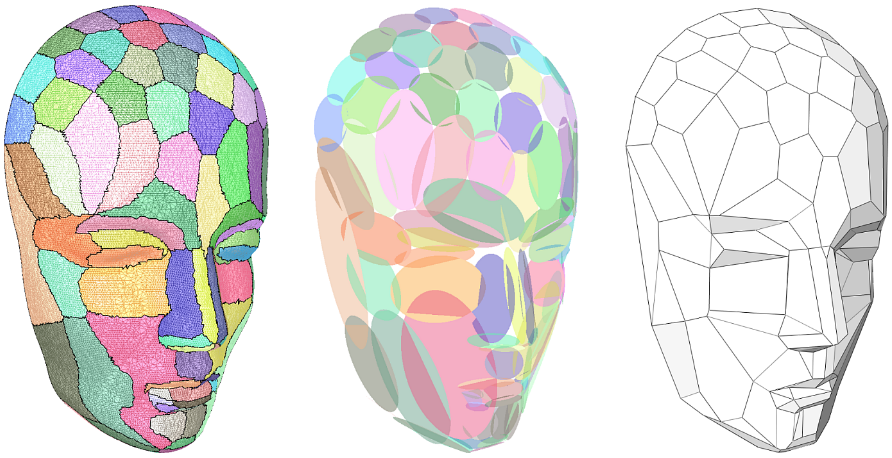


Figure 2.3: Initial mesh partition obtained by flooding (left), proxies for the partitions' regions presented as ellipses (middle), and final simplified mesh (right). (Image taken from Cohen-Steiner et al. [6].)

One drawback of the flooding approach is that small clusters can be trapped in a local minimum between larger clusters, without any possibility to "escape" into areas where they could be more useful. Cohen-Steiner et al. mitigate this problem by applying *region teleportation*, which uses a heuristic for finding trapped regions and moving them to useful areas.

Their algorithm behaves similar to k-means, but produces connected clusters (regions). It requires only one parameter – the number of clusters – and produces very good results in shape approximation.

2.1.4.2 *Adaptations for variational clustering*

McKenzie et al. adapt the approach of Cohen-Steiner et al. to vector fields. Instead of 2D surface regions, their algorithm applies to 3D regions in space [21].[4] Surface elements, like triangles, are replaced by volume elements, like tetrahedrons. The *proxy* of each region is simply defined by a velocity vector $\mathbf{V}_i$ and its position $\mathbf{x}_i$.

The first distortion measure introduced in the paper is *position independent*, i. e., it only uses velocity information. The distortion error of a region $\mathcal{R}_i$ is defined as the integral over the squared $\mathcal{L}^2$-distances between velocities $\mathbf{v}(\mathbf{x})$ at point $\mathbf{x}$ and the proxy $\mathbf{V}_i$:

$$E_{\mathcal{L}^2}(\mathcal{R}_i, \mathbf{V}_i) = \iiint\limits_{\mathbf{x}\in\mathcal{R}_i} \|\mathbf{v}(\mathbf{x}) - \mathbf{V}_i\|^2 \, d\mathbf{x}. \tag{2.9}$$

[3] The exact distortion measures and mean computation formulas are interesting, but irrelevant for this work.

[4] McKenzie et al. treat mainly 2D vector fields in their paper, this summary concentrates on the 3D aspects and applications.

In the discrete mesh, a region $\mathcal{R}_i$ consists of cells i with discrete velocities $\mathbf{v}_i$ and volumes $|P_i|$. Then the distortion error for one region is

$$E_{\mathcal{L}^2}\left(\mathcal{R}_i, \mathbf{V}_i\right) = \sum_{i \in \mathcal{R}_i} \|\mathbf{v}_i - \mathbf{V}_i\|^2 \, |P_i|, \tag{2.10}$$

where the optimal vector proxy for a region is the volume weighted mean of its velocities

$$\mathbf{V}_i = \frac{\sum\limits_{i \in \mathcal{R}_i} |P_i| \mathbf{v}_i}{\sum\limits_{i \in \mathcal{R}_i} |P_i|}. \tag{2.11}$$

The global error of any partition with k regions is then defined as the summed distortion errors of all regions ($E(\mathcal{R}, \mathcal{V}) = \sum_{i=1}^{k} E(\mathcal{R}_i, \mathcal{V}_i)$).

McKenzie et al. report, that this simple distortion error produces "physically relevant partitions". In addition they introduce higher order measures which are based on divergence, gradient, and curl. These higher order measures satisfy specific visualization purposes.

After defining how to measure the distortion and how to compute average proxies from regions, the algorithmic framework of Cohen-Steiner et al. can be applied without adaptation. At first, k random seed cells are picked and their velocities are used as proxies for the flooding stage. Afterwards, the average proxy for each region is recomputed. From every region, the cell with the least distortion from the region's proxy is chosen as new seed cell, and the iteration starts again.

Several stopping criteria can be applied, most importantly the maximum number of iterations and the minimal change of global distortion. The authors also mention that the application of *region teleportation* similar to Cohen-Steiner et al. improves the results, but they do not detail their heuristics. The paper is closed by visualization techniques based on streamline tracing from the cluster centers.

Figure 2.4 shows the clustering result of the algorithm on a 3D car dataset. Notice that the flow field concentrates on the vehicle cabin and the vehicle vicinity and is different from an underhood flow field in both resolution and the regions of interest.

By design, the described algorithm always produces connected regions. Its results are expected to be "close to optimal", even if the optimality proofs of k-means cannot be applied directly. The only input parameters are the number of clusters (u) and the stopping criterion, i. e., the number of iterations (t) or some total error threshold ϵ.

McKenzie et al. themselves do not state the required computational effort, their largest 3D dataset is the car dataset shown in Figure 2.4 and has 1.25 million cells [21]. For the underhood flow application however, the computational effort is a major concern, as the number

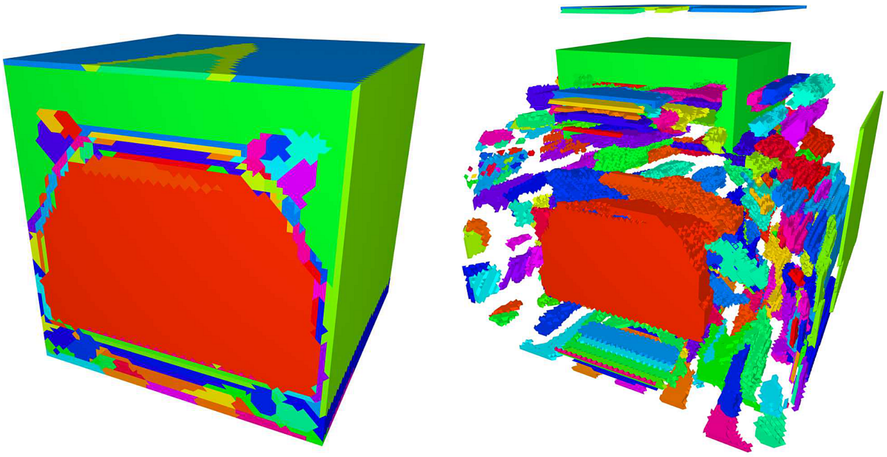


Figure 2.4: 3D clustering result of the flow field "in the wake of a moving automobile, 1.25 million tetrahedra" into 200 clusters at the left. Exploded view of the same dataset at the right. (Image taken from McKenzie et al. [21]. Additional visualization images omitted.)

of cells n is very high. The required effort is $O(n \cdot u \cdot t)$. For 9.1 million cells, 1000 clusters and 20 iterations this leads do 182 billion distance computations.

Notice, that the simple $\mathcal{L}^2$-distance measure given in Equation 2.9 does not incorporate spatial information and only produces compact regions under the assumption that cells with similar velocity lie compact. This might not be the case for the complicated underhood flow dataset and can lead to complex clusters.

Nevertheless, the algorithm is simple and powerful and a first candidate for implementation and comparison. It will therefore be evaluated in the results section (Chapter 5). The two major concerns of the algorithm, namely computational effort and the possibility of complex clusters will also be analyzed and discussed.

2.2 RELATED WORK FROM MEDICAL IMAGE PROCESSING

Similar to vector flow visualization, the visualization of medical diffusion data poses problems because of data redundancy and finding representations suitable for human interpretation. Although the field is not directly related to vector field clustering, some of the methods involve streamline clustering and are therefore interesting for this thesis.

2.2.1 *Introduction to diffusion tensor imaging*

Magnetic Resonance Imaging (MRI) is a widely used non-invasive 3D imaging method in radiology. *Diffusion MRI* is a specialized form of

A vector can be seen as tensor of rank 1. While the mental model for a 3D vectors field is a cloud of arrows, one can think of 3D tensor fields as a cloud of ellipsoids.

MRI which produces a regular 3D grid with one scalar diffusion value per voxel. The results are sufficient for analyzing isotropic regions of tissue. To completely capture areas with anisotropic diffusion, one 3×3 matrix, the *diffusion tensor*, per voxel is required. A diffusion tensor encodes the diffusion rates into all directions. The method to acquire data of this type is called Diffusion Tensor Imaging (DTI). It combines several diffusion weighted images along several gradient directions. As diffusion tensors are symmetric, at least 6 of these images are required, in addition to one reference image without diffusion weighting [16].

2.2.2 *Diffusion tensor imaging tractography*

Fibrous structures are characterized by high diffusion in fiber direction compared to lower diffusion in the other directions, which is restricted by physical or chemical barriers. Therefore, general directions of fibrous structures like muscles in the body or axons in the brain can be derived from DTI scans.

This has proven particularly useful for determining the connectivity of white brain matter. The resulting directions allow tracing streamlines in a similar way as in vector fields. In this domain streamlines are also called *fibers*, referring to their biological meaning. The streamlines trace the direction of neural axons and therefore the connectivity within the brain. However, traced streamlines do not directly represent neural axons, because the dimensions of axon structures are magnitudes below the resolution of DTI, which is a few mm^3 [16] per voxel. Instead they represent general axon directions, capturing several axons going into similar directions [28].

DTI tractography algorithms try to identify and extract the *neural tracts* which connect different parts of the brain. A well researched method to achieve this is to generate the streamlines described above and cluster similar ones to bundles. It is therefore similar to the novel streamline bundling method proposed in this thesis.

The introductory statements on streamline clustering are based on an excellent overview article by Schultz [28].

2.2.3 *Distance measures for streamlines*

The approaches for clustering similar streamlines in DTI tractography are based on distance measures between two streamlines F_i and F_j. The streamlines are represented by their points $\mathcal{P}_i = \{\mathbf{p}_{(i,1)} \cdots \mathbf{p}_{(i,n)}\}$ and $\mathcal{P}_j = \{\mathbf{p}_{(j,1)} \cdots \mathbf{p}_{(j,m)}\}$, respectively, which are implicitly connected by lines. Using this definition of discrete streamlines, two distance measures are commonly used: the *mean distance* and the *Hausdorff distance*.

2.2.3.1 *Mean distance of two streamlines*

The *mean distance* d_μ between the streamlines F_i and F_j is defined as ([28, notation adapted])

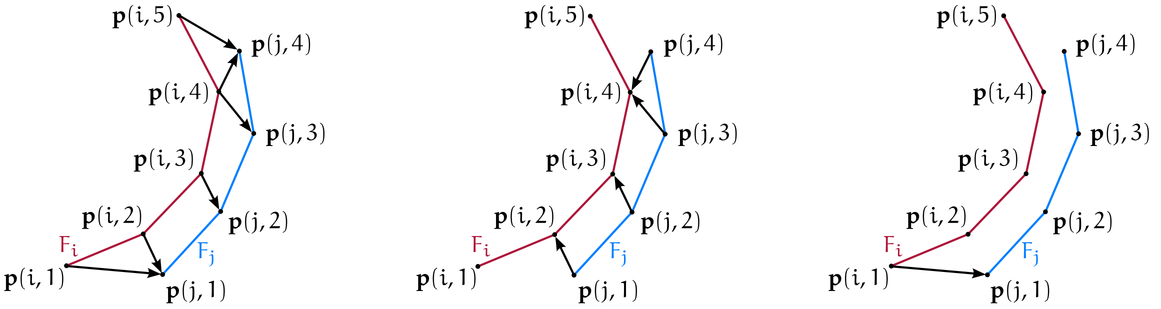
$$d_\mu(F_i, F_j) = d_\mu(F_j, F_i) = \frac{\overline{d_\mu}(F_i, F_j) + \overline{d_\mu}(F_j, F_i)}{2}, \tag{2.12}$$

where the one sided distance $\overline{d_\mu}$ is defined as

$$\overline{d_\mu}(F_i, F_j) = \frac{\sum_{\mathbf{p}_{(i,k)} \in \mathcal{P}_i} \left(\min_{\mathbf{p}_{(j,l)} \in \mathcal{P}_j} \|\mathbf{p}_{(i,k)} - \mathbf{p}_{(j,l)}\| \right)}{|\mathcal{P}_i|}. \tag{2.13}$$

This is the sum of distances from every point in F_i to its closest neighbor in the other streamline.

The one sided distances are depicted in Figure 2.5a and Figure 2.5b. Notice that the streamlines can consist of a different number of points and that the mean distance is symmetric.



(a) One sided streamline distance $\overline{d_\mu}(F_i, F_j)$.

(b) One sided streamline distance $\overline{d_\mu}(F_j, F_i)$.

(c) Hausdorff distance d_H (in this case $d_H = \widetilde{d}_H(F_i, F_j)$).

Figure 2.5: Common distance measures in DTI streamline clustering. The mean streamline distance $d_\mu(F_i, F_j)$ is the average of $\overline{d_\mu}(F_i, F_j)$ (a) and $\overline{d_\mu}(F_j, F_i)$ (b). The Hausdorff distance d_H is the maximum of all closest distances, as shown in (c). (Images based on Schultz [28]. Adapted color and added labels.)

2.2.3.2 *Hausdorff distance of two streamlines*

Another distance measure occasionally used is the *Hausdorff distance* d_H, which is a "worst case distance". It takes the maximum of all the distances used for mean distance computation [28, notation adapted]:

$$d_H(F_i, F_j) = d_H(F_j, F_i) = \max\left(\widetilde{d}_H(F_i, F_j), \widetilde{d}_H(F_j, F_i)\right), \tag{2.14}$$

where the one sided Hausdorff distance is defined as

$$\widetilde{d}_H(F_i, F_j) = \max_{\mathbf{p}_{(i,k)} \in \mathcal{P}_i} \left(\min_{\mathbf{p}_{(j,l)} \in \mathcal{P}_j} \|\mathbf{p}_{(i,k)} - \mathbf{p}_{(j,l)}\| \right). \tag{2.15}$$

Figure 2.5c shows the Hausdorff distance for two simple streamlines. The Hausdorff metric is also symmetric and allows different point counts for the streamlines.

2.2.4 *Streamline clustering approaches*

Many streamline clustering methods based on these and similar distance measures have been proposed. For an extensive summary see the overview paper of Schultz [28]. As an example, the next section will discuss Corouge et al.'s work which also incorporates important ideas for this thesis. Afterwards, other approaches will be described in relation to Corouge et al.'s, and their applicability to this thesis will be discussed.

2.2.4.1 *Corouge et al.'s clustering approach*

Corouge et al. use nearest neighbor clustering, where curves with distances below a given threshold are clustered together. Neighborhood is transitive in their method, i. e., a streamline is part of a cluster if it is close enough to any other streamline within this cluster. Their main distance function is the mean distance d_μ. In addition they employ shape based metrics like length, center of mass and second order moments for clustering. They also suggest the usage of the Hausdorff distance to reject outliers.

After clustering, they fit B-splines to the streamlines within a bundle starting from a common starting plane. The B-splines allow equidistant sampling of all streamlines in a bundle, therefore providing the framework for statistics on shape properties like curvature or vorticity. These statistics can be used for defining a bundle by its *prototype* and the shape property statistics [7].

Notice two aspects of Corouge et al.'s work, which will occur in this works approach later, even if in completely different form:

1. The notion of the prototype, as the main representative of bundles of streamlines.

2. The registration of points on streamlines to each other at similar points on the curve.

2.2.4.2 *Other clustering approaches*

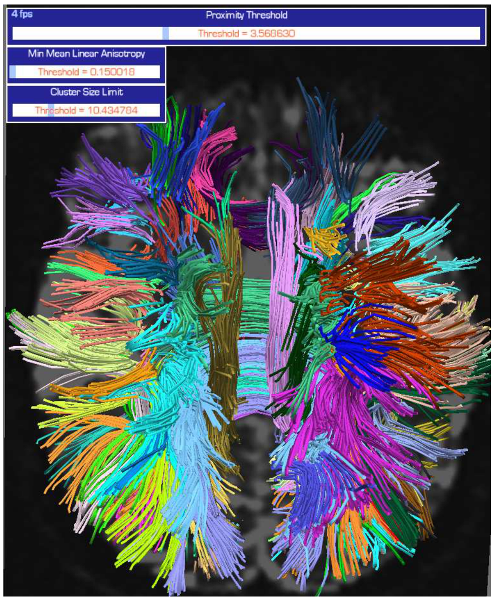
The fundamental differences of other works to Corouge et al.'s are mainly adaptations of the distance measure. These adaptations include

- Ding et al.'s approach of computing distances of succeeding streamline points, except of closest streamline points [8],

- Brun et al.'s simplification of the Hausdorff distance by considering only streamline endpoints [3], and

- Zhang et al.'s sophisticated adaptation of the mean distance d_μ by dropping distances below a *minimum distance threshold* and utilizing the minimum[5] and the maximum[6] of the one sided mean distances instead of their average.

A clustering result example of Zhang et al. is shown in Figure 2.6. Notice that only whole streamlines are clustered together.



Proximity Threshold
Threshold = 3.568630

Min Mean Linear Anisotropy
Threshold = 0.150018

Cluster Size Limit
Threshold = 10.434784

Figure 2.6: Streamline clustering result of axon fiber traces in a human brain by Zhang et al. (Image taken from Zhang et al. [33].)

The usual streamline clustering algorithms in this field use the nearest neighbor scheme and its variants, but region splitting approaches have been introduced too [3, 24]. To reduce the required $O(n^2)$ distance computations for n streamlines, several adaptations have been suggested [32, 19, 24].

The application of streamline-to-streamline distance measures and greedy hierarchical algorithms are successful in DTI fiber clustering, because the goal is to find *bundles of entire streamlines*.

[5] For example $\min(\overline{d_\mu}(F_i, F_j), \overline{d_\mu}(F_j, F_i))$.

[6] For example $\max(\overline{d_\mu}(F_i, F_j), \overline{d_\mu}(F_j, F_i))$

3 THEORY

If you think it's simple, then you have misunderstood the problem.
– Bjarne Stroustrup

The review of existing work showed that existing solutions for vector field clustering suffer from at least one of the following problems:

1. The algorithmic decisions are simple, greedy, and local, which leads to suboptimal results.
2. The algorithm optimizes globally, but potentially creates disconnected clusters for all meaningful distance functions.
3. The algorithm requires high computational effort.

The aim of this thesis is therefore to propose and evaluate a new method which avoids the first two problems, while still requiring only moderate computational effort. The core method investigated in this thesis is *streamline bundling*. In addition applicability of the best candidate of existing methods, namely McKenzie et al. clustering, is evaluated.

Instead of describing the core method right away, the theory behind the algorithmic components will be presented in the order of processing. After introducing the mathematical notation, an overview of the processing framework is given. It is then followed by the theoretic background for each of the individual processing blocks. This approach provides a natural structure for the remaining chapter and allows to easily match the contents of the theory and the implementation chapters.

3.1 MATHEMATICAL NOTATION

In the previous chapter, the mathematical notation of related work was retained closely to the original. For the remaining work however, consistent notation will be used.

This notation is employed to enable usage of standard labels and letters without creating too much confusion.

For *mesh entities* the usual mathematical font will be used: d_i for points, c_i for cells, and $f_{i,j}$ for cell faces. Sets of these entities will be typeset using calligraphic font, e. g., $\mathcal{R}_i$ for sets of cells (regions) and $\mathcal{I}_{i,j}$ for sets of cell faces (interfaces). Bold letters depict vectors.

For *graph entities* typewriter font will be used: $\mathtt{v}_\mathtt{i}$ for vertices, $\mathtt{e}_{\mathtt{i},\mathtt{j}}$ for edges, and $\mathtt{G}(\mathtt{V},\mathtt{E})$ for a whole graph and its vertex and edge set.

$\mathbf{v}(\mathtt{v}_\mathtt{i})$ is the velocity at graph vertex $\mathtt{v}_\mathtt{i}$ and $\mathbf{v}(d_i)$ is the velocity at mesh point d_i.

All mesh and graph entities can feature attached values, e. g., the velocity at a mesh point, the volume of a mesh region or the volume flow rate along a graph edge. For these attached values, *function notation* will be used, e. g., $\mathbf{v}(.)$ is the velocity of an element, $\mathbf{x}(.)$ is its position, and $p(.)$ is its pressure. Notice that many functions apply to both mesh entities and graph entities. The actual meaning of different functions will be explained right before usage.

3.2 PROCESSING FRAMEWORK OVERVIEW

Though this be madness, yet there is method in it.
– William Shakespeare

To derive flow graphs from CFD flow fields, several algorithmic steps have to be performed. Some algorithmic steps are interchangeable, other can be omitted. Figure 3.1 shows an overview of all important algorithmic steps for this thesis.

The chart is divided into four major stages (blue rectangles and text). In the *preprocessing stage*, the deficiencies of the input data are removed and it is prepared for further processing. In the *partitioning stage* the input data is partitioned into similar regions by one out of three processing options: Streamline bundling, McKenzie et al. clustering, or k-means clustering. The *graph mapping stage* replaces these regions by single vertices within a flow graph. The according edge data and vertex data is created by discrete integration over the regions and their boundaries. One exception of this workflow is the direct computation of approximate flow graphs from the result of streamline bundling. Ideally, the resulting flow graph describes the flow well enough to be utilized in an *application stage*.

On a finer level, the figure consists of processing blocks (black rectangles with text) and intermediate data (maroon text). The arrows depict data flow and show that several paths of processing are possible through the graph. The "Graph collapse" block is optional (black dashed) and can be omitted. Any path from the input (top) to one of the application blocks (bottom) describes a valid processing chain. The possible chains differ in required computational effort and in the quality of the results.

3.3 PREPROCESSING OVERVIEW

When importing CFD data, or any complex data set for that matter, care has to be taken about data inconsistencies and incompatibilities. The occurring problems depend on the import/export functionality of the used programs, i. e., the tasks of the preprocessing step depend on the employed software.

The Toyota dataset was generated by a FLUENT 12 [12] simulation and post-processed using the EnSight [10] software package. The dataset shows two major problems after loaded into VTK [30]:

INCONSISTENT NORMALS: The point order of some of the cells is wrong. Therefore the surface normals are pointing inwards for some cells and outwards for others.

OUTER GEOMETRY: The dataset contains a thin layer of extra cells which are outside of the actual car.

