Diss. ETH No. 16640

Selective Visualisation of Unsteady 3D Flow using Scale-Space and Feature-Based Techniques

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH) Zürich

for the degree of
Doctor of Technical Sciences

presented by

Dirk Bauer
Dipl.-Inf., University of Tübingen
Swiss Federal Institute of Technology (ETH) Zürich
born January 1, 1971
citizen of Germany

accepted on the recommendation of

Prof. Dr. Markus Gross, examiner
Prof. Frits Post, co-examiner
Dr. Ronald Peikert, co-examiner

2006

ABSTRACT

During the recent years, visualisation has become an important part of scientific and engineering work. In parallel to the continuously increasing computing power available, the amount of data to be processed has become larger quickly. As is the case for the internet, there is a growing gap between the quantity of information that is theoretically available and the ability to efficiently process and handle this information. The extraction of the essential contents of the information and its reduction to a reasonable degree thus becomes a more and more important task.

This also holds for the field of *flow visualisation*, where the underlying industrial datasets mostly are based on large computational grids originating from CFD (Computational Fluid Dynamics). A grid of this type often contains about one million or even more cells and nodes. The data given on these structures usually are vector and scalar fields, often given for several hundred time steps in the case of *unsteady* flow. As a consequence of this, the raw data consume significant disk space, and it is not possible to inspect and analyse every detail of the flow.

Nevertheless there is a way out of this dilemma, since most of the flow details are not needed for understanding the general flow behaviour. The designers of water turbines, for example, are mainly interested in special flow structures like *vortices*, since these can cause undesired vibrations and resonance, leading to reduced efficiency, increased material abrasion, and in the worst case to machine damage. The essential parts of the flow, which are called *features*, can be extracted from the flow using special criteria and algorithms.

The concept of feature extraction is part of the more general task of *selective visualisation* of flow. Several vortex criteria can be found in literature, which comprise the definition of line-type features like *vortex core lines* and surface-type features like *vortex hulls*. Also, a stream of mass-less *particles* in a flow can help identify its critical regions. All these paradigms can be applied to steady and time-dependent flow likewise. In the latter case, *feature tracking* is performed over successive datasets.

In addition to conventional rendering and displaying techniques, *virtual environments* are becoming more and more popular in science and industry. VR technologies enhance the visual and acoustic perception by use of new input and output devices, such as lightsabers, 3D glasses, head trackers, large projection screens and stereo sound. They provide a way to "immerse" into a 3-dimensional graphical scene and thus to better explore and understand the structure of the visualised data. The field of flow visualisation can also benefit from these techniques and is therefore supported by VR development packages from several current visualisation platforms.

The goal of this thesis is to investigate the essential features and behaviour of unsteady flow in the context of flow visualisation and industrial application. The focus is on new feature-based, region-based and integration-based methods for selective visualisation of 3D flow given in terms of vector and scalar fields on unstructured three-dimensional grids. Scale-space techniques known from computer vision improve the quality of the feature extraction. The results are also presented in a virtual reality environment and help our industry partners improve the design of turbomachinery parts, e.g. of water turbines.

ZUSAMMENFASSUNG

In den letzten Jahren wurde Visualisierung zu einem wichtigen Bestandteil wissenschaftlichen und technischen Arbeitens. Parallel zur stetig wachsenden Rechnerleistung sind auch die zu verarbeitenden Datenmengen schnell angestiegen. Wie beim Internet existiert eine wachsende Lücke zwischen der Quantität an theoretisch verfügbarer Information und der Fähigkeit, diese noch effizient verarbeiten und mit ihr umgehen zu können. Das Herausfiltern wichtiger Inhalte aus der Information und deren Verringerung auf ein vernünftiges Mass wird deshalb zur immer wichtigeren Aufgabe.

Dies trifft ebenfalls auf das Gebiet der *Strömungsvisualisierung* zu, wo die zugrundeliegenden industriellen Datensätze meist auf grossen, aus der rechnergestützten Strömungsdynamik stammenden Rechengittern basieren. Ein solches Gitter enthält oftmals etwa eine Million oder mehr Zellen und Knoten. Die auf diesen Strukturen gegebenen Daten sind gewöhnlich Vektor- und Skalarfelder, oft für mehrere hundert Zeitschritte im Falle einer *zeitabhängigen* Strömung. Als Folge davon verbrauchen die Rohdaten signifikanten Plattenplatz, und es ist nicht möglich, jedes Detail der Strömung zu betrachten.

Trotzdem gibt es einen Ausweg aus diesem Dilemma, da die meisten Details für das Verständnis des generellen Strömungsverhaltens nicht benötigt werden. Die Designer von Wasserturbinen sind zum Beispiel hauptsächlich an speziellen Strukturen wie *Wirbeln* interessiert, da diese unerwünschte Vibrationen und Resonanz verursachen können, welche den Wirkungsgrad erniedrigen, das Material abnutzen und im schlimmsten Fall Maschinenschäden hervorrufen. Die wesentlichen Teile der Strömung, auch *Merkmale* genannt, können mittels spezieller Kriterien und Verfahren ermittelt werden.

Die Merkmalsextraktion gehört zur *selektiven Visualisierung*. In der Literatur finden sich hierfür verschiedene Wirbelkriterien für Linien-Merkmale wie *Wirbelkernlinien* und Oberflächen-Merkmale wie *Wirbelhüllen*. Ein *Partikelstrom* kann ebenfalls helfen, die kritischen Regionen einer Strömung zu identifizieren. All diese Paradigmen können auf konstante und zeitabhängige Strömung gleichermassen angewendet werden. Im letzteren Fall wird *Merkmalverfolgung* auf aufeinanderfolgenden Datensätzen durchgeführt.

Zusätzlich zu konventionellen Rendering- und Anzeigetechniken werden *virtuelle Umgebungen* in Wissenschaft und Industrie immer beliebter. VR-Technologien erweitern die visuelle und akustische Wahrnehmung durch Gebrauch neuer Peripheriegeräte, wie z.B. Laserschwerter, 3D-Brillen, Headtracker, grosse Projektionsflächen und Stereosound. Sie ermöglichen, in eine graphische 3D-Szene "einzutauchen" und somit die Struktur der visualisierten Daten besser zu verstehen. Strömungsvisualisierung kann ebenfalls von diesen Techniken profitieren und wird deshalb durch VR-Entwicklungspakete mehrerer aktueller Visualisierungs-Plattformen unterstützt.

Das Ziel dieser Dissertation ist, wesentliche Merkmale und Verhalten zeitabhängiger Strömungen im Kontext wissenschaftlicher Visualisierung und industrieller Anwendung zu untersuchen. Der Schwerpunkt liegt dabei auf merkmals-, gebiets- und integrationsbasierten Methoden zur selektiven Strömungsvisualisierung in Form von Vektor- und Skalarfeldern auf unstrukturierten 3D-Gittern. Aus der