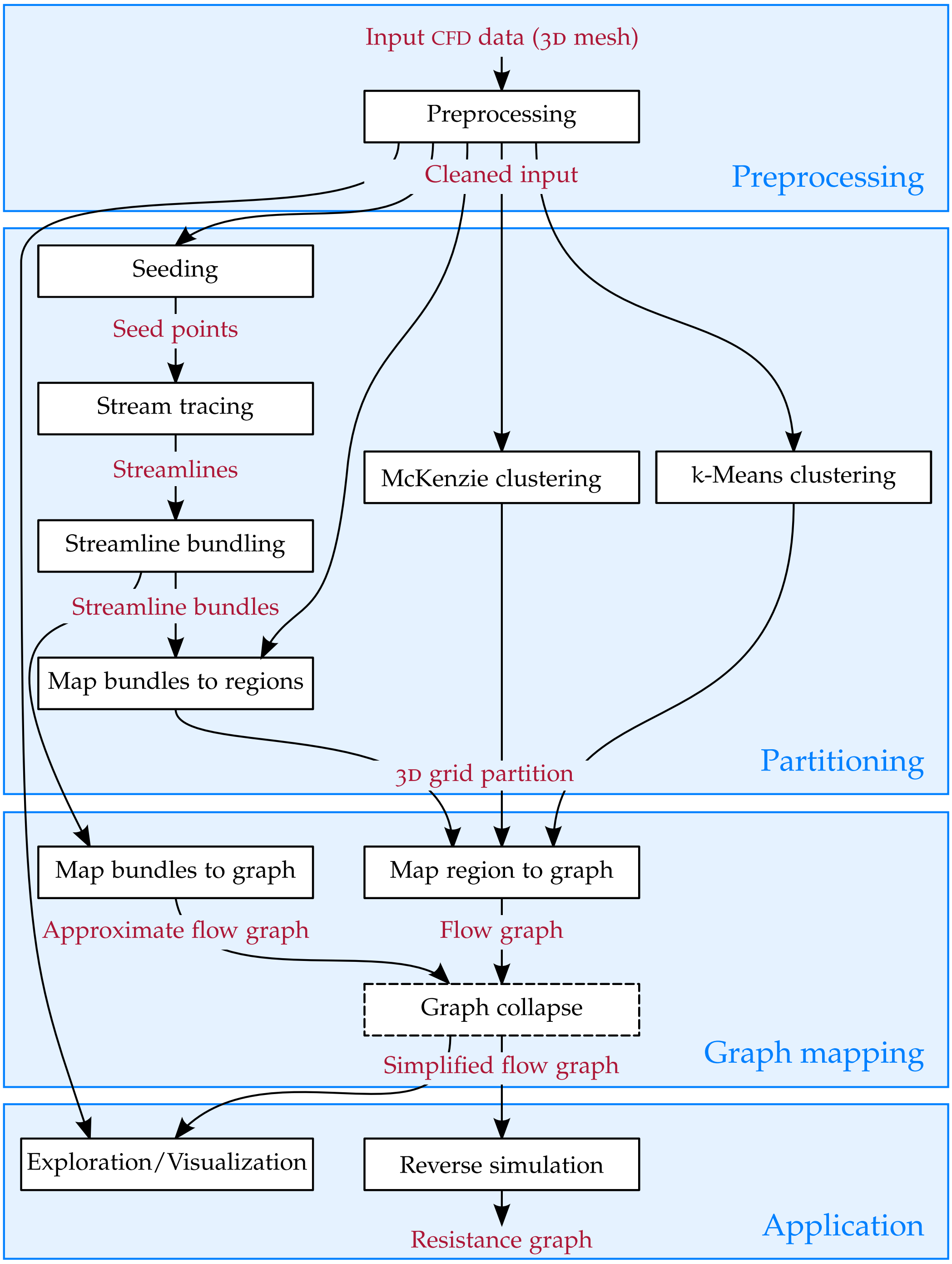


Figure 3.1: Overview of the important processing modules for this thesis. The global view consists of four stages (blue rectangles and text). These stages contain processing blocks (black rectangles and text) and intermediate data (maroon text). Any path from top to bottom is valid, i. e., some of the processing block groups are interchangeable with others. The possible paths differ in processing time and quality of the results. Two applications of flow graphs are considered, firstly visualization and exploration and secondly the generation of resistance graphs. Some paths are better suited for visualization, while others are better suited for producing resistance graphs.

The simple problem of inconsistent surface normals is completely treated in the implementation chapter (Section 4.2.1). The outside geometry problem is more difficult and will be introduced here.

3.3.1 *Undesired outside geometry*

Lisa: *The basis of this game seems to be simple geometry. All you have to do is hit the ball . . . here. (The ball is hit, gets bounced around, and goes into the hole.)*

Bart: *I can't believe it. You've actually found a practical use for geometry!*

– The Simpsons

The Toyota dataset was cut out from a complete CFD simulation box which simulated the inside and the outside of the car together. This cutting process left some outside geometry attached to the dataset.

Figure 3.2 demonstrates this situation. The left image shows the Toyota dataset from the back left side with the magnitude of velocity encoded by color. The high velocity areas at the surface of the hood (green) are undesired cells lying outside of the car. Dark blue areas represent the low velocity right next to inside surfaces of the vehicle hull.

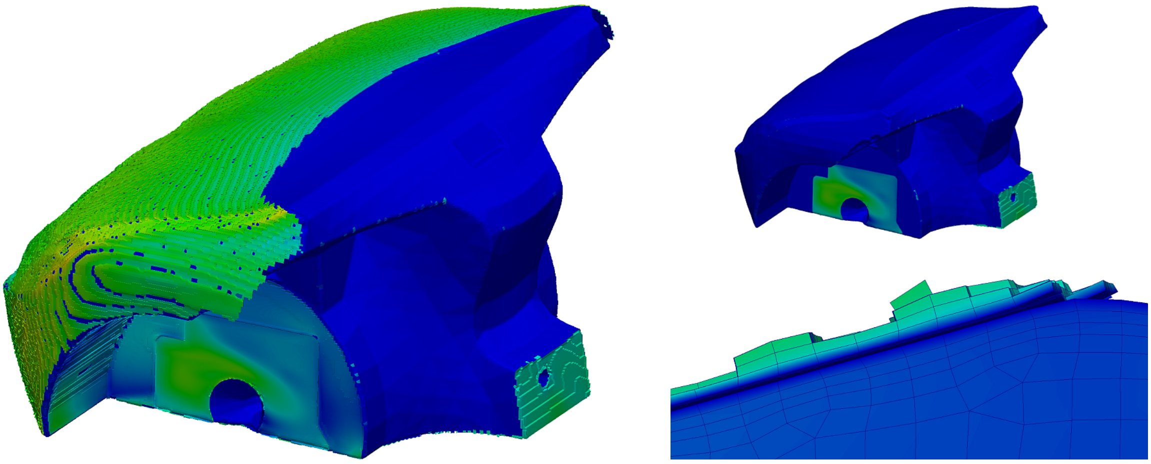


Figure 3.2: Undesired cells lying outside of the car. The left image shows an outside view of the unaltered Toyota dataset colored by velocity magnitude. The greenish cells are outside of the vehicle hull and need to be removed, whereas dark blue cells are inside the car and need to be kept. The right bottom image shows the close-up view of a cut through the car. The outside cells (top) and inside cells (bottom) are separated by a cell-free zone – the metal of the car hood. The top right image shows the smooth, post-processed dataset for comparison.

The right bottom image shows a zoomed view of a longitudinal cut through the car. The cell-free space separating inside and outside geometry was filled by the metal of the car hood during CFD simulation. Outside and inside cells are therefore only connected at holes in the vehicle hull (i. e., air inlets and outlets). At the top right, the dataset is shown without any outside geometry. All surface areas are smooth and blue, except of real interfaces to the outside.

The three main reasons outside geometry needs to be removed are:

1. It makes the task of automatically finding air inlets and outlets difficult.[1]

2. It potentially wastes clusters for outside geometry or makes border clusters inaccurate if they contain outside geometry.

3. It makes volume flow statistics over the dataset imprecise (e. g., total input flow and output flow).

For separating the outside from the inside cells, a criterion differentiating them is required. Two observations and the resulting algorithmic ideas for removing outside cells will be discussed.

The first observation is that undesired regions are rough at the surface, whereas desired regions are flat at the surface. This leads to a *region growing* approach starting from exposed cells.

The second observation is that undesired regions are thin and contain mainly stacked hexahedra and wedges. This motivates a *depth probing* approach.

Details of these two approaches will be presented in Section 4.2.2.

3.4 partitioning

The partitioning phase takes the cleaned input mesh from the preprocessing phase and outputs a partition of this input mesh.

In Figure 3.1 three alternative approaches to partitioning can be identified. The theory for McKenzie et al. clustering (middle) was already treated in Section 2.1.4. k-means clustering (right) was already treated in Section 2.1.2. Several distance functions for k-means are possible, a good candidate is the distance function described for CVT in Section 2.1.3 (Equation 2.8).

The leftmost partitioning option of Figure 3.1 is the core matter of this thesis and consists of four processing blocks:

1. During the *seeding* phase seed points are selected from the mesh.

2. These seed points are used as starting points for tracing dense streamlines in the *stream tracing* phase.

3. *Streamline bundling* finds bundles of similar streamline segments.

4. These bundles are then mapped back to the cells of the 3D mesh to obtain the final partition during the *map bundles to regions* phase.

The following sections will describe these processing blocks in detail.

[1] If outside geometry is removed, inlet and outlet surface tiles can easily be identified by thresholding the angle between flow direction and surface normal. This is also illustrated by Figure 3.2, where inlets and outlets correspond to the non-blue regions in the top right image.

3.4.1 *Seeding*

A streamline is the simulated path of a massless particle through a flow-field and therefore a curve in 3D. To perform this simulation an initial starting point for this particle needs to be defined. These points are called *seed points* and the process of choosing them is referred to as *seeding*.

In usual visualization applications the goal for seeding is to capture the flow with as few expressive streamlines as possible. Therefore great effort has been put into designing sophisticated seeding algorithms. McLoughlin et al. provides an excellent overview [22].

For this application however the requirement is to cover the 3D dataset densely, without leaving regions uncovered by streamlines. On the other hand, generating too many streamlines increases the quality of the result, but at the cost of higher computational effort, especially during streamline bundling. Another important point is that for car cooling applications, the regions near the inlets of the dataset are of special importance. This is because the cooling components are placed there and the velocities are usually high and the flow turbulent. Two simple seeding strategies for generating feasible results are *random seeding* and *interface seeding*.

Random seeding simply picks random points from all available grid points.

Interface seeding picks points only at the inlets and outlets of the dataset. This aims to capture the regions near these interfaces for the mentioned reasons. For identifying grid points at inlets and outlets, the angle between velocity vector and surface normal vector is thresholded. At inlets and outlets, the angle is high, whereas at other surface areas the air flows along the vehicle hull, hence the angle is small. Of all identified surface points the required number of points is randomly chosen.

A good strategy is to apply hybrid seeding, which combines the results of both strategies. Interface seeding ensures that regions near the interfaces are covered very well, whereas random seeding ensures adequate coverage of other areas.

Figure 3.3 shows the result of hybrid seeding as well as the subresults of random and interface seeding for the Toyota dataset.

3.4.2 *Stream tracing*

Stream tracing is the process of generating streamlines from seed points. This is achieved by integration of the displacement in the vector field. The displacement is given by [22]

$$d\mathbf{x} = \mathbf{v}\, dt. \tag{3.1}$$

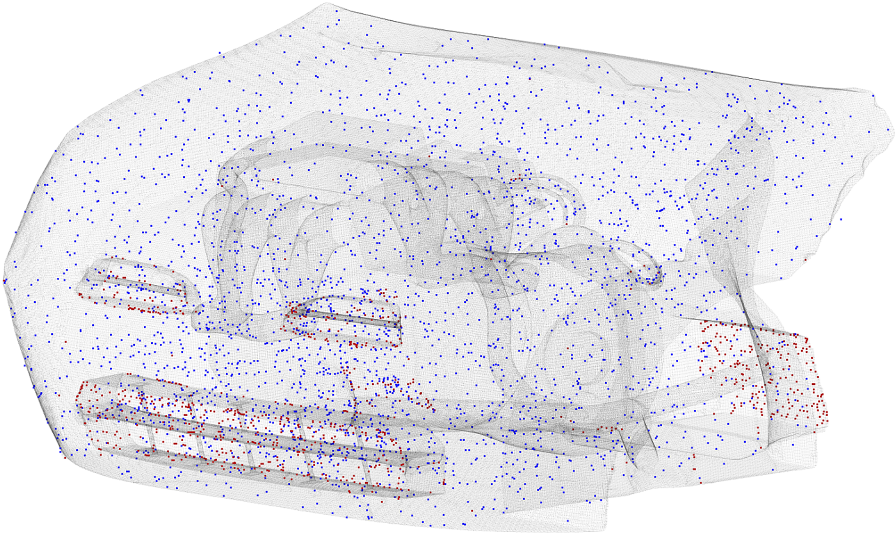


Figure 3.3: Result of hybrid seeding for the Toyota dataset. The blue points are derived by random seeding whereas the red points are derived by interface seeding.

The position $\mathbf{x}$ after time t of a particle is given by

$$\mathbf{x}(t, \mathbf{x}_0(d)) = \int_0^t \mathbf{v}(\tau)\, d\tau, \tag{3.2}$$

if it started at position $\mathbf{x}_0(d)$ in point d at $t = 0$.

For discrete vector fields this integral is computed numerically. In the simplest case, the particle is iteratively moved a small distance (*step size*) according to the velocity at the current point (Euler Scheme). To obtain velocity vectors for arbitrary points an interpolation scheme, e. g., trilinear interpolation, is required. Usually higher order integration methods are employed to increase accuracy (e. g., Runge-Kutta methods).

From an information processing standpoint, dense stream tracing can be seen as a kind of subsampling, as the streamlines preserve the essential features of the vector field around them.

Figure 3.4 shows a typical result of dense stream tracing. The tubes around the streamlines are added for illustrational purposes. Notice the subsampling characteristic: In this example, the dataset consists of 2970 streamlines connecting 500 503 points. The input dataset – the preprocessed Toyota dataset– contained 7.88 million points. By improving the seeding strategy the number of streamlines could be further reduced, as currently many regions are sampled too densely. See Section 6.1 for improvement ideas.

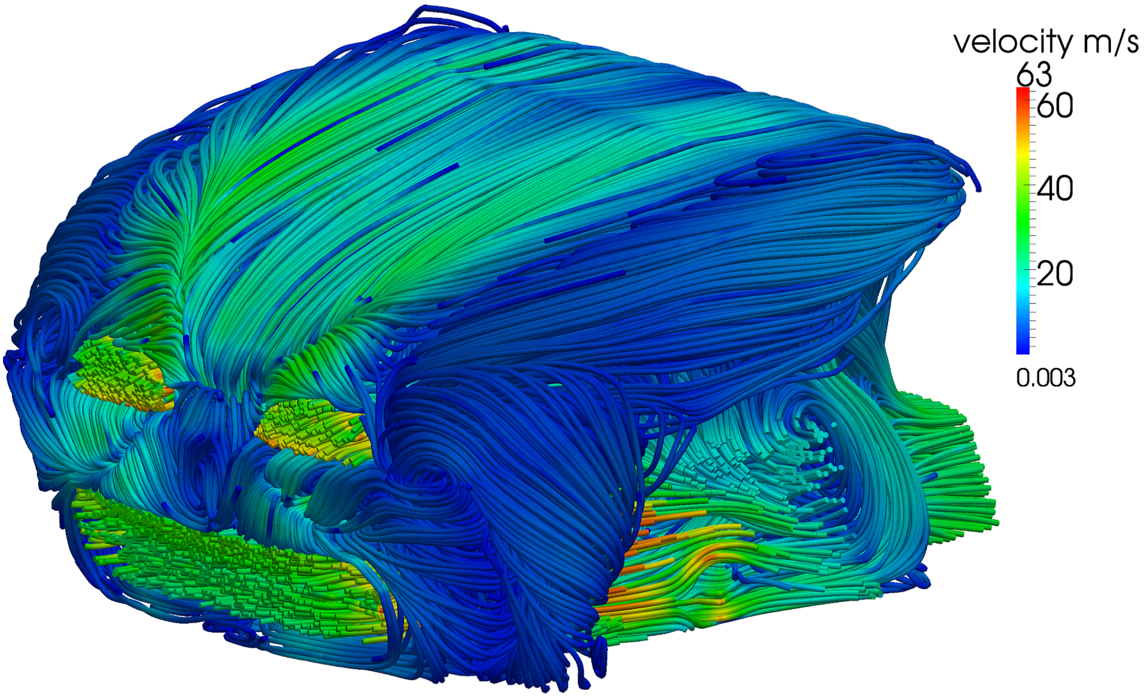


Figure 3.4: Result of dense stream tracing of the Toyota dataset at 190 km/h. Each streamline is represented by one tube. The color encodes velocity magnitude.

3.4.3 *Streamline bundling*

Looking at Figure 3.4, identifying clusters with similar flow can be described as a bundling problem. The goal is to *find bundles of streamline segments which are parallel and close to each other*.

A useful analogy for this problem is the placement of cable ties. What is a good method to organize a dense pack of cables using cable ties?

3.4.3.1 *Utility of streamline distance measures*

For every problem, there is one solution which is simple, neat, and wrong.
– H. L. Mencken

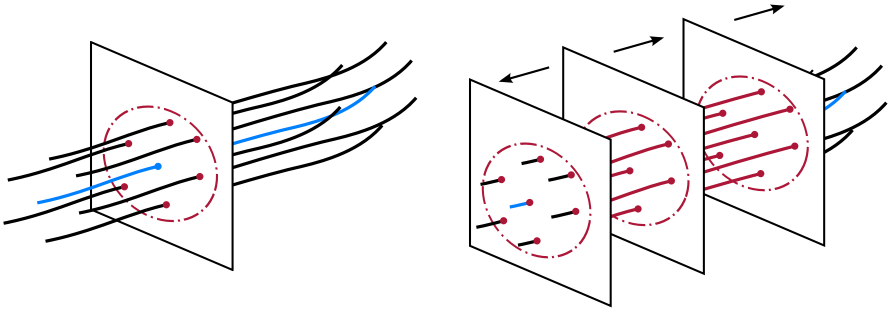
Streamline clustering from medical applications as described in Section 2.2 is hard to apply to this problem, as bundles of streamline segments are sought after, instead of bundles of whole streamlines.

The similarity measures for whole streamlines could be applied to streamline segments too, but there is no simple way of aligning these segments. Imagine two streamlines which start at the same inlet, curve around different sides of the motor and meet again at an outlet. Segments of both streamlines can probably be put into the same streamline segment bundle at the outlet, but how to derive the segments (defined by starting point, end point) that should be compared using the distance measure? The only similarity in this case is spatial closeness. The distances along the curves and even the distances between curve points however, can differ considerably.

Therefore streamline distance measures and clustering methods can only be applied if the segments are already known. This, however, is part of the solution. Notice that the streamline distance measures could be used to evaluate the quality of the resulting bundles. This idea was not adopted for this thesis, as evaluation was performed at a later stage, directly on the mesh partition.

3.4.3.2 *Streamline bundling idea*

Instead of overcoming the difficulties of adapting existing, distance based solutions to the clustering of streamline segments, this work suggests a geometrically driven method for streamline segment bundling. The main idea is illustrated in Figure 3.5.



(a) Initial slice.

(b) Incremental slices.

Figure 3.5: The basic principle of streamline bundling. Starting from a *prototype streamline* with known position and orientation (blue), an orthogonal sweep plane (black) is intersected with all streamlines, the *initial slice* (a). The sweep plane is moved along the prototype performing repeated *incremental slices* at each point of the prototype (b). Finally, streamlines that are similar to the prototype in all slices, the *mates*, are grouped together to form a *bundle*.

Repeated intersections of a sweep plane and nearby streamlines are performed along a *prototype streamline*. The result of one intersection of the sweep plane with nearby streamlines is called a *slice*. Starting with an initial slice, locally similar streamlines (*mates*) will be intersected by several slices in both directions along the prototype. Streamlines which are part of all of these slices are part of the resulting *streamline bundle*.

3.4.3.3 *Streamline bundling problems*

The idea for streamline bundling presented in the previous section is simple, but additional effort is required to solve the following detail issues:

PROTOTYPE SELECTION AND PROCESSING: How are prototype streamlines and starting points selected and in which order should they be processed?

STREAMLINE SIMILARITY: What are "nearby streamlines" and how is the similarity of sliced streamlines defined?

STOPPING CRITERION: Moving the sweep plane farther along the prototype creates longer bundles, but fewer mates will be part of this bundle. When should the sweep plane stop?

BUNDLE COLLISION STRATEGY: How to proceed if identified bundles collide with each other, i. e., if the current bundle grows into an existing one?

The following sections will discuss these issues and their proposed solutions.

3.4.3.4 *Prototype selection and processing*

Notice that prototype selection is merely the selection of discrete points along the streamlines. Stream tracing ensures that no point is used for more than one streamline; therefore selecting a point also selects a unique streamline.

An ideal prototype selection scheme would select prototype points that lead to large bundles first, followed by prototypes which lead to smaller and smaller bundles until the whole dataset is covered by bundles. Several schemes to approximate this ideal goal are possible.

A simple but effective scheme is *subsampling*. For every n^{th} point on every streamline an initial slice is performed (e. g., $n = 20$). The number of intersected, similar streamlines is used as an estimator for the number of mates in the final bundle. Each subsampled point has therefore an assigned *rating*, with all other points having a rating of zero. The scheme is then to start streamline bundling from these points in order of their rating, starting with the highest rated points until all points are processed. This scheme requires only simple bundle collision strategies, as existing bundles are expected to be larger than new ones.

A related scheme is *spherical voting*. The principle is similar to the one above, namely assigning a rating to each point and processing the points in decreasing rating order. The difference lies in the generation of these ratings. Instead of explicitly computing a full slice for some points, every point is visited once and votes for nearby, similar points, thereby increasing their rating. After voting, every point has a rating according to the density of similar points around it. "Nearby points" are determined by a sphere of pre-specified radius. Points lying within this sphere can be efficiently computed using kd-trees. The similarity of points is specified by the values attached to these points, e. g., the velocities. The advantage of this method is that every point gets a rating, instead of only a few subsampled ones. The approximation of good prototypes by point density however, does not produce satisfactory results. This selection strategy is therefore only included for completeness.

Another possible scheme is *random selection* of prototype points. The major advantage of this approach is its speed. The major disadvantage is that it is far from the ideal prototype selection scheme outlined at the beginning of this section, i. e., selecting large bundles first. To compensate for this problem, a good bundle collision strategy, which allows replacing small, existing bundles by new, larger bundles, is required.

3.4.3.5 *Similarity of sliced streamlines*

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.
– John von Neumann

Consider a single slicing plane starting at a *slicing point*[2] and being perpendicular to the streamline at this point. The plane intersects many streamlines, close and distant, from the slicing point. The question is: Which of the intersected streamlines are *similar* to the streamline at the slicing point? Similar streamlines can be part of this slice and therefore of the bundle. Expressed in a different way, a streamline bundle consists of all streamline segments that are similar to the prototype in *every single slice* of the bundle.

Each streamline consists of a chain of points with attached position, velocity, and pressure. These values where derived from the initial vector field via stream tracing. In general, the intersection of a streamline with the slicing plane does not coincide with any of the discrete points that span the streamline (Figure 3.6). After the position of the intersection point ($\bar{\mathbf{x}}$) is computed by intersecting the line with the slicing plane, the other values at the intersection $\bar{\mathbf{v}}$ and $\bar{p}$ are interpolated from the neighbor points d_i and d_{i+1}.

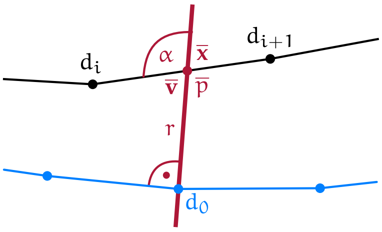


Figure 3.6: Available values for streamline similarity. Starting from a slicing point on a slicing line (blue) a perpendicular plane (thick maroon line) is used to slice nearby streamlines. A nearby streamline (black) intersects the plane between point d_i and d_{i+1}. All streamline points have a position $\mathbf{x}(d)$, a velocity $\mathbf{v}(d)$ and a pressure $p(d)$. At the intersection point these values are interpolated ($\bar{\mathbf{x}}$, $\bar{\mathbf{v}}$, and $\bar{p}$).

Using these values, the streamline bundling algorithm has to decide which intersected streamlines are similar to the slicing streamline and should therefore remain in the bundle. The analyzed criteria

[2] The slicing point for the initial slice is the prototype point. For incremental slices the slicing points lies on the same streamline, but moves farther and farther away from the prototype point.

in this work are only based on values lying on the slicing plane (no lookahead):

SPATIAL PROXIMITY (RADIUS): The distance r of the intersection point $\bar{\mathbf{x}}$ from the center $\mathbf{x}(d_0)$:

$$r = \|\mathbf{x}(d_0) - \bar{\mathbf{x}}\|. \tag{3.3}$$

VELOCITY ANGULAR SIMILARITY: The angle between the center velocity $\mathbf{v}(d_0)$ and the intersection velocity $\bar{\mathbf{v}}$ (approximates the intersection angle α):

$$s_\alpha = \arccos\left(\frac{\mathbf{v}(d_0) \cdot \bar{\mathbf{v}}}{\|\mathbf{v}(d_0)\|\,\|\bar{\mathbf{v}}\|}\right). \tag{3.4}$$

RELATIVE VELOCITY MAGNITUDE SIMILARITY: The relative similarity of the magnitudes of $\mathbf{v}(d_0)$ and $\bar{\mathbf{v}}$. For easier thresholding, the following measure is useful:

$$s_{\|\mathbf{v}\|} = \min\left\{\frac{\|\mathbf{v}(d_0)\|}{\|\bar{\mathbf{v}}\|}, \frac{\|\bar{\mathbf{v}}\|}{\|\mathbf{v}(d_0)\|}\right\}. \tag{3.5}$$

RELATIVE PRESSURE SIMILARITY: Measures how similar the pressures $p(d_0)$ and $\bar{p}$ are in relation to each other. Analogous to Equation 3.5 a good measure is

$$s_p = \min\left\{\frac{p(d_0)}{\bar{p}}, \frac{\bar{p}}{p(d_0)}\right\}. \tag{3.6}$$

Several combinations of these criteria are thinkable. A very robust possibility is using only the spatial proximity measure. Additional criteria can be added depending on the application.

The thresholds for the criteria might also change from slice to slice. For example, increasing the radius between slices allows cone-shaped bundles. A more detailed discussion on the utilized combinations of settings can be found in Chapter 5.

3.4.3.6 *Slicing stopping criterion*

Starting at the initial slice, the streamline bundle is expanded simultaneously into both directions along the prototype streamline by repeated slicing. The resulting bundle consists only of streamlines which are similar to the prototype in *all* slices, the mates. This implies that the bundle potentially loses, but never gains mates during expansion. Without a stopping criterion the resulting bundle would contain only the prototype streamline. Therefore stopping criteria are required. The following stopping criteria where implemented:

LOST MATE RATIO: Stop a direction if the ratio of mates in the initial slice to mates in the current slice drops below a given threshold.

VELOCITY MAGNITUDE CHANGE: Stop if the magnitude of the slice point velocity changed too much since the initial slice.

DIRECTION CHANGE: Stop if the direction of the prototype changed too much since the initial slice. This criterion prevents (or allows) bent bundles and the amount of allowable bending.

STREAMLINE COUNT MINIMUM: Stop if the number of streamlines in the bundle (prototype plus mates) drops below a fixed number, e. g., three. This ensures that no bundle is dropped because it becomes too small.

Different combinations of these stopping criteria allow to tailor the bundling result to specific applications. For the resistance graph application, short and straight bundles with many streamlines are desirable. This can be achieved by strict thresholds on the lost mate ratio and the directional change.

For visualization purposes, long and possibly bent bundles can be desirable. In this case loose thresholds on the lost mate ratio and very loose thresholds on directional changes lead to favorable results.

3.4.3.7 *Bundle collision strategy*

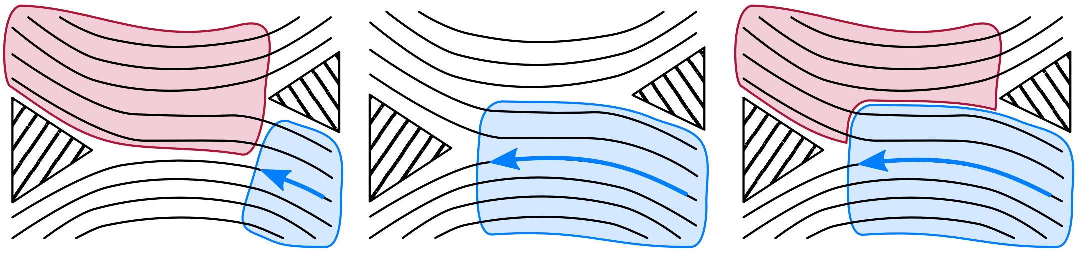
The previous sections described all criteria for generating the bundle for a single prototype streamline and starting point. To cover the whole streamline dataset with bundles, this step has to be repeated multiple times. It will therefore occur that the current bundle expands into an existing bundle. The following three strategies for these bundle collisions where investigated, to match the prototype selection schemes described in Section 3.4.3.4:

KEEP EXISTING BUNDLES: This strategy always keeps the existing bundle and is especially useful if the probability of the existing bundle being "better" than the current bundle is high. It is therefore well suited if intelligent prototype selection with prototype ratings is used.

REMOVE POOR EXISTING BUNDLES: This simple strategy removes the existing bundle completely, if the current bundle is "better". The definition of "better" can for example mean "more mates" or "more covered volume".

OVERWRITE POOR EXISTING BUNDLES: The idea for this strategy is to overwrite overlapping parts of the existing bundle, if the current bundle is better.

Figure 3.7 shows examples of these three strategies.



(a) Keep existing bundles.

(b) Remove existing bundles if worse.

(c) Overwrite existing bundles if worse.

Figure 3.7: Different bundle collision strategies. The maroon bundle already exists, whereas the blue bundle is currently expanding from right to left. In (a) the existing bundle is always kept, the expansion therefore stops. In (b), the existing bundle is completely removed if the currently expanding one is better. In (c) the expanding bundle overwrites any overlapping parts of bundles that are worse.

3.4.4 *Mapping bundles to regions*

In the final step of this partitioning algorithm, the dense streamline bundles are mapped back to the 3D mesh. The simplest method is nearest neighbor mapping, where each cell is mapped to its spatially closest bundle. If the detected bundles do not cover the dataset densely, this approach leads to bad assignments of cells that are too far away from all bundles.

An alternative approach is to map each cell to its closest bundle, but only up to a maximum distance. All cells that are too far away from any bundle are assigned to a separate region. These dead zones usually contain only a few streamlines curling at low speeds. Clustering these zones into disconnected regions has only little influence on the quality of the result. Problems occur if the bundle coverage of the dataset is poor or the chosen maximum distance is too small. Then individual, small dead zones merge together and form one large dead zone, which is undesirable.

3.4.5 *Streamline bundling recap*

After the theory behind streamline bundling is discussed, there are still open questions about the best configurations of the presented options for different tasks. Good configurations of the presented building blocks are not obvious from a theoretic point of view and will therefore be described in Chapter 5.

3.5 MAPPING REGIONS TO A GRAPH

Nature laughs at the difficulties of integration.
– Pierre-Simon de Laplace

The previous stages resulted in a partition of the 3D mesh. This partition is now mapped to a flow graph $\mathtt{G}(\mathtt{V} = \{\mathtt{v}_i\}, \mathtt{E} = \{\mathtt{e}_{i,j}\})$ in the obvious way: The regions $\mathcal{R}_i$ consisting of cells $\{c_k : c_k \in \mathcal{R}_i\}$ are represented by vertices v_i. Neighboring regions are connected by a set of edges $e_{i,j}$. The region $\mathcal{R}_i$ that led to a specific vertex v_i will be called *vertex region*. The common surface between two regions $\mathcal{R}_i$ and $\mathcal{R}_j$ will be called *edge interface* $\mathcal{I}_{i,j}$ and leads to the edge $e_{i,j}$. The edge interface consists of individual cell faces $\{f_{r,s} : c_r \in \mathcal{R}_i, c_s \in \mathcal{R}_j, c_r \text{ and } c_s \text{ are neighbors}\}$.

Center positions, pressures, velocities, and volumes of the above 3D mesh entities ($\mathcal{R}_i$, and c_k) will be identified as the functions $\mathbf{x}(.)$, $p(.)$, $\mathbf{v}(.)$, and $V(.)$ respectively. The areas, velocities, and normal vectors of the above 2D mesh entries ($f_{r,s}$, and $\mathcal{I}_{i,j}$) will be described as the functions $A(.)$, $\mathbf{v}(.)$. and $\mathbf{n}(.)$ respectively.[3]

Accordingly, the values associated with graph entities (v_i, and $e_{i,j}$) will be identified as: $\mu(.)$ for means, $\sigma(.)$ for standard deviations, $N(.)$ for counts, $V(.)$ for volumes, $A(.)$ for areas, $F(.)$ for scalar volume flows, and $\mathbf{F}(.)$ for vector volume flows.

Read this paragraph carefully, the orientation of edge values can be confused very easily.

The volume flow over a border face between two cells i and j is given by $A\mathbf{n} \cdot \mathbf{v}$, where $A \geqslant 0$ is the area of the face, $\mathbf{v}$ is its velocity, and $\mathbf{n}$ is its unit normal vector. *The orientation of* $\mathbf{n}$ *determines if the flow is computed from* i *to* j *or from* j *to* i. Therefore the *projected area* $\mathbf{A}_{\text{proj}} = A\mathbf{n}$ in flow graphs of this thesis is an *oriented measure*, depending on the edge direction. (An alternative approach would be to fix the orientation of the projected area and switch the orientation of the velocity according to the edge direction.)

In the following formulas, each edge exists twice; once for each direction. The values of both directions differ only in the sign for oriented values ($\mathbf{A}_{\text{proj}}$, F, and $\mathbf{F}$), and are equal for all other, non-oriented values. The solution to avoid double computation will be treated in Section 4.4.

The unit normal vectors will be represented by $\mathbf{e}_0 = (1,0,0)$, $\mathbf{e}_1 = (0,1,0)$, and $\mathbf{e}_2 = (0,0,1)$.

There is something in statistics that makes it very similar to astrology.
– Gian-Carlo Rota

For computing the weighted means and standard deviations, the following formulas can be utilized to avoid holding all values in memory (running computation):

$$\begin{aligned}
W_0 &= 0 & W_i &= W_{i-1} + w_i && \text{(weights)}\\
\mu_0 &= 0 & \mu_i &= \mu_{i-1} + \tfrac{w_i}{W_i}(x_i - \mu_{i-1}) && \text{(means)}\\
Q_0 &= 0 & Q_i &= Q_{i-1} + w_i(x_i - \mu_{i-1})(x_i - \mu_i) && \text{(helper sums)}\\
\sigma_0 &= 0 & \sigma_i &= \sqrt{\tfrac{Q_i}{W_i}} && \text{(standard deviations)}
\end{aligned} \tag{3.7}$$

Many applications are imaginable for flow graphs. It is therefore hard to determine which data to store in vertices and regions. For

[3] The normal vectors always point outwards, from the first index to the second index, i. e., from i to j or from r to s.

this reason, all natural vertex and edge values were evaluated and associated to the graph. At the moment this includes the following values:

VERTEX SPATIAL MEANS (VECTOR): The weighed component-wise average spatial position of the vertex region. The weights are given by the cell volumes:

$$\mu_{\mathbf{x}}(v_i) = \frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\, \mathbf{x}(c_k). \tag{3.8}$$

VERTEX SPATIAL STANDARD DEV. (VECTOR): The standard deviations from the above means, weighted by cell volumes, and computed separately for each component:

$$\sigma_{\mathbf{x}}(v_i) = \sqrt{\frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\ (\mathbf{x}(c_k) - \mu_{\mathbf{x}}(v_i))^2}. \tag{3.9}$$

VERTEX PRESSURE MEAN (SCALAR): The pressure mean, weighted by cell volume, within the vertex region:

$$\mu_p(v_i) = \frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\, p(c_k). \tag{3.10}$$

VERTEX PRESSURE STANDARD DEV. (SCALAR): The standard deviation from the above mean, also weighted by cell volume:

$$\sigma_p(v_i) = \sqrt{\frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\ (p(c_k) - \mu_p(v_i))^2}. \tag{3.11}$$

VERTEX VELOCITY MEANS (VECTOR): The weighed component-wise average velocity of the vertex region. The weights are given by the cell volumes:

$$\mu_{\mathbf{v}}(v_i) = \frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\, \mathbf{v}(c_k). \tag{3.12}$$

VERTEX VELOCITY STANDARD DEV. (VECTOR): The standard deviations from the above means, weighted by cell volumes, and computed separately for each component:

$$\sigma_{\mathbf{v}}(v_i) = \sqrt{\frac{1}{V(\mathcal{R}_i)} \sum_{c_k \in \mathcal{R}_i} V(c_k)\ (\mathbf{v}(c_k) - \mu_{\mathbf{v}}(v_i))^2}. \tag{3.13}$$

VERTEX CELL COUNT (SCALAR): The number of cells within the region $\mathcal{R}_i$ that is represented by v_i:

$$N_c(v_i) = |\mathcal{R}_i|\,. \tag{3.14}$$

VERTEX VOLUME (SCALAR): The volume of the region represented by this vertex:

$$V(v_i) = V(\mathcal{R}_i) = \sum_{c_k \in \mathcal{R}_i} V(c_k). \tag{3.15}$$

VERTEX FACE COUNT (SCALAR): The number of cell faces at the surface of the vertex region:

$$N_f(v_i) = |\{f_{r,s} : r = i\}| = \sum_j N_f(e_{i,j}). \tag{3.16}$$

EDGE PROJECTED AREAS (VECTOR): The projected areas of the interfaces between the regions $\mathcal{R}_i$ and $\mathcal{R}_j$ to the xy-, xz-, and yz-planes:

$$\mathbf{A}_{\text{proj}}(e_{i,j}) = -\mathbf{A}_{\text{proj}}(e_{j,i}) = \sum_{f_{r,s} \in \mathcal{I}_{i,j}} A(f_{r,s})\, \mathbf{n}(f_{r,s}). \tag{3.17}$$

EDGE VELOCITY MEANS (VECTOR): The weighted component-wise average velocity of the edge interface. The weights are given by the cell face areas:

$$\boldsymbol{\mu}_{\mathbf{v}}(e_{i,j}) = \boldsymbol{\mu}_{\mathbf{v}}(e_{j,i}) = \frac{1}{A(\mathcal{I}_{i,j})} \sum_{f_{r,s} \in \mathcal{I}_{i,j}} A(f_{r,s})\, \mathbf{v}(f_{r,s}). \tag{3.18}$$

EDGE VOLUME FLOWS, EXACT (VECTOR): The sum of volume flows over each individual face of the edge interface, i. e., the volume flows are computed first for each projected face and accumulated afterwards. Gives volume flows separated in x-, y- and z-direction:

$$\mathbf{F}_{\text{exact}}(e_{i,j}) = -\mathbf{F}_{\text{exact}}(e_{j,i}) = \sum_{f_{r,s} \in \mathcal{I}_{i,j}} A(f_{r,s})\, \mathbf{n}(f_{r,s}) \cdot \mathbf{v}(f_{r,s}). \tag{3.19}$$

EDGE VOLUME FLOWS, EXACT SUM (SCALAR): The sum of the entries of the above vector:

$$F_{\text{exact}}(e_{i,j}) = -F_{\text{exact}}(e_{j,i}) = \sum_{t=1}^{3} \mathbf{e}_t \cdot \mathbf{F}_{\text{exact}}(e_{i,j}). \tag{3.20}$$

EDGE VOLUME FLOWS, APPROX. (VECTOR): The element-wise multiplication of "edge projected areas" and "edge velocity mean" gives an approximate volume flow:

$$\mathbf{F}_{\text{approx}}(e_{i,j}) = -\mathbf{F}_{\text{approx}}(e_{j,i}) = \sum_{t=1}^{3} \mathbf{e}_t \, \big(\mathbf{e}_t \cdot \mathbf{A}_{\text{proj}}(e_{i,j})\big) \, (\mathbf{e}_t \cdot \mu_{\mathbf{v}}(e_{i,j})). \tag{3.21}$$

EDGE VOLUME FLOWS, APPROX. SUM (SCALAR): The sum of the entries of the above vector gives the approximate scalar volume flow rate:

$$F_{\text{approx}}(e_{i,j}) = -F_{\text{approx}}(e_{j,i}) = \sum_{t=1}^{3} \mathbf{e}_t \cdot \mathbf{F}_{\text{approx}}(e_{i,j}). \tag{3.22}$$

EDGE FACE COUNT (SCALAR): The number of faces in the edge interface:

$$N_f(e_{i,j}) = N_f(e_{j,i}) = \left|\mathcal{I}_{i,j}\right|. \tag{3.23}$$

VERTEX IN FLOW (SCALAR): The total volume flow into this vertex; determined by integration over the surface. This is the sum of the negative "edge volume flow exact sum"-values (i. e., the incoming edges) multiplied by -1.

$$F_{\text{in}}(v_i) = -\sum_j \min\left(0; F_{\text{exact}}(e_{j,i})\right). \tag{3.24}$$

VERTEX OUT FLOW (SCALAR): The total volume flow out of this vertex; determined by integration over the surface. This is equal to the sum of positive "edge volume flow exact sum"-values (i. e., outgoing edges).

$$F_{\text{out}}(v_i) = \sum_j \max\left(0; F_{\text{exact}}(e_{i,j})\right) \tag{3.25}$$

3.6 MAPPING BUNDLES TO A GRAPH

An alternative to calculating the exact flow graph from the partitioned 3D mesh is to calculate an approximate flow graph directly from the bundles. This approach is faster and uses less memory than mapping the bundles to the 3D mesh. As in the previous scheme, each bundle results in one vertex. However, in contrast to the previous scheme, an edge is created between two vertices, if the respective bundles are directly connected by a streamline.

The calculation of vertex and edge data in this case is a rough estimation based on streamline counts and velocities. Therefore this approach is applicable only to visualization tasks.

Even for visualization tasks of dense bundles however, the exact, mesh-based calculation is advantageous, because the computational overhead is not very large.

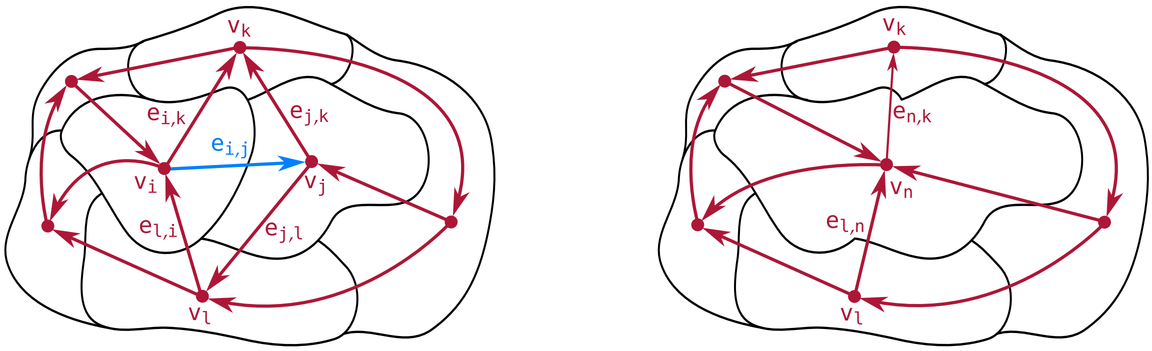
The only use case for this algorithmic step is therefore pure visualization of *sparse bundles*. If only a few large bundles and their interconnections are sought-after, a complete mapping to the 3D mesh is not advantageous. In this case the flow graph has to be derived directly from the sparse bundles.

Mapping from bundles to a graph has been implemented, but was neither optimized nor utilized. It will therefore be excluded from further discussion.

3.7 GRAPH COLLAPSE

There is no way to directly influence the final number of vertices in the flow graph during streamline bundling, because the number of bundles cannot be enforced.

To reduce the number of vertices a series of edge collapses can be performed. The graph operation *edge collapse* is the removal of an edge from a graph by combining the two vertices that the edge connects. Therefore, each edge collapse reduces the vertex count by one. Figure 3.8 illustrates this operation for flow graphs.



(a) Before edge collapse.

(b) After edge collapse.

Figure 3.8: The edge collapse operation for a flow graph. The maroon network depicts the graph, whereas the black outlines show the underlying regions. The left figure shows the graph before collapsing the blue edge $e_{i,j}$. The right figure shows the graph after collapsing it.

Three steps have to be performed during edge collapse:

1. Merge the edges of shared neighbors. In Figure 3.8a there are two shared neighbors: v_k and v_l. The according edges need to be merged: $e_{l,i}$ with $e_{j,l}$, and $e_{i,k}$ with $e_{j,k}$. During merging,

the edge directions have to be taken into account for oriented values like volumetric flow rates.

2. Merge vertices $\mathtt{v_i}$ and $\mathtt{v_j}$ into $\mathtt{v_n}$. This involves combining the vertex values described in Section 3.5 and associating the combined values to $\mathtt{v_n}$. Most of the combinations are straight forward, with the exception of the standard deviations. The following formula was used to combine standard deviations (for vectors it was applied component-wise):

$$\sigma(\mathtt{v_n}) = \sigma(\mathtt{v_i} \cup \mathtt{v_j}) = \sqrt{\frac{V(\mathtt{v_i})}{V(\mathtt{v_n})}\,\sigma^2(\mathtt{v_i}) + \frac{V(\mathtt{v_j})}{V(\mathtt{v_n})}\,\sigma^2(\mathtt{v_j}) + \frac{V(\mathtt{v_i})\,V(\mathtt{v_j})}{V(\mathtt{v_n})}\,(\mu(\mathtt{v_i}) - \mu(\mathtt{v_j}))^2},$$
$$\text{where} \quad V(\mathtt{v_n}) = V(\mathtt{v_i}) + V(\mathtt{v_j}). \tag{3.26}$$

3. Remove the edge $\mathtt{e_{i,j}}$ and its associated data.

3.8 FLOW GRAPH ERROR MEASURES

For selecting good candidate edges to collapse, a global measure $E(\mathtt{G})$ is required. Then, by repeatedly collapsing the best current candidate edge $\mathtt{e_{i,j}}$ according to that error measure, the targeted number of vertices is reached. The best candidate edge has the lowest increase $\Delta E(\mathtt{e_{i,j}})$ of the global error measure. All vertex and edge values described in Section 3.5 can be utilized to construct this error measure. The error measure used within this thesis is the total Sum of Squared Errors (SSE) for velocities, $E_{\mathbf{v}}(\mathtt{G})$. It is derived as follows.

The velocity SSE vector for one region (vertex) is defined es

$$\mathbf{E}_{\mathbf{v}}(\mathtt{v_i}) = \sum_{c_k \in \mathcal{R}_i} V(c_k)\;(\mathbf{v}(c_k) - \boldsymbol{\mu}_{\mathbf{v}}(\mathtt{v_i}))^2. \tag{3.27}$$

The direct computation requires the individual cell values $V(c_k)$ and $\mathbf{v}(c_k)$ and therefore does not allow deriving this measure for a graph vertex. However, it can easily be computed from the velocity standard deviations and region volumes that are stored with each vertex (see Section 3.5) as

$$\mathbf{E}_{\mathbf{v}}(\mathtt{v_i}) = \boldsymbol{\sigma}^2_{\mathbf{v}}(\mathtt{v_i})\,V(\mathtt{v_i}), \tag{3.28}$$

The calculation of the norm was postponed as long as possible to minimize directional errors.

which makes the scalar, total error of the whole graph

$$E_{\mathbf{v}}(\mathtt{G}) = \sum_{\mathtt{v_i} \in \mathtt{V}} \|\mathbf{E}_{\mathbf{v}}(\mathtt{v_i})\|. \tag{3.29}$$

Finally, the cost of collapsing edge $e_{i,j}$ according to this error measure is

$$\Delta E_{\mathbf{v}}(e_{i,j}) = \|\mathbf{E}_{\mathbf{v}}(v_n) - \mathbf{E}_{\mathbf{v}}(v_i) - \mathbf{E}_{\mathbf{v}}(v_j)\|, \tag{3.30}$$

where v_n is the merged vertex.

4 IMPLEMENTATION

In theory, there is no difference between theory and practice. But, in practice, there is.
– Jan L.A. van de Snepscheut

The realization of the described algorithms is greatly aided by using the VTK framework [30]. ParaView is a visualization application based on VTK [25]. They will be shortly introduced in the next section. The following sections will describe the implementation of the algorithmic building blocks in detail. Finally, the last section of this chapter will briefly address the user interface.

4.1 VTK AND PARAVIEW

VTK is an open source visualization framework for C++ and includes a variety of algorithms and data structures to process and visualize geometric data. It was introduced by Schroeder et al. [27].

All VTK algorithms and operations are encapsulated in classes with the common interface `vtkFilter`. This approach allows the arbitrary combination of compatible filters to *filter chains*. Each black block in the processing overview of the previous chapter (Figure 3.1) is implemented by a combination of these filters. Some of them came with VTK, the others were implemented as part of this thesis. VTK also supports the visualization of its data structures via OpenGL. Chaining instances of `vtkFilter` and visualizing the output requires only a few lines of code.

3D CFD mesh data is stored within an instance of `vtkUnstructuredGrid`, surface and streamline data is stored within an instance of `vtkPolyData`, and graph data is stored within an instance of `vtkGraph`.

The former two data structures consist in essence of points, cells, and associated data, which are all stored in arrays. Removing points or cells from these structures would require the removal of elements from the middle of arrays. It is therefore a slow operation and not supported by public VTK interfaces. Hence, VTK algorithms usually do not alter the input, but create a new output from scratch. The `vtkFilter` interface enforces this convention by passing the empty output data structure together with the inputs. This separation of input and output encourages clean algorithms, but consumes more memory.

Consider for example the preprocessing stage. Fixing the cell normals is possible in-place, because no cells or points are removed. For removing the outside cells however, a separate output dataset is required. This almost doubles the required memory during this step to about 2 GB for the Toyota dataset.

VTK has no direct user interface, but is accessed through C++ or any of the other wrapper languages, like Java and Tcl. Sometimes constant changes to the filter chain or the visualization settings are required, for example during exploration of new datasets or debugging of new filter outputs. The repeated compilation and execution for these small adaptations quickly becomes a cumbersome task.

All dataset figures of this thesis where rendered using ParaView.

In these cases *ParaView* is a helpful additional tool. ParaView is a Graphical User Interface (GUI) front end for VTK, allows the creation and adaptation of filter chains, and therefore makes most of the functionality of VTK easily accessible.

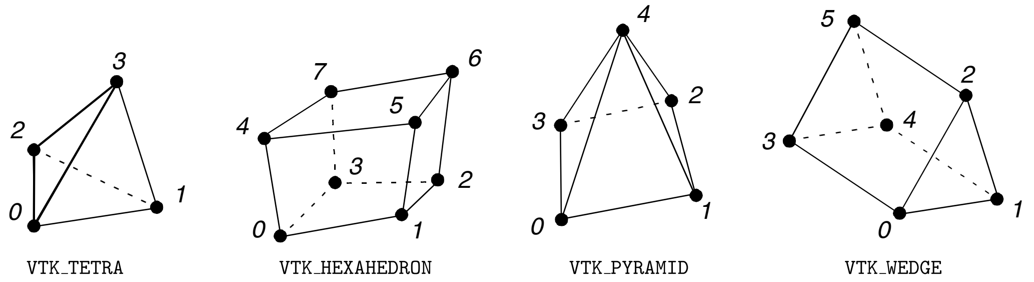
4.2 PREPROCESSING

Basic preprocessing steps can be performed directly in ParaView. This includes loading data with different formats, cleaning unnecessary or redundant CFD values and removing uninteresting parts of the dataset, like 2D surface structures.

More complex preprocessing steps, as the removal of outside geometry, are implemented as custom subclasses of `vtkFilter`.

4.2.1 *Rectifying inconsistent surface normals*

Figure 4.1 shows the four common cell types supported in this thesis, together with their VTK names and point orders.



VTK_TETRA VTK_HEXAHEDRON VTK_PYRAMID VTK_WEDGE

Figure 4.1: The common 3D cell types used for CFD simulations and their VTK names and point orders. (Images taken from the document "VTK File Formats" [31].)

After importing the Toyota dataset into VTK, the surface normals of most cells are flipped. This means that the points of most cells are labeled in the wrong order. Consider for example, the tetrahedron in Figure 4.1. After the points 0, 1, and 2 are fixed, point 3 is expected to lie *above* the triangle when using the right hand rule. If point 3 is *below* the triangle however, all surface normals point inwards instead of outwards. Determining the source of the problem is out of scope for this thesis, but it must lie either in EnSight export or in VTK import.

All tetrahedra, hexahedra, and pyramids show this problematic behavior. The surface normals of the wedges are correct, because they have different point orders in EnSight and VTK which compensates the problem.[1]

If a dataset is imported from EnSight and processed using VTK or ParaView, newly created cells have the *correct* point order. This leads to an even worse situation, where cells having either correct or incorrect surface normals are randomly mixed. Correct surface normals are not only required for visualization, but more importantly for integrating flows between regions in the "Map Region to Graph" module.

Therefore the correctness of surface normals has to be ensured for all cells during the preprocessing phase. This is done by checking simple geometric constraints and switching point orders if necessary.

The cells in VTK are stored within a linearized cell array that arranges the point Identifiers (IDs) for each cell in succeeding order. Fixing surface normals can therefore be achieved by swapping the point IDs of the affected cells. This does not affect the point coordinates or any other cell. The applied checks and changes are as follows (see also Figure 4.1):[2]

VTK_TETRA: If $(\widehat{0;1;2})$ points away from 3, swap the point IDs of 0 and 1.

VTK_HEXAHEDRON: If $(\widehat{0;1;2})$ points away from 4, swap the point IDs of 0 and 2, and the point IDs of 4 and 6.)

VTK_PYRAMID: If $(\widehat{0;1;2})$ points away from 4, swap the point IDs of 0 and 2.

VTK_WEDGE: If $(\widehat{0;1;2})$ points *into* 3, swap the point IDs of 0 and 1, and the point IDs of 3 and 4.

4.2.2 *Removing undesired outside geometry*

The problem descriptions and basic solution ideas behind these algorithms were treated in Section 3.3.1. The following sections cover the implementation in greater detail.

4.2.2.1 *Region growing*

To separate the outside cells from the inside cells by region growing, two problems need to be solved. The first one is finding appropriate

[1] In VTK, the normal vector (using the right hand rule) of the wedge base points *away* from the wedge. In EnSight this normal points *into* the wedge. Notice that the VTK wedge is inconsistently defined to the other VTK cell types, which always have the base normal pointing inwards. This has led to confusion in several cases in VTK. For example the "Normal Glyphs Filter" of ParaView (Version 3.8.0) produces incorrect results for wedges.

[2] $(\widehat{0;1;2})$ denotes the normal vector of the plane which is defined by the three given points using the right hand rule. $(\overrightarrow{0;1})$ denotes the vector pointing from the first point to the second point. To determine if vectors point into or away from each other, the scalar product is used.

seed cells to start the growing process, and the second one is finding a suitable stopping criterion. Without a stopping criterion, the outside region would grow also to inside cells, because outside and inside cells are connected through holes in the vehicle hull, e. g., at air intakes.

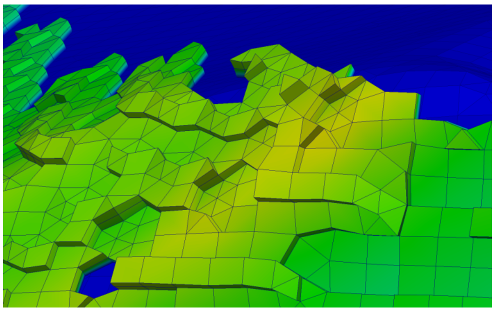


Figure 4.2: Closeup view of undesired geometry. The surface structure of undesired geometry is rough. Some of the cells are very exposed and only have one neighbor. The idea is to start region growing from these exposed cells.

Figure 4.2 shows a closeup view of typical outside cells. Notice the exposed cells at the surface near the top. These can be used to start region growing. The first step in the region growing approach is therefore to identify surface cells. This is achieved using the existing `vtkDataSetSurfaceFilter`. Afterwards, exposed surface cells having only a few neighbors are identified as seed cells.

Finding a suitable stopping criterion is more difficult. One possibility is to grow only into cells with weak support, i. e., cells with relatively few neighbors. Inside cells have strong support, because of their compact arrangement, whereas outside cells have weak support, as they are thin and contain holes. This approach leads to practical results, but misses small islands of outside cells that support each other.

Another idea is to grow only into cells that are within a certain distance from the seed cells. For datasets with rough geometry at the inlets however, like the Toyota dataset, this grows into inside cells near the inlet too.

The best results can be achieved by combining both stopping criteria. The problem that some rough inside cells near the inlets are labeled as outside cells is reduced, but remains.

The best thresholds for the Toyota dataset are as follows:

- Surface seed cells are chosen to be pyramids and tetrahedra with only one neighbor, or hexahedra and wedges with up to two neighbors.

- Starting from the seed cells, the outside region grows only into pyramids and tetrahedra with up to two neighbors, or into hexahedra and wedges with up to three neighbors.

- The maximum distance of any cell from a seed cell (*seed distance*) is a parameter of the filter.

4.2.2.2 *Depth probing*

Identifying the outside cells by depth probing is based on two observations:

1. Outside geometry is thin in at least one direction whereas inside geometry is compact, and therefore thick, in every direction.
2. Outside geometry consists mainly of flat wedges and hexahedra, i. e., it is mainly composed of stacks of hexahedra and wedges.

These observations are utilized by performing a *depth probe* starting at every surface cell of the whole dataset. Figure 4.3 demonstrates this for all cells within a zoomed cut through the dataset.

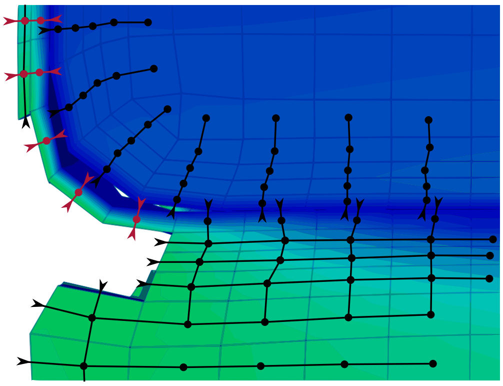


Figure 4.3: Illustration of depth probing within a zoomed cut at the transition from outside cells to inside cells. Depth probes are performed from all surface cells. Short stacks of cells are classified as outside (maroon chains).

Starting from all faces at the surface, iteratively jump to the neighbor at the opposite face of this cell, creating a stack of neighboring cells. Stop if there is no cell at the opposite face, or a maximum depth (i. e., number of jumps) is reached. In the former case, the whole stack is considered "thin" and therefore classified as being outside (maroon chains).

Notice that the notion of an opposite face is only defined for all faces of hexahedra and the triangular faces of wedges. All other cell types have several opposite face candidates. One approach to solve this problem would be to create cell trees instead of cell chains. In the current problem however, almost all cells are either hexahedra or wedges stacked with their triangular faces. The jump to the opposite face is therefore defined as follows:

1. If the cell is a hexahedron, jump to the obvious opposite face.

2. If the cell is a wedge and the incoming face is a triangle, jump to the opposite triangle.
3. In all other cases:
   a) If there is no neighbor cell at any opposite face, report the end of chain, i. e., a thin stack was found.
   b) If there is exactly one neighbor cell at any opposite face, jump to this candidate.
   c) Otherwise stop as if the maximum depth is reached.

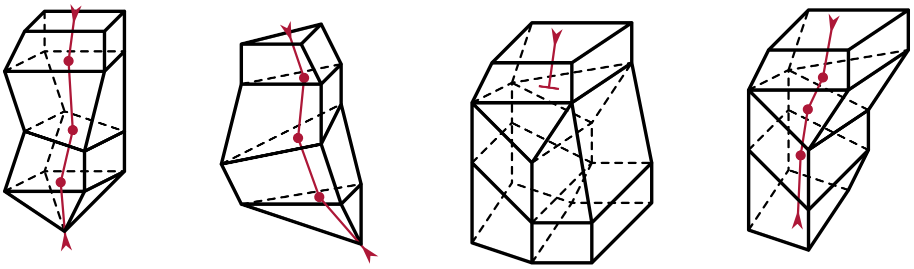


Figure 4.4: Example of some depth probes with different cell types. Leftmost figure: Traversing a stack of hexahedra with a final pyramid. Second from left: Same situation for wedges and a tetrahedron. Second from right: Stop depth probe, if multiple neighbors are found at opposite faces. Rightmost figure: Continue depth probe, if only one opposite face connects a neighbor.

Figure 4.4 illustrates and explains the different jump types. All stacks are traversed from top to bottom. The leftmost figure shows how a stack of hexahedra is traversed. The bottom cell is a pyramid, but has no other neighbors. The traversal is therefore valid, and the stack will be classified as thin/outside. The second figure from the left demonstrates the same situation for a stack of wedges and a final tetrahedron. The figure next to it illustrates that depth probing is stopped, if multiple jumps are possible. The classification in this case is thick/inside. The rightmost example shows the same situation, but with only one possible jump at the wedge, which is then performed. This results in a thin/outside stack again.

A cell may be visited multiple times, if it is close to multiple faces on the dataset surface. It is classified thin/outside, if it is part of *any* thin stack. Therefore, after all surface tiles have been processed every cell which is part of any thin stack is classified as being outside.

Results for both approaches to remove undesired geometry can be found in Section 5.1.

4.3 PARTITIONING

The theory behind the following sections can be found in Section 3.4. The partitioning algorithm of McKenzie et al. was implemented as described in their paper. From the implementation perspective, the k-means algorithm differs only in the details from McKenzie et al.'s algorithm. Both implementations allow choosing from different distance measures, e. g., the distance function introduced by Du and Wang.

The implementation of the streamline bundling algorithm is subject of the following sections.

4.3.1 *Seeding*

Random seeding is straightforward, uniform, random sampling from all available grid points. The first idea to improve random seeding was to assign different selection probabilities to the grid points according to their velocity. Increasing the selection probability of grid points with low velocity was expected to increase the overall coverage of the dataset. In practice however, the results of later stages did not improve because streamlines starting at low velocity grid points are quickly dragged into higher velocity areas.

For *interface seeding*, the first step is to find grid points which are part of in- or outlets. This is accomplished by first selecting all grid points at the dataset surface using the built-in VTK algorithm `vtkDataSetSurfaceFilter`. Afterwards, the surface normals of these surface grid points are computed by averaging over the normals of the neighboring surfaces. This functionality also comes with VTK (`vtkPolyDataNormals`).

At this point the surface normal vector $\mathbf{n}(d_i)$ and velocity $\mathbf{v}(d_i)$ of all surface grid points d_i are known. Grid points at in- or outlets have a small angle between these two vectors, whereas for other grid points this angle is close to 90°. Interface grid points can therefore be identified by thresholding this angle:

$$|\cos(\alpha)| = \left| \frac{\mathbf{v}(d_i)}{\|\mathbf{v}(d_i)\|} \cdot \mathbf{n}(d_i) \right| > \cos(\alpha_{\text{thres}}) \qquad (4.1)$$

From all surface grid points which satisfy this condition, the desired number is randomly chosen.

The number of required seed points depends on the flow field complexity of the dataset, i. e., to cover many small and detailed areas of the dataset, many streamlines are required. The number and placement of seed points also depends on the application. If only the main flow paths are sought-after – e. g., for a visualization application – fewer streamlines starting only from inlets and outlets will suffice.

4.3.2 *Stream tracing*

Stream tracing is a standard method and is included in VTK. `vtkStreamTracer` features many parameters to control the integration scheme, the integration direction, the step size, and the termination conditions.

The stream tracing phase consumes a minor part of the total computation time. Therefore the decisions for these algorithmic parameters can be made to improve streamline quality instead of stream tracing speed. Using the best available integrator type (Runge-Kutta 4.5) and integrating into both directions is therefore an easy decision.

The Runge-Kutta integrators can automatically adapt the step size during integration. The step size parameters allow defining the minimum, the maximum, and the initial step size. These lengths can be defined either absolute or relative to the current cell size.

The termination settings can and should be used to reduce curling streamlines in dead zones. The best way to reduce these curls is to set a high terminal speed. This instructs `vtkStreamTracer` to stop stream tracing if the particle velocity drops below the given threshold.

4.3.3 *Streamline bundling*

Science is what we understand well enough to explain to a computer, Art is all the rest.
– Donald E. Knuth

The main idea, the basic algorithm, and the theoretic formulas for bundling streamline segments were already described in Section 3.4.3. This section will connect these theoretic pieces using pseudo code and accompanying text.

4.3.3.1 *Data types*

Before discussing the algorithm, the following basic data types are described: *slice*, *segment*, *bundle*, and *bundle store*.

SLICE A slice is the result of the intersection of a plane with a streamline field (Function `sliceAt`). It contains a list of intersections and the following associated data:

1. A reference to the intersected streamline (e. g., cell ID of the streamline).
2. A reference to the streamline point index before the intersection, e. g., 5 if the streamline is intersected between its 5th and 6th point.
3. The coordinates of the exact intersection.
4. The distance of the exact intersection to the slice point (the *radius*).
5. Interpolated values of all associated data (e. g., velocity, pressure. . .) for the exact intersection.

As slices are just temporary structures, only insertion and iteration are required. The list of intersections is therefore best within a dictionary sorted by radius, because this is the usual iteration order.

segment A segment specifies a part of a streamline and consists of a reference to the streamline (cell ID), and the segment index interval. The interval is specified by the first and last point index on this streamline.

A point index on a streamline is not *the point ID. It is an index into the cell array of this streamline. The cell array stores the point IDs.*

bundle A bundle is a list of segments. As the segments are usually accessed by streamline cell ID, a dictionary is a good choice. The dictionary must support multiple entries with the same key, because different segments of the same streamline might be part of the same bundle.

bundle store A bundle store is a collection of bundles. In the current implementation, it also maintains an inverse map from point IDs to their respective bundles.

4.3.3.2 *Global bundling algorithm*

The global bundling loop is simple. It traces bundles and adds them to the bundle store until a stopping criterion is met (Algorithm 4.1).

Algorithm 4.1: Streamline bundling overview

```
  Input: Streamlines
  Output: Streamline segment bundles
1 perform prototype selection preprocessing;
2 repeat
3     protoPoint ← select next prototype starting point;
4     bundle ← traceBundle(protoPoint);
5     insert bundle into bundleStore according to collision strategy;
6 until any bundling stopping criterion met;
```

line 1: Currently two prototype selection schemes are in use: subsampling and random selection. The former one requires expensive preprocessing, whereas the latter one requires none at all. For a theoretic discussion of prototype selection schemes see Section 3.4.3.4.

Implementation of these schemes is straightforward, but there is one implementation detail: For the initial slices during prototype subsampling, it is advisable to use stricter similarity thresholds than for the following incremental slices. This ensures that the assigned rating of an initial slice is a good estimator of the bundle size not only for this one slice, but is also maintained after a few incremental steps. Without this “reliable start”, a subsampling slice could contain many streamlines at the very edge of similarity. After a few incremental slices, most of these would be dropped from the bundle because of dissimilarity. In short, the initial rating would be less expressive.

LINES 3 AND 4: After a prototype point has been selected, it is used as a starting point for tracing a bundle. Function `traceBundle` is treated in the following section.

LINE 5: Finally the bundle is inserted into the bundle store according to the bundle collision strategy. All three collision strategies described in Section 3.4.3.7 are supported.

LINE 6: No stopping criterion is required for bundling with *subsampling prototype selection*. Streamline bundling is simply stopped after all prototype candidates are processed. In order to support *random prototype selection*, there are three options to stop the global bundling loop:

1. When a *maximum number of bundles* is reached.
2. When a *minimum bundle coverage*, i. e., the percentage of streamline points which are bundled, is reached.
3. By *user intervention* (manual stop).

4.3.3.3 *Bundle tracing function*

We think in generalities, but we live in details.
– Alfred North Whitehead

Bundling from a single prototype point is done by performing an initial slice and moving it along the prototype streamline as described in Section 3.4.3.2. Figure 4.5 illustrates the actual process in 2D.

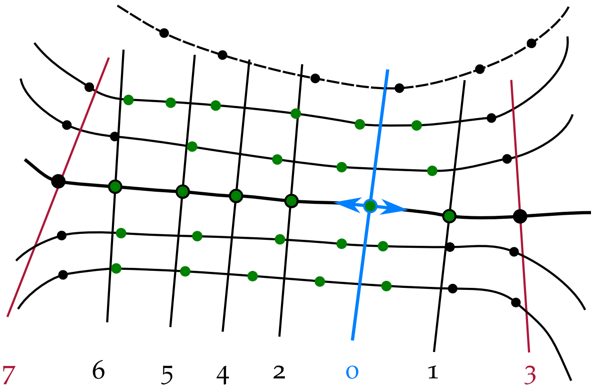


Figure 4.5: Two-dimensional view of the formation of a bundle showing *streamlines* (curved lines) with *prototype streamline* (thick curved line), *streamline points* (dots), *initial slice* (vertical blue line), *incremental slices* (other vertical lines), *stop slices* (maroon lines), the incremental slice order (numbers), a *lost mate* (dashed line), and the points in the final bundle (green).

At the initial prototype point (blue circle) an initial slice (blue line) is created. In the subsequent steps, incremental slices are performed along the prototype streamline. The incremental slices are carried out at the prototype streamline points, and initially alternate into both directions. The expansion of the bundle proceeds until a stopping criterion for each direction is met.

Alternating the initial directions aims to improve the bundles in case of prototype selection by subsampling. Growing into only one direction from "good" initial slices leads to smaller bundles than growing simultaneously into both directions. After the initial slice was performed (0), the next incremental slice is at its right (1) and loses one mate at the top. The next incremental slice (2) is immediately left of the initial slice and preserves all mates. The third incremental slice (3) would lose many non-parallel mates. It is therefore not incorporated into the bundle and bundling is stopped in this direction. The following three slices (4, 5, and 6) are performed into the only remaining direction until finally the seventh slice stops the bundle tracing. All green points are part of the fresh bundle.

Function `traceBundle` lists the bundle tracing algorithm.

```
Function traceBundle(protoPoint)
    Input: protoPoint
    Output: bundle
1   set initial slice radius in sliceParameters;
2   initialSlice ← sliceAt(protoPoint, sliceParameters);
3   bundle ← createInitialBundle(initialSlice);
4   adapt slice radius in sliceParameters according to
    initialSlice;
5   repeat
6   |   oldBundle ← bundle;
7   |   pick next slicePoint;
8   |   slice ← sliceAt(slicePoint, sliceParameters);
9   |   bundle ← mergeBundleWithSlice(bundle, slice);
10  |   if any slicing stopping criterion is met by bundle then
11  |   |   bundle ← oldBundle;
12  |   |   stop iteration into current direction;
13  |   end
14  until iteration stopped for both directions;
15  clear bundle if it does not meet the minimum requirements;
16  return bundle;
```

LINE 1: The slice parameters contain the similarity thresholds. For the initial slice, a maximum bundle radius is employed. Using no bundle radius (i. e., complete slice through the whole dataset) is also supported, but slow for large datasets.

LINE 2 AND 3: After performing the initial slice (see Section 4.3.3.4), the initial bundle is constructed from it (see Section 4.3.3.5).

LINE 4: For incremental slices, the slice radius is given in relation to the actual radius of the initial slice. The ratio is a parameter.

Notice that the actual radius of the initial slice is usually smaller than the maximum initial bundle radius. Another parameter controls the cone-shape of the final bundle, e. g., a value of 2.0 allows the bundle radius to increase up to twice the radius of the initial slice.

LINE 6: Incremental slicing starts by backing up the current bundle to support rollback functionality.

LINE 7: Afterwards, the next slice point is picked as explained in Figure 4.5 and its accompanying text.

LINES 8 AND 9: After the incremental slice has been performed (see Section 4.3.3.4), it is merged with the current bundle (see Section 4.3.3.6).

LINES 10 TO 12: If the resulting bundle meets any slicing stopping criterion, expansion in the current direction is stopped and the merge is rolled back. Slicing stopping criteria are discussed in Section 3.4.3.6

LINE 14: The bundle tracing ends if expansion is stopped in both directions.

LINE 15: Finally, if the resulting bundle does not meet minimum requirements, the bundle is cleared and an empty bundle is returned. Currently implemented parameters are:

- The minimum number of slices the bundle consists of.
- The minimum length of the bundle in world coordinates, measured along the prototype streamline.
- The minimum number of streamline segments in the bundle.

4.3.3.4 *Slice creation function*

If you had done something twice, you are likely to do it again.
– The Unix Programming Environment, by Brian Kernighan and Bob Pike

A slice is created by intersecting a plane, with dense streamlines. The plane is given by a point and its normal vector. Function `sliceAt` shows this process in detail.

LINE 1: Most of the time, a slice is only required to contain intersections up to the given radius from the slice point. Therefore only a subset of all streamlines segments needs to be considered for intersection. A kd-tree over all streamline points is constructed in order to quickly find all points within the given radius. These points are then mapped to all streamline segments that lie within the radius.

LINE 2: The slice plane through the given slice point d_i is uniquely defined by the slice point position $\mathbf{x}(d_i)$ and its normal vector $\mathbf{v}(d_i)/\|\mathbf{v}(d_i)\|$.

Function sliceAt(point, sliceParameters)

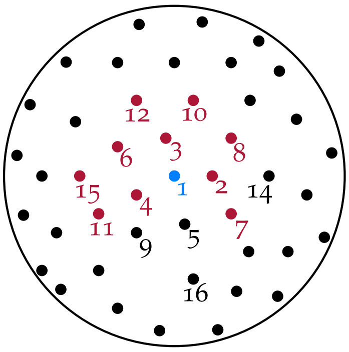
```
  Input: point, sliceParameters
  Output: slice
1 closeSegments ← find streamline segments within given radius;
2 determine slicePlane from point;
3 foreach segment in closeSegments do
4     intersections ← intersect segment with slicePlane;
5     foreach intersection in intersections do
6         append intersection and interpolation data to slice;
7     end
8 end
9 return slice;
```

lines 3 and 4: Each of the close segments is then intersected with the slice plane, by iterating over the individual line pieces of the segment. This can result into multiple intersections of the same streamline, if the streamline segment is curled.

lines 5 to 7: All found intersections are stored within the slice, together with the interpolated values at the intersection points.

4.3.3.5 *Initial bundle creation function*

During initial bundle creation, the entries of the initial slice are processed with increasing distance from the prototype point. The image of Figure 4.6 (left) shows this principle.



mate	total	ratio
4	5	0.8
7	9	0.78
10	13	0.77
10	14	0.71
11	16	0.69

Figure 4.6: Depiction of initial bundle creation (left). The circle shows the 2D projection of an *initial slice* with its *initial slice radius* (black circle). The intersected streamlines are represented by points which are numbered in processing order. *Slice point* (blue), *mate points* (maroon), and *non-mate points* (black) are also highlighted. The table at the right demonstrates the evolution of the *slice mate ratio* ("purity") of the bundle. Bundle creation stops after point 16, when the mate ratio drops below 0.7.

Starting with the prototype point 1 (blue), the sliced points are added to the bundle if they are similar (mates), and ignored if not.

Counters keep track of the number of mates (maroon) and the number of total entries. Bundle expansion stops early, if the ratio "number of mates to number of total entries" falls below a pre-specified threshold. This *slice mate ratio* allows to specify the initial bundle purity. Lowering the mate ratio enables non-circular bundles, but at the cost of introducing noisy bundles.

The table in Figure 4.6 (right) shows how the slice mate ratio develops for one specific example. Table entries are only given if an additional non-mate was found (i. e., if the ratio decreases). The processing is stopped if the mate ratio drops below 0.7. In the end, the initial bundle contains the blue and all maroon segments.

Function `createInitialBundle` explains the process using pseudo code. Implementation details are given in the following code descriptions.

```
Function createInitialBundle(initialSlice)
   Input: initialSlice
   Output: bundle
1  totalCnt ← 0;
2  mateCnt ← 0;
3  foreach entry in initialSlice ordered by radius do
4      isIntersecting ← does entry intersect an existing bundle?;
5      isSimilar ← is entry similar to slice point (initial settings)?;
6      isMate ← isSimilar and not(isIntersecting);
7      if isMate then
8          insert line segment of entry to bundle;
9          mateCnt ← mateCnt + 1;
10     end
11     totalCnt ← totalCnt + 1;
12     if $\frac{mateCnt}{totalCnt}$ < minimum initial mate ratio then
13         Break;
14     end
15 end
16 return bundle;
```

line 4: An initial slice entry might intersect segments of an existing bundle. These segments are never considered mates for the initial slice, regardless of the collision strategy. The intersecting segments are identified by querying the bundle store.

line 5: Similarity measures were discussed in Section 3.4.3.5. In the current implementation a threshold parameter for each of the discussed similarity measures is provided. The final similarity decision is a *logical and* of the individual similarity threshold decisions.

LINES 7 TO 10: As bundle tracing does not introduce new points into the dataset, only points lying *completely within two slices* are part of the final bundle. For example, if a streamline is intersected between the point at index 7 and the point at index 8, the interval is $(8, 7)$ and has length -1. If then, at an incremental slice, the same streamline is intersected between index 10 and index 11, the end-index of this interval will be shifted to be 10, making the interval $(8, 10)$.

LINES 12 TO 14: Application of the minimum initial mate ratio (purity) as discussed immediately before this listing.

4.3.3.6 *Bundle-slice merging function*

The merging of slices into bundles follows the same structure as the initial bundle creation described in the previous section. The differences are highlighted in Function `mergeBundleWithSlice` and the following descriptions.

```
Function mergeBundleWithSlice(oldBundle, slice)
  Input: slice
  Input: oldBundle
  Output: newBundle
1  mateCnt ← 0;
2  totalCnt ← 0;
3  foreach entry in slice ordered by radius do
4      breakBecauseIntersection ← handle Intersections;
5      isSimilar ← Is entry similar to slice point?;
6      isMate ← isSimilar and not(breakBecauseIntersection);
7      if isMate then
8          if line of entry exists in oldBundle then
9              interval ← find closest interval on line;
10             if distance(entry, interval) < maximum line segment
               distance then
11                 enlarge interval to include entry;
12                 insert segment for interval into newBundle;
13                 mateCnt ← mateCnt + 1;
14             end
15         end
16     end
17     totalCnt ← totalCnt + 1;
18     if $\frac{mateCnt}{totalCnt}$ < minimum incremental slice mate ratio then
19         Break;
20     end
21 end
22 return newBundle;
```

LINE 4: The check for intersections with existing bundles is different from initial bundle creation, because the intersections are handled according to the collision strategy (see Section 3.4.3.7):

KEEP EXISTING BUNDLES: Set `breakBecauseIntersection` to true, if a collision was found.

REMOVE EXISTING BUNDLES IF WORSE: Remove the existing bundle, if it has less mates then the currently traced bundle. Set `breakBecauseIntersection` to true, if a collision with an existing, but better bundle was found.

OVERWRITE EXISTING BUNDLE IF WORSE: Set `breakBecauseIntersection` to true, if a collision with an existing, but better bundle was found. Otherwise set it to false. Overwriting is performed automatically during insertion into the bundle store.

LINE 8: If a slice entry is found to be a mate, a check if the streamline is already a mate in the existing bundle is performed. (A mate must be similar in *all* slices).

LINES 9 AND 10: For helix-shaped streamlines multiple intervals of the same streamline can be segments in the same bundle. Therefore different intervals for the same streamline need to be maintained. If there are too many points along the streamline between the closest interval and the current intersection, the entry is not considered a part of an existing interval. Hence, it is not included in the bundle.

LINES 11 AND 12: Once a valid line and interval is found, the interval is extended to include the current entry, and the extended segment is inserted into the new bundle.

LINES 18 TO 20: The purity of the individual bundle slices can be controlled – similar to the initial slice – using the *incremental slice mate ratio*.

4.3.4 *Mapping bundles to regions*

The theory for this section was treated in Section 3.4.4. The implementation is presented in Algorithm 4.2.

LINES 1 TO 10: The assignment of each cell to its closest streamline bundle is straightforward using `vtkKdTreePointLocator`, `vtkCellCenters`, and standard VTK data structures.

LINE 5: The threshold distance is a parameter of the algorithm. High values assign all cells to their closest bundles.

Algorithm 4.2: mapBundlesToRegion

```
   Input: bundles
   Input: mesh
   Output: meshPartition
1  build kd-tree from all bundled points in bundles;
2  foreach cell in mesh do
3      bundlePoint ← find closest point to cell center in bundles by
       utilizing kd-tree;
4      bundle ← bundle of bundlePoint;
5      if distance(cell center, bundlePoint) < threshold then
6          mark cell as belonging to bundle in meshPartition;
7      else
8          mark cell as not belonging to any bundle;
9      end
10 end
11 foreach cell in mesh do
12     if cell does not belong to any region in meshPartition then
13         grow a new region from cell;
14         mark all cells of the new region in meshPartition;
15     end
16 end
17 return meshPartition;
```

LINES 11 TO 16: If cells remain unassigned, simple region growing is performed until all cells are assigned to a region. Notice that only the first visited cell of each connected, unclustered region triggers a region growing operation. After this operation all other connected cells belong to the new region and do not start further region growing.

4.4 MAPPING REGIONS TO A GRAPH

To compute the vertex and edge values described in Section 3.5, some implementation problems have to be solved. Basically the algorithm needs to iterate over all cells and update the according vertex data. In addition it needs to find interfaces to neighboring cells and update the according edge data (*summation phase*).[3]

The final result should be a directed graph, with edge directions matching the net flow rates between neighboring regions. Notice that the individual interfaces between two regions can have different net

[3] Usually numerical issues need to be addressed if many small values are accumulated into a grand sum. This is even more true for sums of squared values which are required for running standard deviation computations. The results of the "naive" summation where therefore compared to results of summations within binary trees. No differences where found, because the accumulated values are all within the same, small magnitude.

flow directions. Therefore the edge direction is unknown until the summation of all interface flow rates has been finished. In addition, the naive approach would count each interface flow twice, once from cell c_i to its neighbor c_j and once from c_j to its neighbor c_i. To solve these problems the following approach was chosen:

- Each region is strictly linked to one fixed vertex by using ordered IDs.
- During the summation phase, a *undirected graph* is used. All edges are *implicitly oriented* from higher to lower vertex ID. This solves the edge orientation problem.
- During the summation phase, all interfaces from regions with lower IDs to regions with higher IDs are ignored. This solves the double counting problem.
- After the summation phase, the undirected graph is converted into a directed graph. This is achieved by orienting edges with *positive flow rate* from *high vertex ID* to *low vertex ID* and vice versa.

The pseudo code is presented in Algorithm 4.3. There are no additional descriptions, as the everything has been explained either by Section 3.5 or the introductory text above.

Algorithm 4.3: mapRegionsToGraph

```
Input: partitionedMesh
Output: dirGraph
for each region create an empty vertex in undirGraph;
foreach cell in partitionedMesh do
    region ← region of cell;
    vertex ← vertex of region;
    update vertex with data of cell;
    foreach neighborCell of cell do
        neighborRegion ← region of neighborCell;
        neighborVertex ← vertex of neighborRegion;
        if ID of region > ID of neighborRegion then
            face ← face between cell and neighborCell;
            create/update undirEdge between vertex and
            neighborVertex;
        end
    end
end
optionally convert undirGraph to dirGraph;
return dirGraph;
```

4.5 GRAPH COLLAPSE

Section 3.7 treats the theory of graph collapse and explains the underlying edge collapse operation. Graph collapse was implemented for *undirected graphs* in this thesis.[4] Several implementation problems arise from the way VTK stores and manages undirected graphs. Instead of describing these problems and their solutions here, the simpler implementation for *directed graphs* is listed in Algorithm 4.4. The individual steps are thoroughly explained by Figure 3.8 and its accompanying notes.

Algorithm 4.4: graphCollapse

```
Input: inGraph
Input: targetVertexCnt
Output: outGraph

outGraph ← copy of inGraph;
initialize errorList;
foreach edge in outGraph do
    errorList(edge) ← compute collapsing error of edge;
end
repeat
    edge ← edge with lowest collapsing error;
    srcVtx ← source vertex of edge;
    dstVtx ← destination vertex of edge;
    edgePairList ← edges to/from shared neighbor vertices;
    foreach edgePair in edgePairList do
        mergedEdge ← merge edgePair into new edge connecting
            srcVtx and shared neighbor;
        insert mergedEdge into outGraph and into newEdgesList;
        remove old edges in edgePair from outGraph;
    end
    remove edge from outGraph;
    merge srcVtx and dstVtx into srcVtx;
    delete (now isolated) dstVtx;
    update collapsing error in errorList for all edges in
        newEdgesList;
    clear newEdgesList;
until number of vertices in outGraph ⩽ targetVertexCnt;
return outGraph;
```

4.6 USER INTERFACE

For a list of all the ways technology has failed to improve the quality of life, please press three.
– Alice Kahn

Usually VTK filters are chained together using C++ code or any of the supported wrapper languages (e. g., Python, Tcl, and Java). Alternatively VTK filters can be made available to ParaView via plugins.

The problem with these approaches is the way VTK chains the filters by default. It basically has two modes, namely

1. Store all intermediate data to Random Access Memory (RAM), and

2. Store no intermediate data.

The length of the filter chain and size of the dataset makes both of these approaches unsuitable for this thesis. If all intermediate data of processing large datasets within long filter chains is stored to RAM, memory becomes short quickly. If no intermediate data is stored at all, even minor changes to late filters lead to the time intense recomputations of the whole chain. To circumvent these problems, *user interaction wrappers* for the filters were introduced.

4.6.1 *User interaction wrappers*

The contract for User Interface (UI) wrappers is defined in the C++ interface `IUIWrapper`. Implementations of this interface wrap chainable VTK functionality and provide a unified interface to the UI layer. Each instance usually wraps exactly one VTK filter. The interface supports to set the *mode of loading* ("From file", "From RAM", "From both", or "None") and *saving* ("To file", "To RAM", "To both", or "None"). In addition, the interface supports unified access to filter parameters, which are split into *simple parameters* and *advanced parameters*. The interface also specifies two actions, namely,

- `update`, which recomputes or loads the result of the underlying VTK filter, and

- `render`, which renders this result.

The usual use case is to define a wrapper chain, which implicitly defines a filter chain. After the user has selected the parameters for saving, loading, and processing, a call of `update` triggers the actions described in Algorithm 4.5.

This approach enables the user to tune saving and loading of intermediate results very precisely. A good default is to "save to" and "load from" file, and to recompute if any parameter has changed. Files are written to a user specified folder using fixed names for each wrapper. The UI wrappers are designed mainly for data processing.

[4] This is mainly because graph collapse was implemented before the converter from undirected to directed graphs.

Algorithm 4.5: userInterfaceUpdate

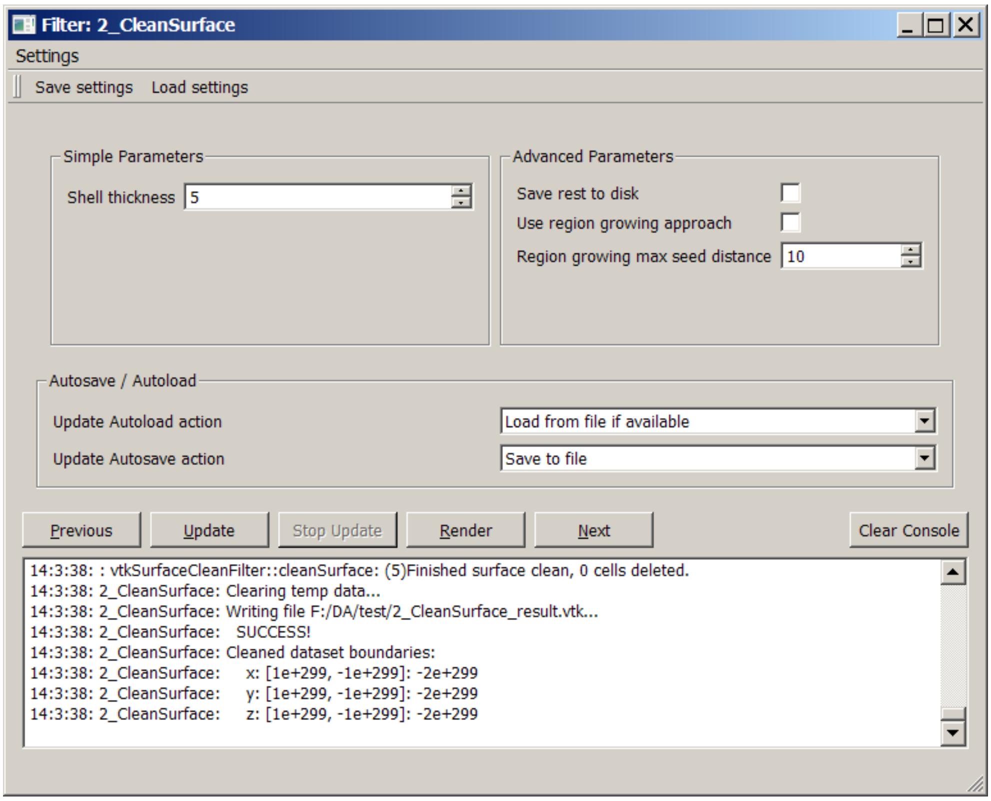
```
Output: result
if result should be loaded and is loadable then
    result ← load from file or RAM, according to load settings;
else
    inputs ← trigger update on input wrappers;
    result ← recompute this wrapper using inputs;
    save result according to save settings;
end
return result
```

The `render` action is only intended to *preview* the result. In-depth exploration of results is best accomplished by utilizing ParaView.

4.6.2 *Graphical user interface*

Read the directions and directly you will be directed in the right direction.
– Doorknob, Alice in Wonderland [4]

The front end for the user interaction wrappers was implemented using Qt [26]. Initially, the user selects one of the possible paths defined in Figure 3.1. This automatically constructs and chains the according filter wrappers. The following screens allow to jump from wrapper to wrapper using the "Previous" and "Next" buttons. Figure 4.7 shows an example screen for the surface clean wrapper. On each screen, the settings of the wrapper can be adapted, and the `update` and `render` actions can be triggered. The console at the bottom shows progress, debug, and error messages.



Shell thickness 5

Save rest to disk □

Use region growing approach □

Region growing max seed distance 10

Update Autoload action: Load from file if available

Update Autosave action: Save to file

```
14:3:38: : vtkSurfaceCleanFilter::cleanSurface: (5)Finished surface clean, 0 cells deleted.
14:3:38: 2_CleanSurface: Clearing temp data...
14:3:38: 2_CleanSurface: Writing file F:/DA/test/2_CleanSurface_result.vtk...
14:3:38: 2_CleanSurface:   SUCCESS!
14:3:38: 2_CleanSurface: Cleaned dataset boundaries:
14:3:38: 2_CleanSurface:     x: [1e+299, -1e+299]: -2e+299
14:3:38: 2_CleanSurface:     y: [1e+299, -1e+299]: -2e+299
14:3:38: 2_CleanSurface:     z: [1e+299, -1e+299]: -2e+299
```

Figure 4.7: Screenshot of the Qt-GUI with *simple parameters*, *advanced parameters*, *autosave* and *-load* parameters, *action buttons*, and *console output*.

5 RESULTS

He [Taniyama] was gifted with the special capability of making many mistakes, mostly in the right direction. I envied him for this and tried to imitate him, but found it quite difficult to make good mistakes.
– Goro Shimura
(Nova program on Fermat's Last Theorem)

This chapter starts with briefly discussing the results and limitations of the preprocessing stage described in Section 3.3.1.

Afterwards, the applicability of streamline bundling and McKenzie et al. clustering to resistance graph simulation is evaluated. This is followed by the description of an envisioned flow graph exploration tool that is based on streamline bundles. Both of these evaluations are performed on the Toyota dataset. Finally the result of streamline bundling on a centrifugal pump dataset is briefly discussed.

All computations were performed on a laptop with an Intel Core 2 Duo Central Processing Unit (CPU) with 2.4 GHz and 4 GB of RAM. All stated calculation durations apply to this setup and do not include the time for saving intermediate data to the hard disk.

5.1 PREPROCESSING RESULTS

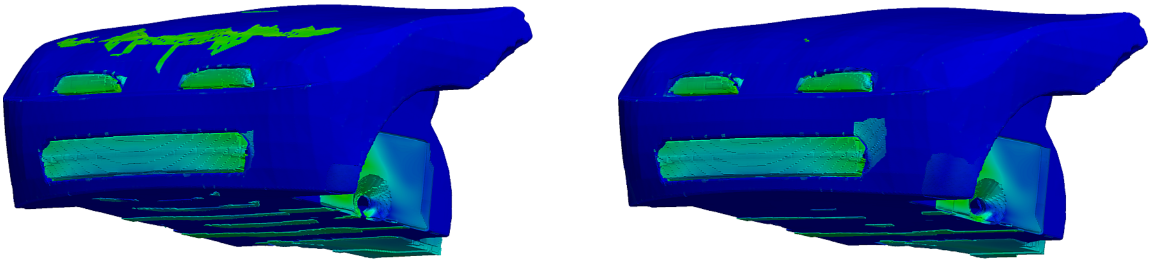
In Section 3.3.1 and Section 4.2.2 two approaches for removing undesired outside geometry where discussed: region growing and depth probing. The results for each of them will be discussed in the following sections.

5.1.1 *Region growing*

The major problem with region growing is the removal of inside cells near complex interface areas. No satisfying criterion could be derived to stop regions from growing into these areas. For the Toyota dataset all front inlets are problematic zones in this sense. The hybrid approach described in Section 4.2.2.1 leads to the results shown in Figure 5.1.

Figure 5.1a shows the result for region growing from exposed cells up to a seed distance of 50. Not all outside cells were removed, but some inside cells around the inlets already are. With a seed distance of 85, almost all outside cells on the hood are removed, although there is still outside geometry left at the vehicle bottom (Figure 5.1b). In addition, the number of removed inside cells at the problematic inlets is even higher.

The calculation durations where about 2 min for a seed distance of 50 and about 3 min for a seed distance of 85. Exact durations cannot be given, as they varied due to the unpredictable handling of memory requests by the Operating System (OS).



(a) Maximum seed distance of 50.

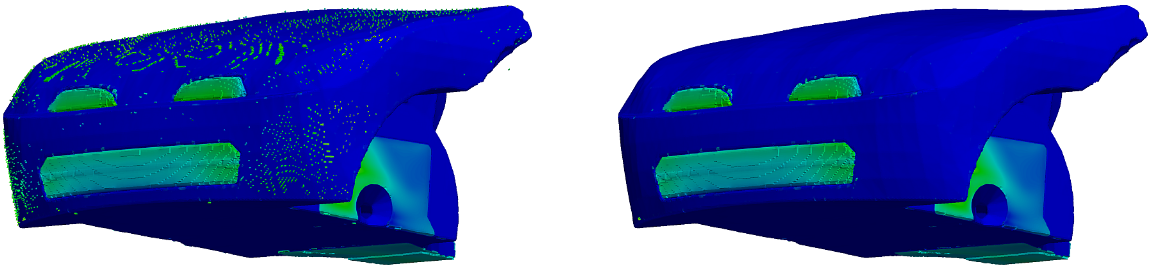
(b) Maximum seed distance of 85.

Figure 5.1: Results of the region growing approach. Figure (a), shows the result for a seed distance of 50. Not all outside cells are removed as desired (green cells on the hood), but some inside cells already are (top right corner of the bottom inlet). Figure (b) shows the same situation for a seed distance of 85. Outside geometry is largely eliminated, but the problematic areas at the inlets lack even more inside cells.

5.1.2 *Depth probing*

The second approach for removing outside geometry was depth probing as discussed in Section 3.3.1 and Section 4.2.2.2. It removes almost all outside cells for the Toyota dataset, which is illustrated in Figure 5.2.

Figure 5.2a shows the outcome of depth probing if the maximum depth is set too low, in this case 3. Stacks that are higher than maximum depth are left untouched by the algorithm. Figure 5.2b shows the outcome for the correct maximum depth for this dataset (5). The image also demonstrates that depth probing is well suited for cleaning the Toyota dataset and similar datasets.



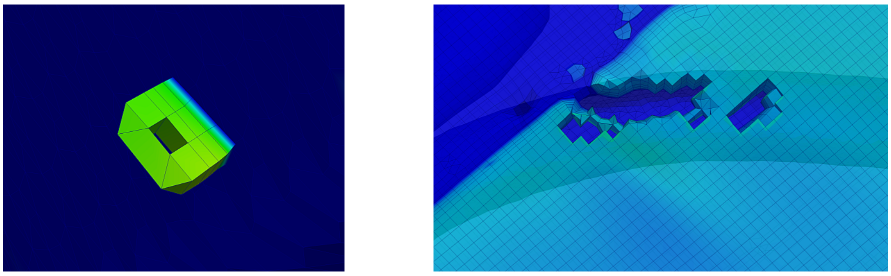
(a) Maximum depth of 3.

(b) Maximum depth of 5.

Figure 5.2: Results of the depth probing approach. Figure (a), shows the result for maximum depth of 3. Some stacks of outside cells are too high and are not removed as required (green spots on the hood). Figure (b) shows the same situation for a maximum depth of 5. The outside geometry is eliminated almost entirely without affecting any inside cells near the inlets.

Nevertheless, two problems remain with depth probing. Figure 5.3 illustrates both of them.

The first problem (Figure 5.3a) is created by groups of self supporting cells, i. e., several stacks of cells with neighboring pyramids or tetrahedrons at either end. These problems could be eliminated by



(a) Self supporting outside geometry remains.

(b) Thin inside geometry removed.

Figure 5.3: Problems of the depth probing approach.

running the depth probing algorithm twice, thus removing the cells from a perpendicular direction. However, as such islands are very rare, this was not carried out.

The second problem is more severe. Imagine inside geometry that is *thin itself*, i. e., that contains stacks of inside cells that are lower than the maximum depth. All stacks of these type are removed by the depth probing algorithm, even if this is not desired. The depth probing approach is therefore useless for these datasets.

Fortunately there is only a small region with thin inside geometry in the Toyota dataset. A pipe is running near the rear bottom of the dataset and generates thin geometry within a small area. Figure 5.3b shows how cells were erroneously removed from that area.

5.2 APPLICABILITY FOR SIMULATION

The best evaluation for the resistance graph simulation application would be direct simulation. However, as stated before, a resistance graph simulator is not available, as this thesis is a part of its preliminary work. In consequence, this section can only evaluate the fundamental properties of good flow graph partitions: spatial *compactness* and *similarity* of velocities and pressures.

Usually, these two properties contradict each other. More compact clusters are less similar and vice versa. Therefore, clustering algorithms require the possibility to trade them off against each other. For McKenzie et al.'s approach, this is accomplished through the distortion function. For streamline bundling, several of the described parameters influence this trade off.

In the next section the employed error measures are explained. The following section treats the choice of parameters for the simulation application. Afterwards, the general characteristics of McKenzie et al.'s approach and streamline bundling are illustrated with the help of a standard example. Finally, the error measures for different operating points and region counts are compared briefly.

5.2.1 *Error measures*

The following error measures are computed on whole flow graphs and measure the global error of the underlying partition. As the best tradeoff between spatial closeness, velocity similarity and pressure similarity is not known, they will be treated by three separate error measures.

The spatial, velocity and pressure errors of the flow graphs ($E_{\mathbf{x}}(G)$, $E_{\mathbf{v}}(G)$ and $E_p(G)$ respectively) are defined as global SSEs. $E_{\mathbf{v}}(G)$ was already defined in Section 3.8, in the context of choosing good candidate edges during the graph collapse stage. The spatial and pressure errors are defined analogously as

$$E_{\mathbf{x}}(G) = \sum_{v_i \in V} \|\mathbf{E}_{\mathbf{x}}(v_i)\|, \quad \text{where } \mathbf{E}_{\mathbf{x}}(v_i) = \boldsymbol{\sigma}^2_{\mathbf{x}}(v_i)\, V(v_i), \tag{5.1}$$

and

$$E_p(v_i) = \sigma^2_p(v_i)\, V(v_i). \tag{5.2}$$

5.2.2 *Parameters for streamline bundling*

One big disadvantage of streamline bundling is the large amount of required parameters, even if the parameters are quite comprehensible and allow a great level of control. Thankfully, most parameters never need to be modified, i. e., have well performing defaults. The following paragraphs provide guidelines for all available parameters.

PARAMETERS FOR SEEDING AND STREAMTRACING The simulation application requires a *complete partition* of the dataset. This can only be achieved by dense streamlines traced from dense seed points. For the Toyota dataset this leads to hybrid seeding with 3000 random points and 5 % of all interface points ($\cos(\alpha_{\text{thres}}) = 0.8$). Increasing these values will improve the result slightly, but also increases the processing time considerably.

The parameters for streamline tracing are explained in detail within the VTK documentation.[1] It is important to trace into both directions and to set a high terminal speed (e. g., 0.5 m/s) to prevent curls. The streamlines should be as long as possible (maximum propagation length: 5 m, maximum step count per line: 10 000). The required time for stream tracing can be reduced by increasing the step size parameters from the defaults (initial: 0.5 cells, minimum: 0.05 cells, maximum: 0.75 cells).

PARAMETERS FOR STREAMLINE BUNDLING The different prototype selection and bundle collision strategies were discussed in Sec-

[1] It can currently be found at `http://www.vtk.org/VTK/help/documentation.html`.

tion 3.4.3.4 and Section 3.4.3.7, respectively. To reach maximum coverage a good choice is to *overwrite existing bundles if worse*.

The target is to create compact and similar regions. These are created from short, thick bundles of parallel streamlines. The main parameter to create short, thick bundles is the "lost mate ratio" (0.75). To avoid bent bundles, the total change of the prototype direction is restricted to 15°.

The initial bundle purity ("initial slice mate ratio") is set to 0.9 and the size of the initial slice is restricted by the "initial slice radius" to 0.2 m. The maximum slice radius is set to twice the initial radius in order to prevent extremely cone-shaped bundles.

Some outliers within the individual slices can be tolerated. Therefore only the spatial proximity criterion – i. e., the radius – given in Equation 3.3 is used for choosing mates, while all other criteria are neglected.

PARAMETERS FOR THE REMAINING STEPS *All* cells need to be mapped to a region. Therefore no threshold-radius was employed and each cell was mapped to its spatially closest bundle.

The number of regions – and therefore the number of flow graph vertices – cannot be chosen in advance for the streamline bundling algorithm. Hence, graph collapse operations are required to thin the graph to the final number of vertices. The employed version of graph collapse has no parameters (see Section 3.7 and Section 4.5).

One could see this approach as a sophisticated way of initializing a simple region merging algorithm.

5.2.3 *Parameters for McKenzie et al.*

The two main parameters of McKenzie et al.'s algorithm are the *targeted number of regions* and the *number of iterations*. The former is predetermined by the setup, the choice of the latter is discussed in the Section 5.2.4.4.

Two distortion functions were selected to be included in the evaluation: McKenzie et al.'s own *velocity-only* distortion function (see Equation 2.10) and another function suggested within their paper, namely Du and Wang's distance function (see Equation 2.8). For Du and Wang's distance function the user interface provides a *velocity influence factor* between 0.0 and 1.0. Setting it to 0.5 leads to $w = \frac{1}{2L^2}$ as advocated by Du and Wang, 1.0 sets $w = 0$, and 0.0 translates to $w = \frac{5}{2L^2}$. The remaining values are linearly interpolated.

5.2.4 *Standard example results*

Nine configurations of the Toyota dataset were evaluated in this work. These configurations arise from partitioning the datasets of the three available operating points (30 km/h, 190 km/h and $v_{\max}$) into 100, 500 and 1000 regions. This section discusses the central setup of this three-times-three configuration table – 190 km/h and 500 regions – in detail.

5.2.4.1 *Naive cuboid partitioning*

By utilizing the error measures introduced in Section 5.2.1, the results of the different approaches can be compared to each other. However, they do not allow statements about the *absolute* quality of the results.

To enable absolute statements to some extent, a naive partition was created to provide baseline errors. The partition is created by splitting the dataset into cuboids of constant size, as depicted in Figure 5.4. The dataset is split into 7, 10 and 11 evenly spaced intervals in x-, y- and z-direction, respectively. After removing all empty cuboids, this results in 506 regions; close enough to 500 to qualify for comparison.

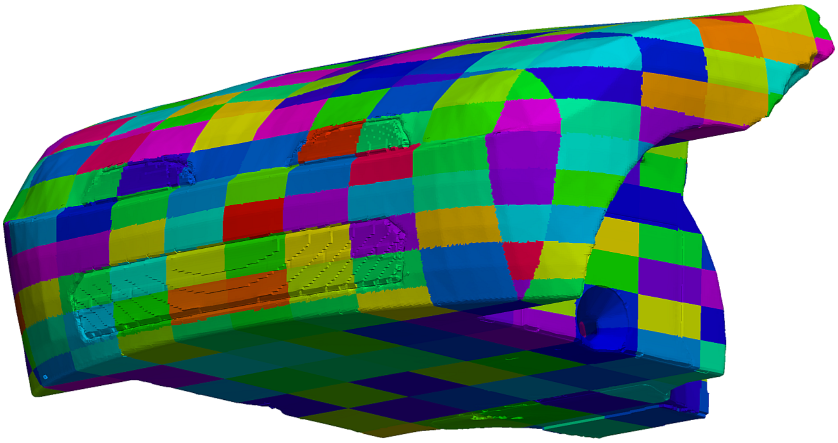


Figure 5.4: Partition of the dataset into 506 cuboids of constant size. This partition is used for comparison purposes.

Naturally, this partition leads to regions that are spatially very compact. The velocity and pressure SSEs on the other hand should offer plenty of space for improvement.

5.2.4.2 *Error evaluation for the standard example*

Table 5.1 shows the algorithms and settings for different setups. In combination with the previous discussion on parameter selection, the setups are completely defined.

Figure 5.5 shows a graphical illustration of the SSEs for all setups. The three error measures are shown in three separate groups, which are individually normalized to the SSEs of the naive cuboid partition. Table 5.2 contains the according absolute values of the individual bars.

The first observation is that no approach can match the compactness of the naive partition (maroon). This is because the naive partition has a very good spatial layout. The higher spatial errors of the

Color	Description
🟥	Naive cuboid partition
🟦	Streamline bundling with subsampled prototype selection (5%). Leads to ≈ 3100 bundles. Then graph collapse.
🟩	Streamline bundling with random prototype selection to 3100 bundles. Then graph collapse.
▨	McKenzie et al. with DuWang distortion function (velocity influence 0.0, $w = 5/L^2$). 5 Iterations.
▨	McKenzie et al. with DuWang distortion function (velocity influence 0.25, $w = 2.5/L^2$). 5 Iterations.
▨	McKenzie et al. with DuWang distortion function (velocity influence 0.5,$w = 1/L^2$). 5 Iterations.
▨	McKenzie et al. with DuWang distortion function (velocity influence 1.0, $w = 0$). 20 Iterations.
⬛	McKenzie et al. with velocity only distortion function. 20 Iterations.

Table 5.1: The setups (algorithms, settings and associated colors) for error evaluation.

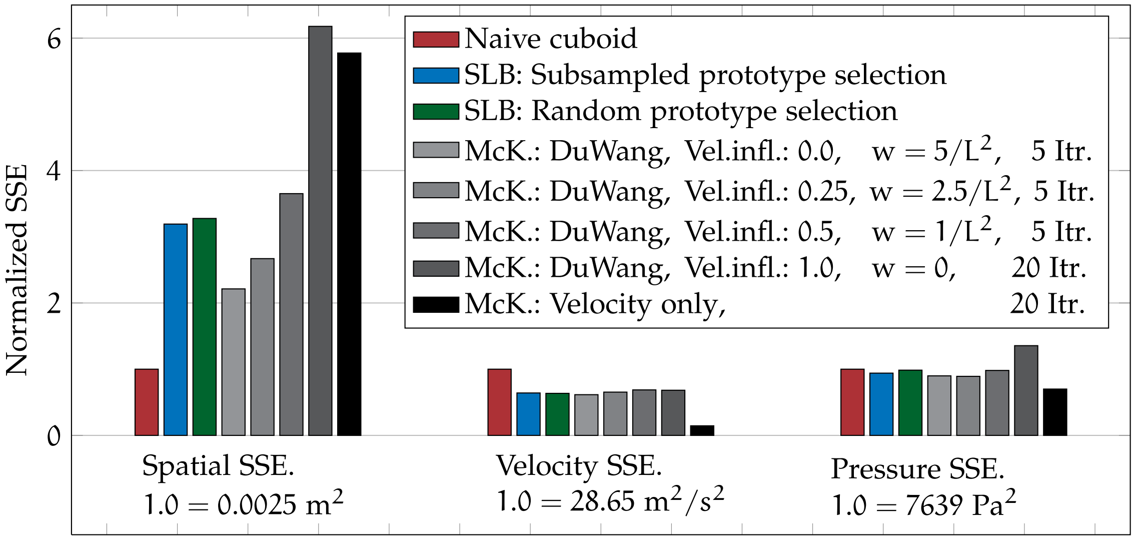


Figure 5.5: Absolute SSE values for different approaches and settings. All three error groups are normalized to the SSEs of the naive cuboid partition. See Table 5.2 for the absolute values.

Color	Description	Spatial (m^2)	Velocity (m^2/s^2)	Pressure (Pa^2)
🟥	Naive cuboid	0.0025	28.65	7639
🟦	SLB: Subsampled	0.0081	18.36	7183
🟩	SLB: Random	0.0083	18.19	7526
◼	McK: DuWang-0.00-5	0.0056	17.62	6869
◼	McK: DuWang-0.25-5	0.0067	18.73	6813
◼	McK: DuWang-0.50-5	0.0092	19.69	7495
◼	McK: DuWang-1.00-20	0.0156	19.54	10 352
⬛	McK: VelocOnly-20	0.0146	4.15	5347

Table 5.2: Absolute SSE values for different approaches and settings. See Figure 5.5 for a graphical representation.

other approaches are therefore not as inadequate as it may seem, they are expected.

Another critical observation concerns McKenzie et al.'s approach with Du and Wang's distortion function; the velocity and pressure errors only depend slightly on the influence factor.[2]

Also notice, that no approach could decrease the velocity SSE nearly as well as McKenzie et al.'s in combination with their own velocity distortion function. This is also true for the Du and Wang distance function with no spatial influence ($w = 0$, i. e., the function at its velocity bias limit). The price for McKenzie et al.'s result is the high spatial SSE.

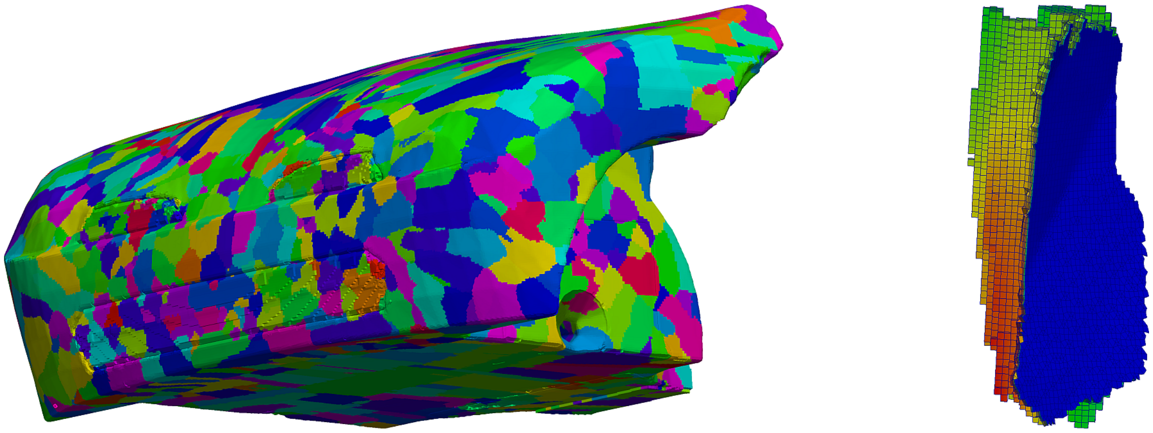
5.2.4.3 *Typical region shapes*

Best insight can be gained by directly examining the partitions in ParaView. Use the threshold filter to isolate regions.

The different error structures shown in the previous section provoke the examination of the region shapes for the different setups. To provide a good insight into the kinds of regions produced, the overall clustering result of each setup is shown together with one typical sample of an isolated region.

Figure 5.6a shows the partition created by the random streamline bundling approach, before graph collapse. The dataset is segmented into 3100 partitions (one partition per identified bundle). The parameters forced the algorithm to find thick, straight bundles. Therefore the basic regions, which are covered by bundles, are shaped similar to cylinders. However, because of outlier cells which are not directly covered by a bundle, these basic regions can grow into more complex shapes. Figure 5.6b shows a typical one, the elongated bulge at the bottom right is due to uncovered outlier cells.

[2] There is room for even greater spatial influence by using higher values for w. However, the need for this was not anticipated (currently a "velocity influence factor" of 0.0 yields a four times higher w than recommended by Du and Wang.



(a) Whole partition.

(b) Isolated region.

Figure 5.6: 🟩 The left image shows a typical partition created by the streamline bundling approach. It illustrates the state after mapping the bundles to regions (≈3100 partitions), but *before* reducing the vertices by collapsing edges. The right image shows one typical example region from this partition, a cylinder-shaped region with a bulge that stems from uncovered cells.

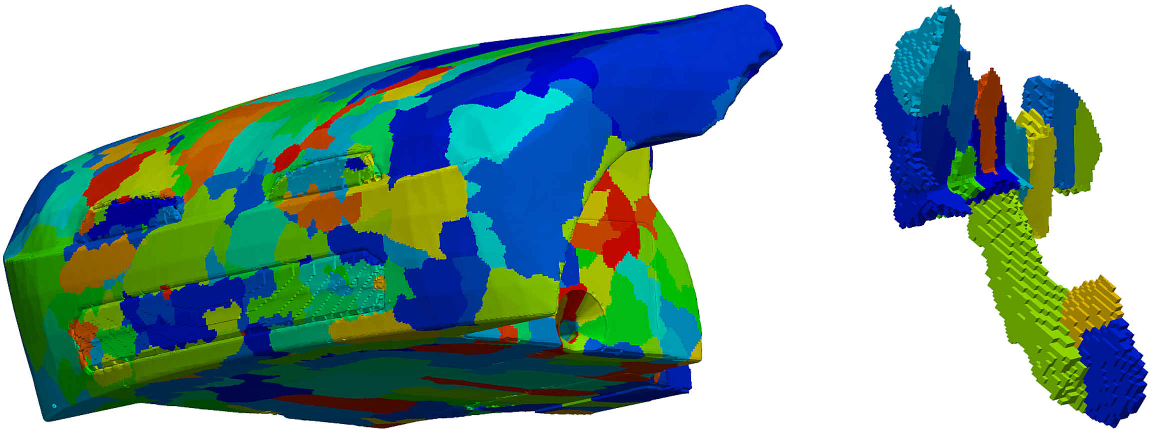
Figure 5.7a shows the situation after graph collapse was performed on the streamline bundling result. As the collapse operation is purely velocity guided, complicated shapes are theoretically possible. In practice, complicated shapes only occur in low interest areas, i. e., in border areas with relatively low velocity. The regions in the center area usually stay compact during the graph collapse stage.

Figure 5.7b shows a partition after graph collapse at the transition between these two areas. The colors encode the regions before graph collapse. The top region lies adjacent to the center region and is therefore compact, even though it consists of 10 bundle regions. The bottom extension branch curls into a low velocity region and happened to have similar velocity to the bulk at the top.

Figure 5.8a displays the partitioning outcome of McKenzie et al. clustering with the velocity-only distortion function. The global optimization nature of the algorithm is immediately recognizable by the smooth region borders. Also notice the sound segmentation of the inlet regions. The little colored disturbances that are visible in some of the regions are due to flat regions that develop near the surface of calm areas. The disturbances stem from the regions lying beneath.

The individual regions are smooth and rounded and tend to be more complex than the ones shown previously. See Figure 5.8b for an example.

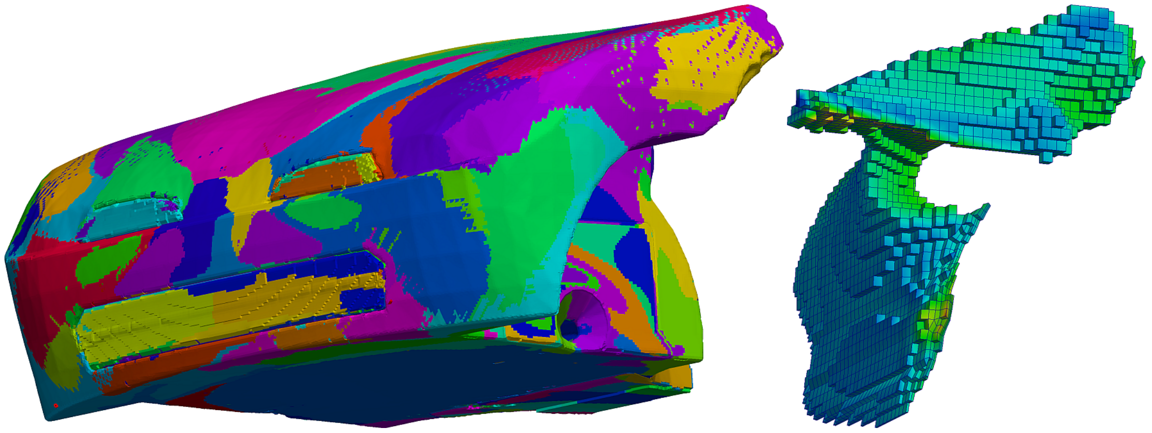
Figure 5.9a shows the overall partition of McKenzie et al. clustering with the Du and Wang distance function and the recommended w (velocity influence factor 0.5). The outcome looks very similar to the previous one, but lacks the small disturbances because of stronger spatial bias.



(a) Whole partition

(b) Isolated region.

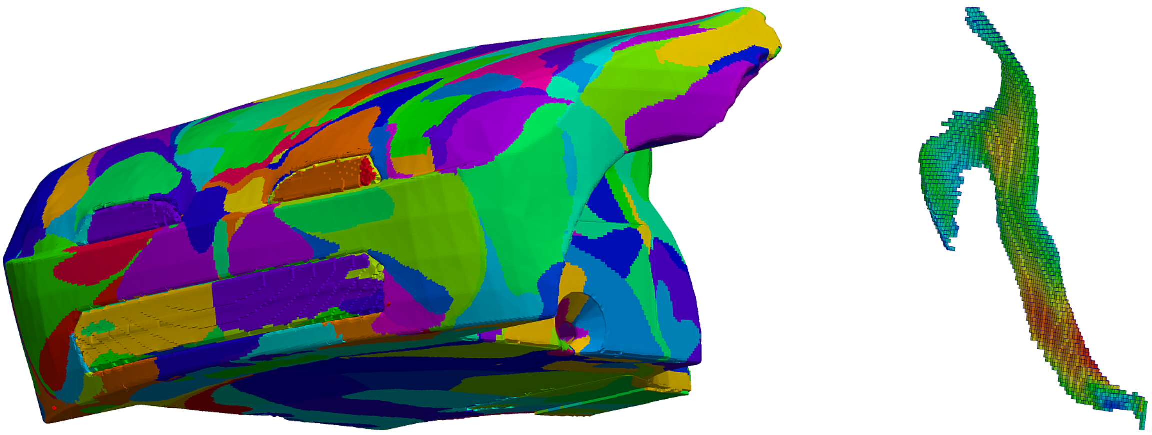
Figure 5.7: 🟩 The left image shows a typical partition of the streamline bundling approach for the Toyota dataset. It illustrates the state *after* collapsing edges to reach 500 vertices. The right image shows one typical example region from this partition.



(a) Whole partition.

(b) Isolated region.

Figure 5.8: ⬛ A partition created by McKenzie et al.'s algorithm with their own velocity-only distortion function (left). The right image shows one typical example region from this partition.



(a) Whole partition

(b) Isolated region.

Figure 5.9: ▬ A partition created by McKenzie et al.'s algorithm with Du and Wang's distance measure (left). The right image shows one typical example region from this partition.

The distance function produces similarly shaped cluster as the velocity-only distortion function. Apart from that, it has the tendency to produce more elongated clusters like the one shown in Figure 5.9b

5.2.4.4 *Required iterations for McKenzie et al.'s approach*

Figure 5.10 and Figure 5.11 show the error development for 40 iterations of McKenzie et al. clustering. The former illustrates the development for the velocity-only distortion function, the latter for Du and Wang's distance function.

Don't make the mistake of thinking Du and Wang's distance measure produces poor results. Iteration zero is the situation after the initial flooding stage, which can already be a good partition.

From these two plots, reasonable iteration counts can directly be derived. 20 iterations is a reasonable choice for the velocity-only distortion function, as the minor improvements after that do not justify the increased calculation time. When utilizing Du and Wang's distance function, there is no use in iterating very long. The algorithm manages to drive down the distortion function, but this does not improve the relevant error measures. Five iterations suffice in this case.

When comparing the values of the two plots, please account for the normalization factors. The plots are mainly meant for guiding the choice of the number of iterations.

5.2.4.5 *Calculation durations*

If computers take over (which seems to be their natural tendency), it will serve us right.
– Alfred Alistair Cooke

All durations in this section correspond to the full Toyota dataset at 190 km/h. The durations for reading the input, fixing the surface normals and cleaning the dataset from outside geometry is the same for all approaches (30 s, 5 s, and 1 min 50 s, respectively). It totals to about 2 min 30 s.

The required time for McKenzie et al.'s approach (velocity-only distortion function) and 500 target clusters is about 1 min 35 s per iteration. Including setup, 20 iterations required 47 min 19 s for the

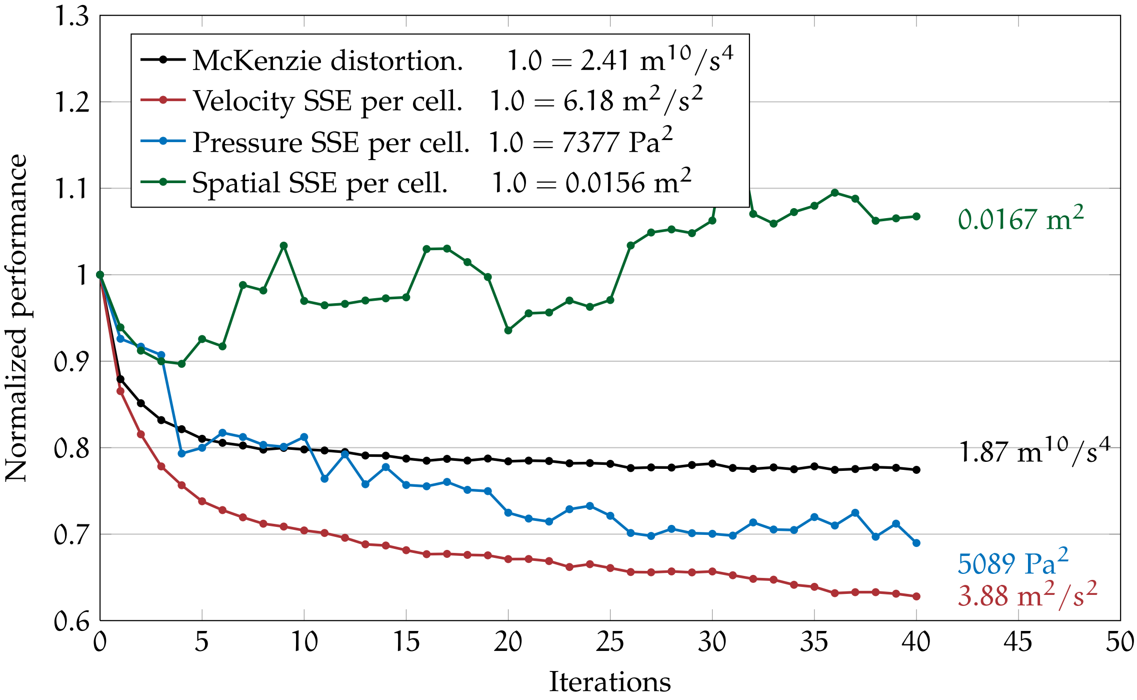


Figure 5.10: Error development of McKenzie et al.'s algorithm over 40 iterations with the velocity-only distortion function. All four value series are normalized to start at 1.0 – see the legend for normalization factors. As expected, the distortion series is constantly decreasing, and the global velocity and global pressure SSEs are both correlated with it. As a tradeoff, the global spatial SSE increases over its initial value.

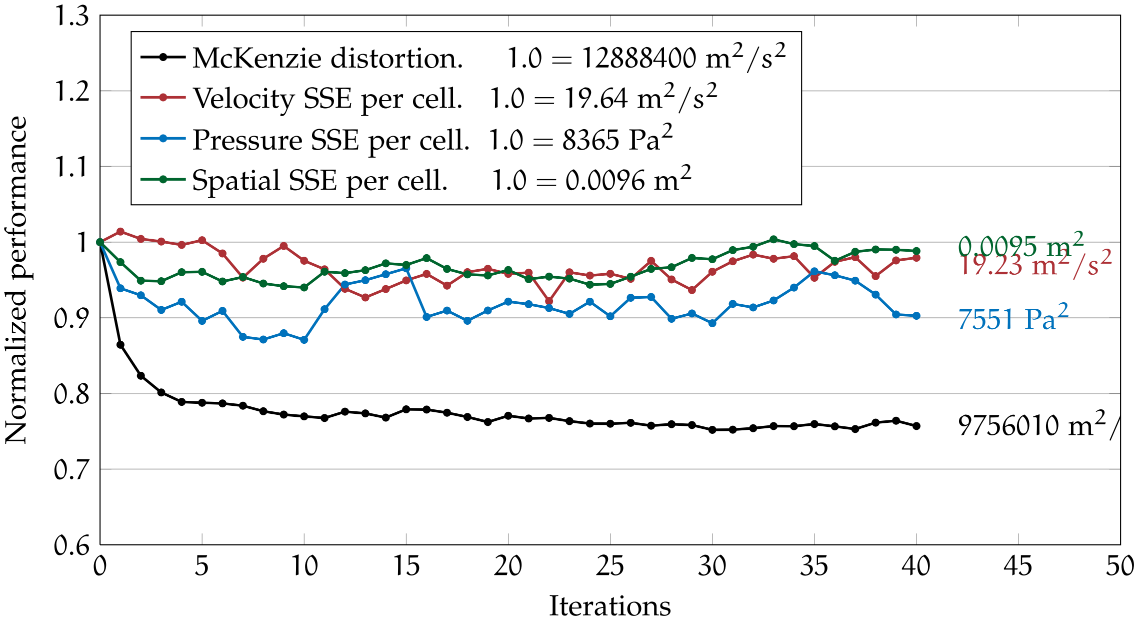


Figure 5.11: Error development of McKenzie et al.'s clustering over 40 iterations with Du and Wang's error measure. All four value series are normalized to start at 1.0 – see the legend for normalization factors. Here, the distortion series decreases too, but it seems to be a mediocre choice for driving down the "real" error functions.

standard example. Mapping the partition of 500 regions to a graph required 5 min 47 s. Therefore this approach requires a *total processing time* of 55 min, when ignoring the times for storing sub results to the hard disk. As large amounts of memory are requested from and returned to the Operating System, the processing time can vary by a few minutes.

Streamline bundling consists of more steps, but the individual steps are carried out faster. Hybrid seeding was finished after 1 min 5 s, stream tracing required 2 min 32 s.

The computational requirements of streamline bundling depend heavily on the used parameters. For the same bundle count, prototype selection by subsampling requires significantly more time than random prototype selection. As the results of both approaches are comparable (see Table 5.1), it is advisable to use random prototype selection for this application. The 3100 bundles were created after 2 min 13 s with random prototype selection.[3] Mapping the bundles to regions is a rather quick operation and was finished after 1 min 44 s. Mapping these regions to a graph is the same operation as in the McKenzie et al. processing chain and required 4 min 13 s. The final collapse graph operation took 13 s.

Hence, the streamline bundling approach required a *total processing time* of 15 min for this application. As before, memory handling by the Operating System leads to varying processing durations.

5.2.5 *Results for different configurations*

Figure 5.12, Figure 5.13 and Figure 5.14 illustrate the behavior for all configurations. The results were created using McKenzie et al.'s approach. All plots show the expected behavior, namely,

- all error measures decrease with increasing region count,
- the spatial SSE is invariant against velocity changes, and
- the velocity and pressure SSEs decrease with decreasing velocity.[4]

.

5.2.6 *Discussion of results for simulation*

This section discussed the results of the implemented algorithms for the resistance graph simulation application. It showed that the methods cover the different tradeoffs between spatial similarity and velocity and pressure similarity quite well.

[3] The subsampling approach would have taken at least 20 min.

[4] The pressure and velocity SSE at 30 km/h are not zero. They are just too small for the scale of the y-axis, which must contain a squared error measure of pressure values in Pa and m^2/s^2, respectively.

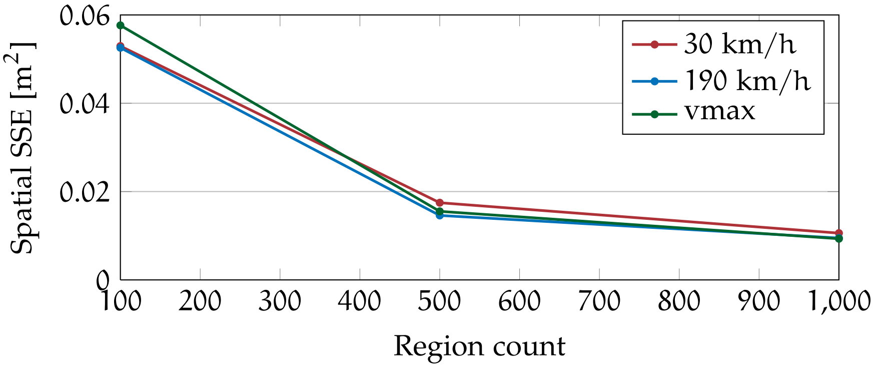


Figure 5.12: Spatial SSE for different operation points (velocities) and region counts. As expected, the spatial SSE decreases with increasing region count and is invariant to velocity changes.

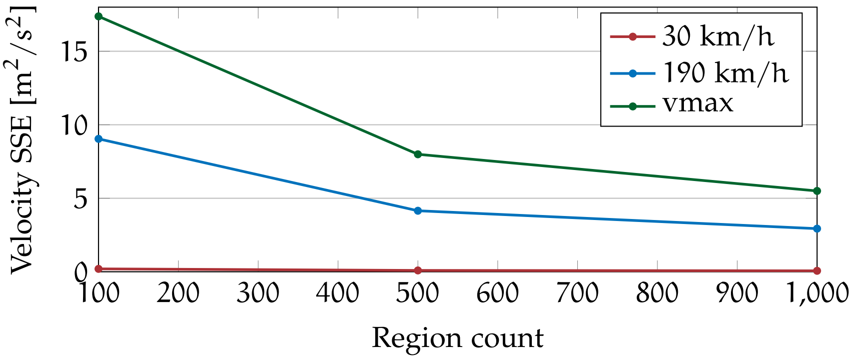


Figure 5.13: Velocity SSE for different operation points (velocities) and region counts. As expected, the spatial SSE decreases with increasing region count and decreasing velocity (less turbulence).

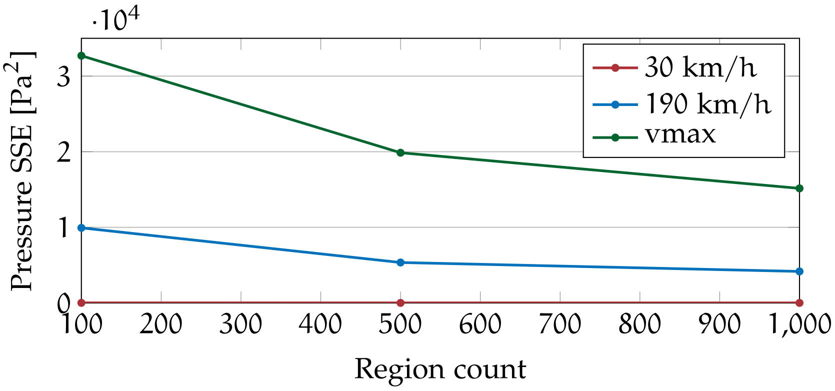


Figure 5.14: Pressure SSE for different operation points (velocities) and region counts. As expected, the pressure SSE decreases with increasing region count and decreasing velocity (less turbulence).

When designing a resistance graph simulator, it is advisable to start with results from McKenzie et al.'s approach. The available choices for this algorithm cover the tradeoffs very well, and the algorithm is simple to configure.

If computation time is of primary concern, one could also look into the streamline bundling approach. It is more fragile and has many parameters, but it computes considerably faster.

To answer the question about the suitable trade-off for resistance graph simulation with certainty, direct simulation is required. However, it is expected that similarity of velocities is more important than spatial compactness.

5.3 APPLICABILITY FOR VISUALIZATION

An attempt at visualizing the Fourth Dimension: Take a point, stretch it into a line, curl it into a circle, twist it into a sphere, and punch through the sphere.
– Albert Einstein

Streamline bundling is well suited for identifying large and homogeneous regions within flow fields, but it is not suited for partitioning turbulent, non-uniform areas. Hence, it is not the optimal choice for the simulation application, as a full partition is required there. However it can be of great value for visualization applications.

To shortly sketch the idea, imagine a streamline bundling algorithm that is configured to identify large, possibly bent bundles of parallel streamlines. Afterwards, by mapping these bundles to regions, a flow graph can be constructed from these regions. Then, two representations of this flow graph can be displayed to the user: the 3D regions of the initial dataset, and the streamline bundles leading to these regions. A third abstract visualization of the flow graph can guide navigation.

The following sections detail different aspects of this idea. Notice, that the described system is only envisioned, not implemented. All screen shots were manually created with the help of ParaView.

5.3.1 *Flow graph representations*

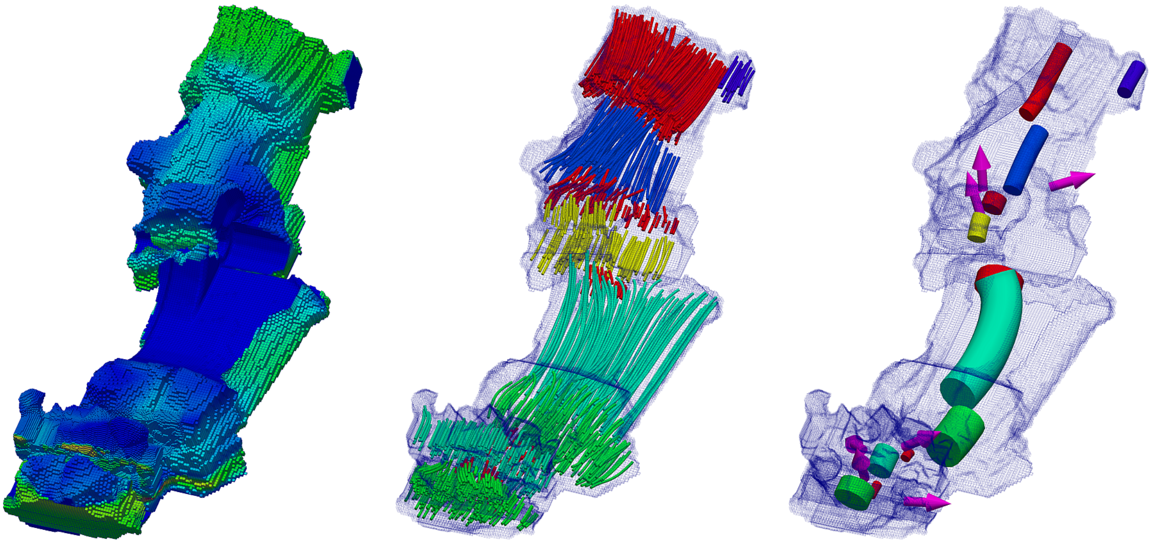
When exploring 3D flow fields there are two important aspects of the individual regions associated to each vertex:

1. The location and extent of a region, and
2. The flow properties within this region.

. A suitable representation for the first aspect is the 3D region itself (3D *representation*), whereas a suitable representation for the second aspect is the underlying streamline bundle (*bundle representation*).

Both of the above representations are still too complex to be understood as a whole for large datasets. If many 3D regions and streamline bundles are displayed at the same time, they hide each other and obscure the view to the overall picture. A third representation is required that introduces some structure and reduces the complexity. One possibility for this representation is a visualization based on simple 3D primitives, like cylinders and arrows (*simple representation*).

Figure 5.15 demonstrates these three representations for the same path within the Toyota dataset. The blue wireframe representation of the region outline aids user orientation.



(a) 3D cell representation. (b) Bundle representation. (c) Simple representation.

Figure 5.15: Different representations of paths through a flow graph.

5.3.2 *Interactive flow graph exploration*

Imagination is more important than knowledge.
– Albert Einstein

The three described representations and the underlying flow graph can be combined into an interactive user interface for exploring and visualizing flow fields. Starting from a default view, the user can interactively explore the dataset by tracing paths through the graph and displaying the associated regions. A click on any inlet traces the strongest path within the underlying flow graph and displays it in simple representation. At any time, the user can switch between the different representations of any path.

Relatively strong edges that are not yet expanded can be displayed as 3D arrows. Clicking on them could expand the strongest path starting from that particular edge.

The following example outlines the envisioned interaction.

5.3.2.1 *Exploration example*

Figure 5.16 shows the beginning of the interaction. The car hull is rendered as a black wireframe to provide initial context. The user chooses to trace the flow graph from two inlet regions, one at the top left inlet and the other at the right of the bottom inlet. The two paths are shown in the simple representation; the magenta colored arrows indicate possible extension points.

The user decides to switch the upper path to the 3D representation, the lower path to bundle representation, and to color both paths by velocity magnitude. The result is shown in Figure 5.17. In order to understand the underlying partitions of the paths and to choose the next extension point, the user colors both paths by region ID (Figure 5.18). By tracing one of the upper extension points, the visualization in Figure 5.19 is obtained. The displayed information becomes overwhelming. Finally, the user switches back to the simple representation to get an overview of the situation, as shown in Figure 5.20.

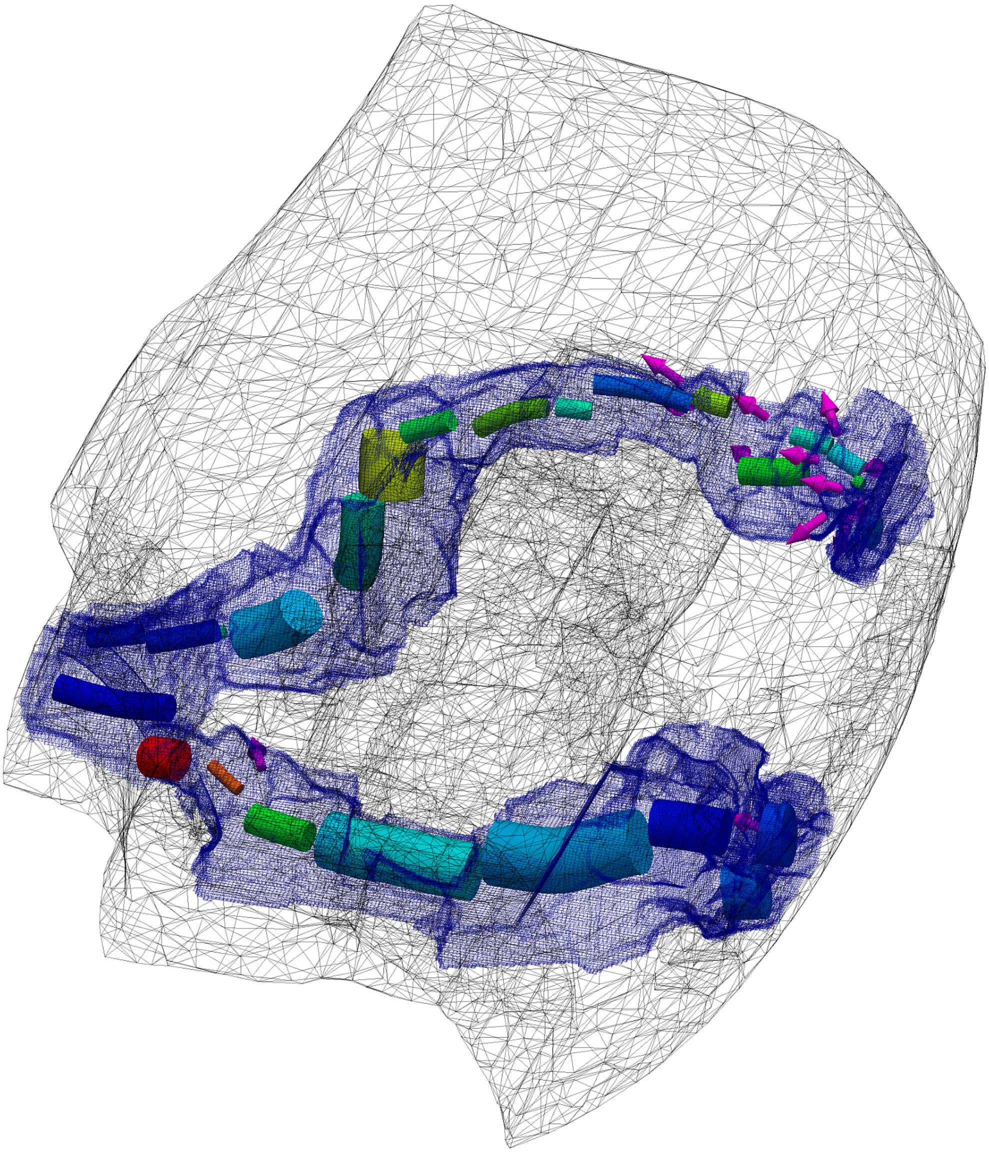


Figure 5.16: Interaction step 1: Initial configuration with black hull wireframe for context and two traced paths in simple representation.

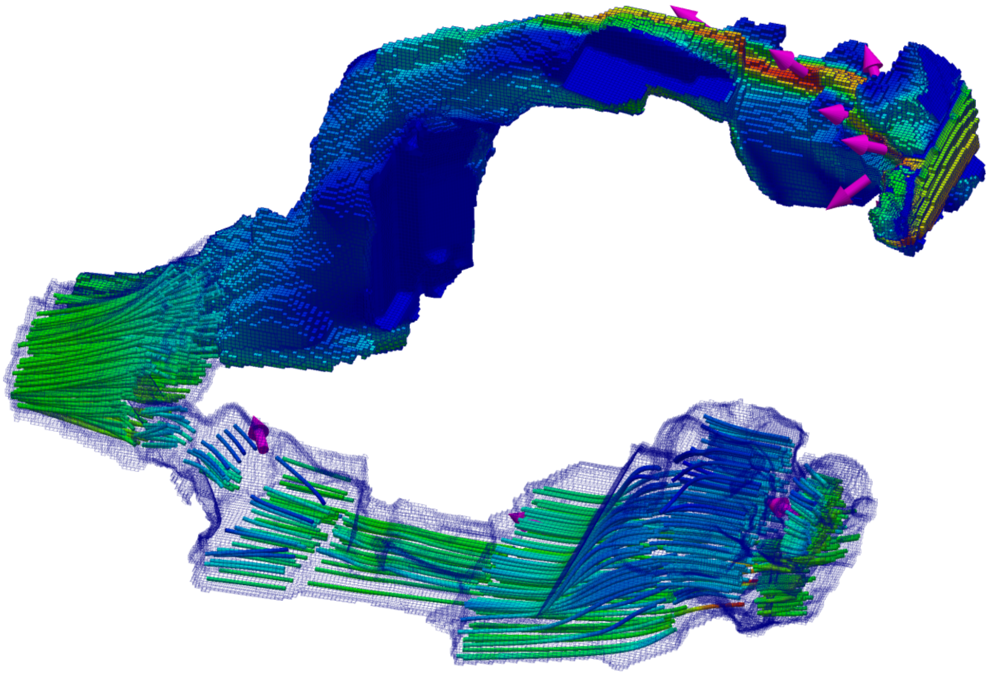


Figure 5.17: Interaction step 2: Situation after hiding the hull and switching the top path to 3D and the bottom path to bundle representation. Both are colored by velocity magnitude.

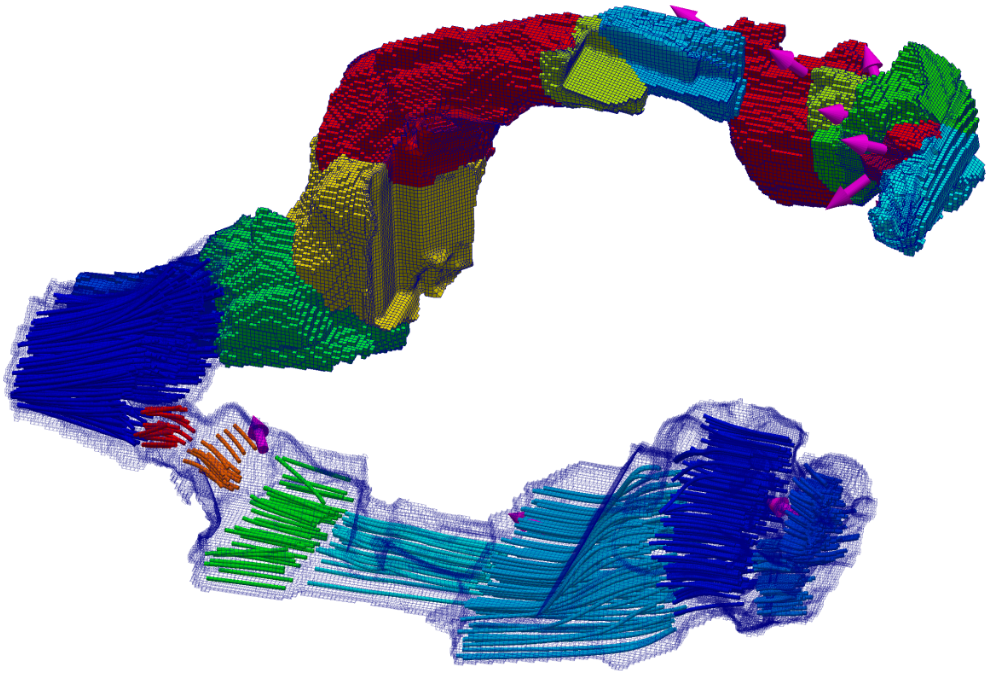


Figure 5.18: Interaction step 3: Result of coloring Figure 5.17 by region identifier.

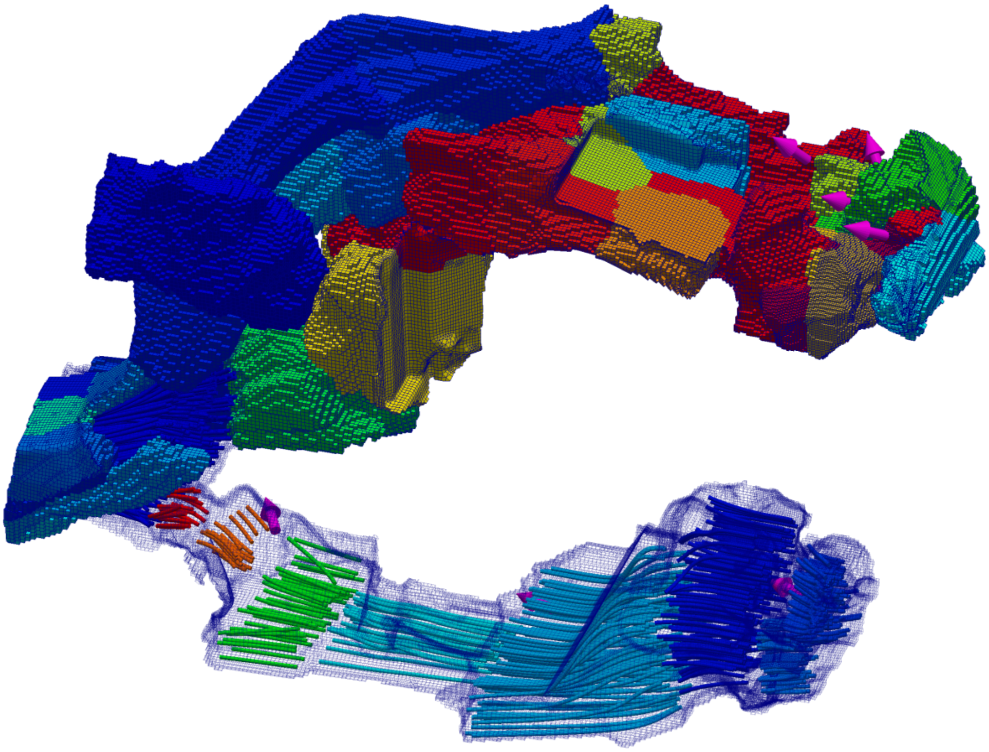


Figure 5.19: Interaction step 4: The result after expanding one of the upper extension points. The screen starts to look cluttered.

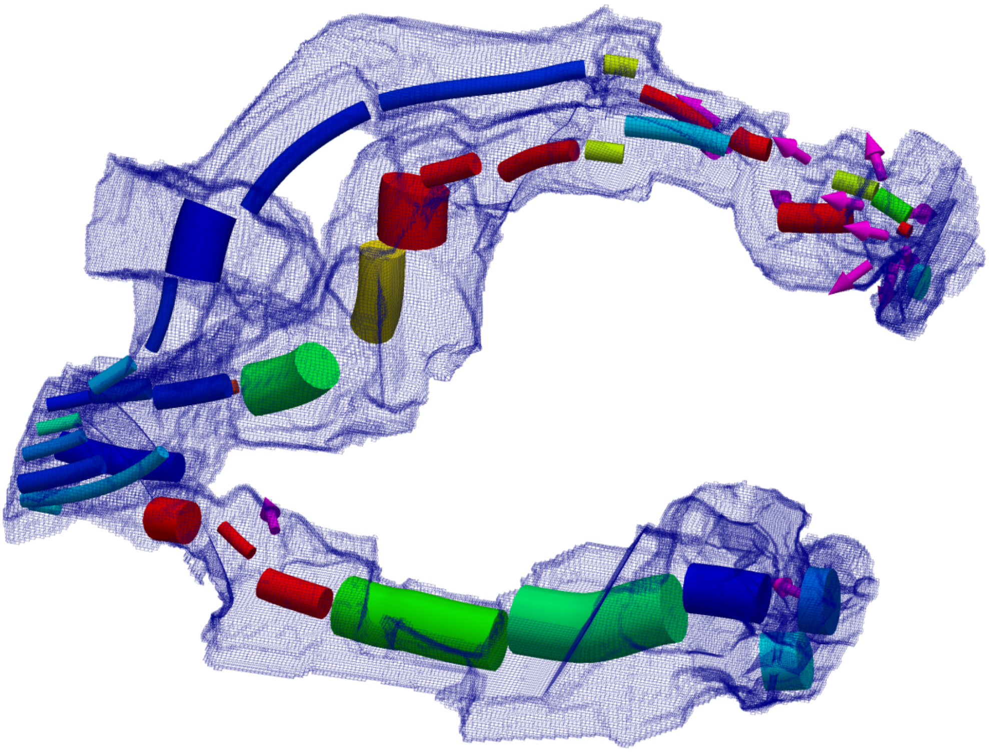


Figure 5.20: Interaction 5: After switching back to the simple representation for all paths, the situation is clear again.

5.4 RESULTS FOR THE CENTRIFUGAL PUMP DATASET

The Centrifugal pump dataset contains three flow simulations of a centrifugal pump that differ only in the simulation approach. It was provided for the IEEE Visualization Contest 2011 [5]. Figure 5.21 displays a view of the dataset cut in half.

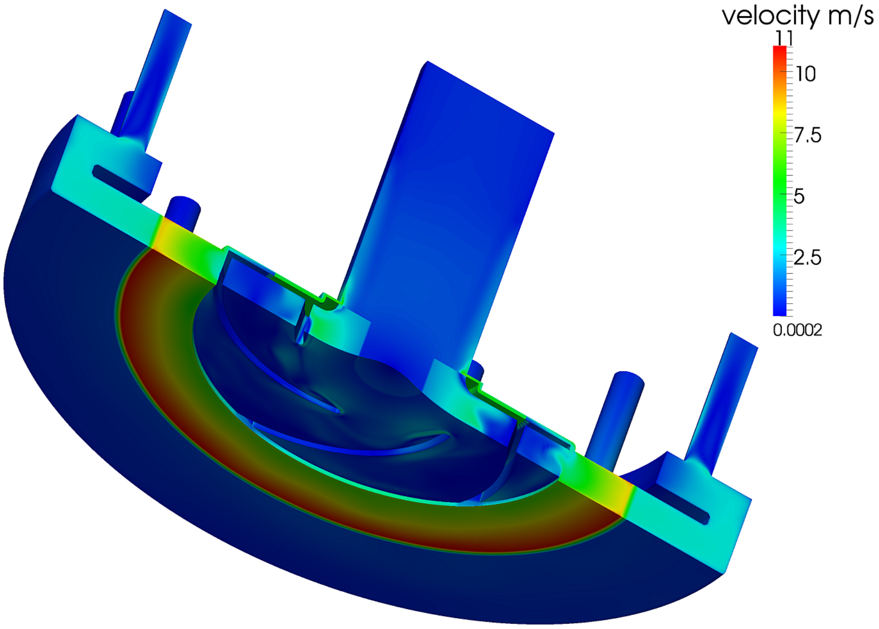


Figure 5.21: The centrifugal pump dataset cut in half and colored by velocity magnitude.

The original goal of the contest was turbulence modeling, which cannot be provided by the algorithms presented in this thesis. Nevertheless the dataset is a good candidate to evaluate streamline bundling, because the general flow of a centrifugal pump is well defined.

The parameters for applying streamline bundling to this dataset were set quite strictly, with small differences in angular velocity (20° for initial slices and 40° for incremental slices) and high mate ratios (0.9 for initial slice, 0.8 for incremental slices and 0.7 to stop bundling). Prototype selection was performed using subsampling, bundle collisions were always resolved by keeping the existing bundle.

Figure 5.22 shows the streamline bundling result with similar view settings as in Figure 5.21. A full view of the result is shown in Figure 5.23.

The inlet in the middle of the dataset is nicely segmented into one bundle (violet bundle), except of the circular flows at the boundaries of the inlet tube. The turbulent flow within the impeller is covered by many small bundles. The result in this region could definitely be improved to better emphasize the flow between the blades of the impeller. The circular flow at the outer ring of the pump is acceptably covered by thick bundles.

Overall, the bundles cover the dataset quite well and almost as expected. The result would definitely allow exploring the dataset using the methods described above. However, some improvements are required to better accommodate the inherent circular flows within the dataset and its resulting flow graph.

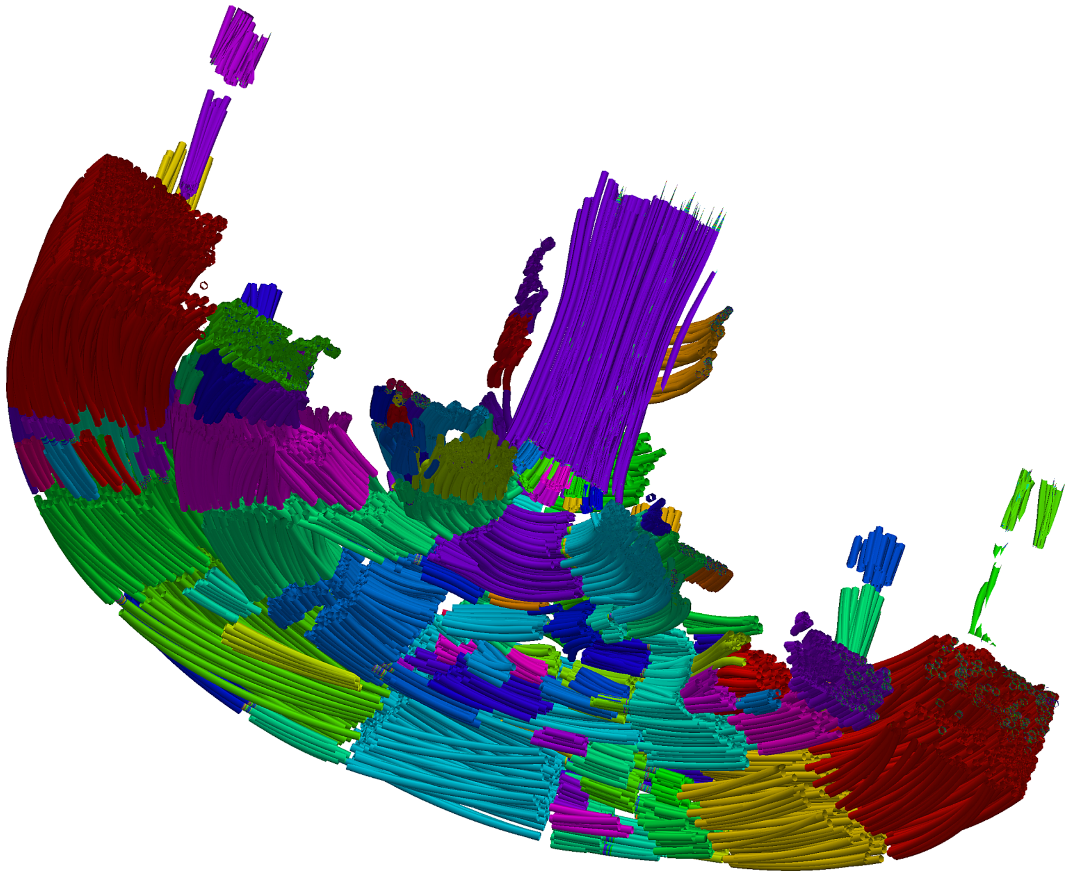


Figure 5.22: Result of streamline bundling for the Centrifugal pump dataset. The result was cut in half to increase insight.

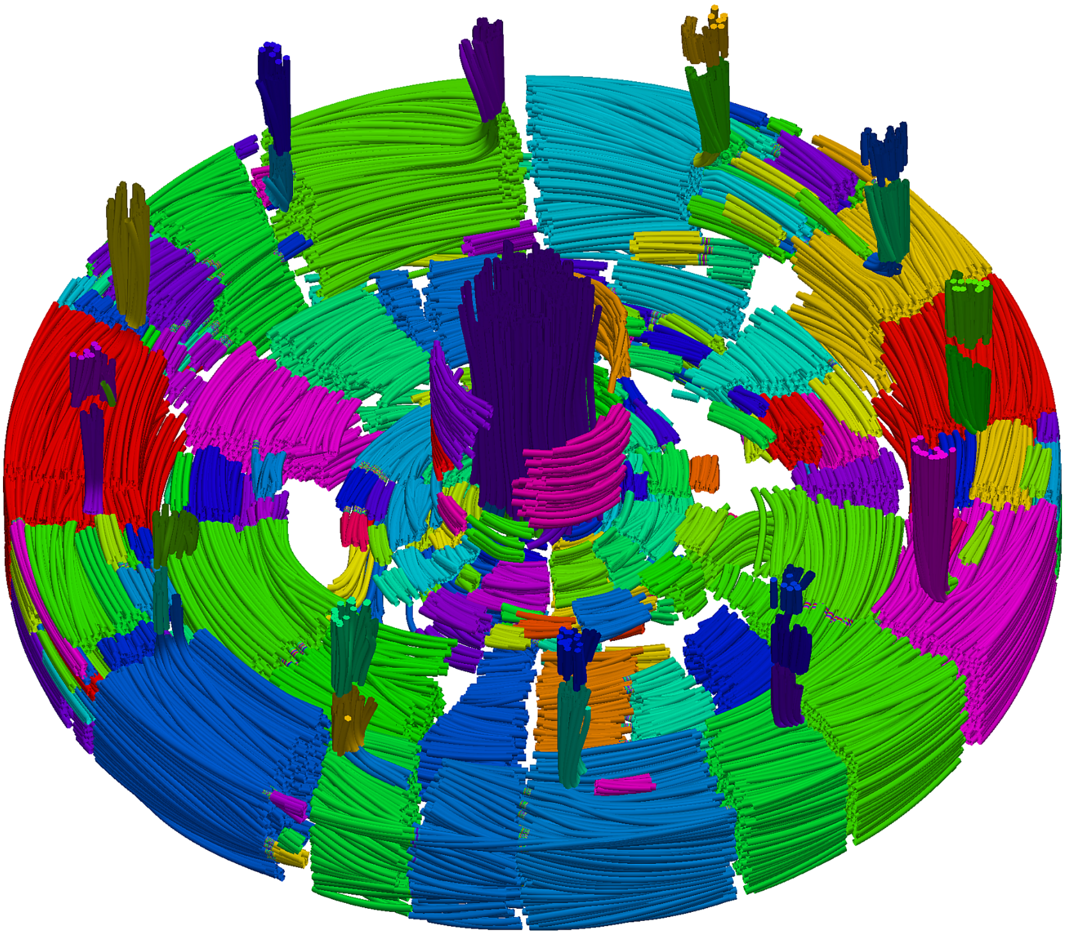


Figure 5.23: Result of streamline bundling for the Centrifugal pump dataset. Full view.

6 CONCLUSION

I have not failed. I've just found 10,000 ways that won't work.
– Thomas Alva Edison (1847-1931)

This work was motivated by the construction of car cooling systems. At first two of the main existing engineering paths, namely expert generated resistance graphs and CFD simulations, were explained. Afterwards, a third path combining the advantages of the former two was suggested, *CFD-derived resistance graphs*. To derive such graphs, CFD simulation results (*flow fields*) need to be simplified to meaningful flow graphs. This can be achieved by clustering similar cells of the CFD simulation result. Furthermore it was argued that meaningful flow graphs can also aid in understanding flow fields of other domains and that they can be employed for visualization purposes.

During the discussion of related work, the main shortcoming of existing point-based cluster algorithms was identified to be the inherent creation of disconnected clusters. McKenzie et al.'s adaption of variational shape approximation was identified as the best existing candidate for creating flow graphs to enable resistance graph simulation.

As McKenzie et al.'s algorithm was expected to require high computational effort, a new method for partitioning flow fields was proposed. The main idea behind *streamline bundling* is to densely cover the dataset with streamlines and bundle parallel and spatially close streamline segments. Afterwards the cells of the flow field are assigned to their closest streamline bundle in order to derive a complete partition.

Related work for clustering streamlines was identified in the field of medical image processing (diffusion MRI). All suggested methods cluster whole streamlines instead of streamline segments. A short discussion showed why these methods cannot be extended easily to bundle streamline segments.

The actual streamline bundling algorithm consists of several steps. After *preprocessing* the dataset, meaningful *seed points* have to be identified. Starting from these seed points a *stream tracer* generates streamlines. The actual *streamline bundling* algorithm is based on repeated intersections of the dataset by a sweep plane which is moved along a prototype streamline. Streamline segments that are intersected by all of these sweep planes are combined to form a bundle.

The main drawback of the proposed solution is the high number of parameters that are required for seeding, for stream tracing, for *prototype selection*, to measure similarity, to resolve bundle collisions, and to define *stopping criteria*. However, for specific applications, many of these parameters can be fixed to default values.

For the resistance graph application, McKenzie et al.'s approach was shown to be the best overall choice, with streamline bundling having advantages only in calculation speed. Further experiments with a resistance graph simulator will be required to determine if the accuracy of the generated flow graphs is sufficient.

Finally, possible visualization applications of streamline bundling and flow graphs were discussed.

6.1 FUTURE WORK

Humans would definitely place cable ties differently, wouldn't we?

When looking at the result figures of the bundling stage, there is clearly potential for improvements. This section briefly treats the most promising ideas for enhancements.

Instead of separate seeding and stream tracing phases, a promising alternative would be a hybrid approach that combines them. The hybrid tracer could alternate between placing random seed points and tracing streamlines. By placing the seed points in the largest gaps between existing streamlines, dense coverage of the dataset could be achieved with fewer streamlines. A single parameter could be used to stop tracing if the largest gap is below a predefined threshold.

Several promising improvements come to mind for streamline bundling. The first of these applies to bundle tracing. Currently the stopping criteria for tracing a single bundle are arbitrary and not very direct. A fast volume measure for streamline bundles could improve this. The tracing could stop, if the volume begins to decrease, which would happen if the loss of streamlines (diameter) decreases the volume more than the increase of steps (length) increases it. This would naturally lead to bundles covering the highest possible volume, without the need to artificially favor thin elongated or thick short bundles.

Another improvement concerns the collision strategy. The "overwrite existing bundles if worse" collision strategy can lead to disconnected or degenerated bundles. The "remove existing bundles if worse" strategy on the other hand, makes local, binary decisions – if two good bundles overlap at only one streamline, one of them is removed. A less local bundle collision approach that compares final bundles against each other and solves collisions globally is expected to improve the results significantly.

In general, the streamline bundling idea would certainly benefit from a less local approach of generating bundles and trading them of against each other.

In summary, streamline bundling is not yet applicably for general purpose applications handling arbitrary flow fields. Too many parameters have to be set and the influence on the result is too indirect for most of these parameters.

However, the two applications discussed in this thesis could be served by the current framework. Resistance graph simulators can readily be built using the implementation of McKenzie et al.'s algorithm. In well defined domains, the current state of streamline bundling already produces viable results for visualization applications, because most of the parameters can be fixed.

A

CONTENTS OF THE ACCOMPANYING DVD

FORMAT: DVD-R, Single Layer, ISO9660-Format

PATH: `/`

`/application/`	Contains `UIChain` for 64 bit versions of Windows (installer).
`/data/`	Contains all datasets that where used within this thesis.
`/data/rawdata/`	Datasets as they were received (EnSight format).
`/data/cleandata.zip`	Cleaned datasets that are ready to use with `UIChain`.
`/dev/`	Files and information for developers.
`/dev/software/`	Some of the required software for developers on Windows.
`/dev/install.txt`	Installation instructions for developers.
`/dev/UIChain.zip`	Source code of `UIChain`.
`/references/`	Contains all cited references.
`/references/related/`	Contains related material that was not cited.
`/results/SST_vis`	3D results for the pump dataset, as presented in Section 5.4.
`/Thesis.pdf`	The electronic version of this thesis.

BIBLIOGRAPHY

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75:245–248, May 2009. (Cited on page 18.)

[2] Robert Bringhurst. *The Elements of Typographic Style*. Version 2.5. Hartley & Marks, Publishers, Point Roberts, WA, USA, 2002. (Cited on page 105.)

[3] A. Brun, H. Knutsson, H. J. Park, M. E. Shenton, and C.-F. Westin. Clustering fiber tracts using normalized cuts. In *Seventh International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI 2004)*, Lecture Notes in Computer Science, pages 368–375, Rennes - Saint Malo, France, September 2004. (Cited on page 27.)

[4] Lewis Carroll and John Tenniel. *Alice in Wonderland*. Peter Pauper Press, Mount Vernon, NY, 1953. (Cited on pages 1, 12, and 71.)

[5] Centrifugal pump dataset. Centrifugal pump dataset (SST simulation). Data provided for the IEEE Visualization Contest 2011 for the VisWeek 2011 held in Rhode Island Convention Center, Providence, USA. The data is courtesy of the Institute of Applied Mechanics, Clausthal University, Germany (Dipl. Wirtsch.-Ing. Andreas Lucius)., 2011. URL `http://viscontest.sdsc.edu/2011/dataset/dataset.html`. (Cited on pages 92, 93, and 94.)

[6] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 905–914, New York, NY, USA, 2004. ACM. (Cited on pages 20, 21, and 22.)

[7] Isabelle Corouge, Sylvain Gouttard, and Guido Gerig. Towards a shape model of white matter fiber bundles using diffusion tensor MRI. In *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*, volume 1, pages 344–347, 2004. (Cited on page 26.)

[8] Zhaohua Ding, John C. Gore, and Adam W. Anderson. Classification and quantification of neuronal fiber pathways using diffusion tensor MRI. *Magnetic Resonance in Medicine*, 49(4):716–721, 2003. (Cited on page 26.)

[9] Qiang Du and Xiaoqiang Wang. Centroidal voronoi tessellation based algorithms for vector fields visualization and segmentation. In *Proceedings of the conference on Visualization '04*, VIS '04,

pages 43–50, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on pages 18, 19, 57, 77, 80, 81, 83, and 84.)

[10] EnSight. Ensight engineering visualization software. Computational Engineering International, Inc. (CEI)., 2010. URL `http://www.ensight.com`. (Cited on pages 30, 52, 53, and 99.)

[11] Brian S. Everitt. *Cluster analysis*. Heinemann Educational for Social Science Research Council, London, 1974. (Cited on page 9.)

[12] FLUENT 12. ANSYS FLUENT Flow Modeling Simulation Software (Release 12). ANSYS, Inc. Southpointe, 275 Technology Drive, Canonsburg, PA 15317, USA, 2010. URL `http://www.ansys.com`. (Cited on page 30.)

[13] R. L. Hardy. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research*, 76, 1971. (Cited on page 16.)

[14] Bjoern Heckel, Gunther Weber, Bernd Hamann, and Kenneth I. Joy. Construction of vector field hierarchies. In *Proceedings of the conference on Visualization '99: celebrating ten years*, VIS '99, pages 19–25, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. (Cited on pages 12, 15, 16, and 17.)

[15] KULI. Software for vehicle heat management optimization. KULI is a Software of: Engineering Center Steyr GmbH & Co KG, Steyrer Strasse 32, 4300 St.Valentin, Austria, 2010. URL `http://www.kuli.at`. (Cited on page 2.)

[16] Denis Le Bihan, Jean-François Mangin, Cyril Poupon, Chris A. Clark, Sabina Pappata, Nicolas Molko, and Hughes Chabriat. Diffusion tensor imaging: Concepts and applications. *Journal of Magnetic Resonance Imaging*, 13(4):534–546, 2001. (Cited on page 24.)

[17] Stuart P. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, IT-28(2):129–137, March 1982. (Cited on page 17.)

[18] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967. (Cited on page 17.)

[19] Mahnaz Maddah, Andrea Mewes, Steven Haker, W. Grimson, and Simon Warfield. Automated Atlas-Based Clustering of White Matter Fiber Tracts from DTMRI. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI 2005)*, volume 3749 of *Lecture Notes in Computer Science*, pages 188–195. Springer Berlin, Heidelberg, 2005. (Cited on page 27.)

[20] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The Planar k-means Problem is NP-Hard. In *Proceedings of the 3rd International Workshop on Algorithms and Computation*, WALCOM '09, pages 274–285, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 18.)

[21] Alexander McKenzie, Santiago V. Lombeyda, and Mathieu Desbrun. Vector field analysis and visualization through variational clustering. In *IEEE EUROGRAPHICS VGTC Symposium on Visualization*, 2005. (Cited on pages x, 12, 20, 21, 22, 23, 29, 30, 33, 56, 57, 73, 75, 77, 79, 80, 81, 82, 83, 84, 85, 87, 95, 96, and 97.)

[22] Tony McLoughlin, Robert S. Laramee, Ronald Peikert, Frits H. Post, and Min Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010. (Cited on page 34.)

[23] D. Mumford. and J. Shah. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, 1989. (Cited on page 11.)

[24] Lauren J. O'Donnell and Carl-Fredrik Westin. Automatic tractography segmentation using a high-dimensional white matter atlas. *IEEE Transactions on Medical Imaging*, 26(11):1562–1575, November 2007. (Cited on page 27.)

[25] ParaView. Paraview, open-source, multi-platform data analysis and visualization application. Released by Kitware, Inc., 28 Corporate Drive, Clifton Park, NY 12065 USA, 2010. URL `http://www.vtk.org`. Version 3.8.0. (Cited on pages 51, 52, 53, 70, 71, 80, 87, and 105.)

[26] Qt. Qt – cross-platform C++ application framework. Released under GNU Lesser General Public License (LGPL) version 2.1. by Nokia Norge AS, Sandakerveien 116, NO-0484 Oslo, Norway., 2010. URL `http://qt.nokia.com/`. Version 4.6.2. (Cited on pages xi, 71, and 72.)

[27] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. (Cited on page 51.)

[28] Thomas Schultz. Feature Extraction for DW-MRI Visualization: The State of the Art and Beyond, 2009. URL `http://www.ci.uchicago.edu/~schultz/papers/schultz-dwmri-feature-survey-2010.pdf`. To appear in:

Proc. Dagstuhl Scientific Visualization Workshop 2009. ISSN 1862-4405. (Cited on pages 24, 25, and 26.)

[29] Toyota dataset. Toyota Prius 3D CFD dataset. Model delivered together with the software packages RadTherm and RadThermIR by Thermoanalytics Inc. CFD flow simulation by ViF., 2010. URL `http://www.thermoanalytics.com/products/radtherm`. (Cited on pages xi, 3, 4, 7, 8, 30, 32, 34, 35, 36, 51, 52, 54, 73, 74, 75, 76, 77, 82, 83, and 88.)

[30] VTK. VTK – the visualization toolkit. Released under BSD License by Kitware, Inc., 28 Corporate Drive, Clifton Park, NY 12065 USA, 2010. URL `http://www.vtk.org`. Version coming with ParaView 3.8.0. (Cited on pages xi, 30, 51, 52, 53, 57, 66, 69, 70, and 76.)

[31] VTK File Formats. VTK File Formats for VTK Version 4.2x. Excerpt from VTK User's Guide. Accessible through Kitware VTK documentation site, 2011. URL `http://www.vtk.org/VTK/help/documentation.html`. (Cited on page 52.)

[32] Yan Xia, And Turken, Susan Whitfield-Gabrieli, and John Gabrieli. Knowledge-based classification of neuronal fibers in entire brain. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI 2005)*, volume 3749, pages 205–212. Springer Berlin, Heidelberg, 2005. (Cited on page 27.)

[33] Song Zhang, Stephen Correia, and David H. Laidlaw. Identifying White-Matter Fiber Bundles in DTI Data Using an Automated Proximity-Based Fiber-Clustering Method. *IEEE Transactions on Visualization and Computer Graphics*, 14:1044–1053, September 2008. (Cited on page 27.)

COLOPHON

The style of this thesis was deliberately chosen with emphasis on typography instead of space conservation. It therefore features large margins to improve readability. Furthermore the margins provide space for quotations, interesting facts and, possibly, your notes.

The following text is adopted unchanged from the Colophon of the `classicthesis` package. I want to thank all of the mentioned people for their contributions to this wonderful template:

> This thesis was typeset with LaTeX 2$_\varepsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)
>
> The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style* [2]. It is available for LaTeX via CTAN as "`classicthesis`"

All illustrations where created using *Inkscape*.[1] The plots were generated in *MATLAB*;[2] automatic conversion for the LaTeX graphics package *PGF and TikZ*[3] was conveniently done using a slightly modified version of *matlab2tikz*.[4] The 3D figures of the datasets were rendered using *ParaView* [25].

Final Version as of December 5, 2011 at 12:55.

[1] `http://www.inkscape.org`.

[2] `http://www.mathworks.de/products/matlab/index.html`.

[3] `http://sourceforge.net/projects/pgf/`.

[4] `http://win.ua.ac.be/~nschloe/content/matlab2tikz`.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, Austria, December 2011

Benjamin Hopfer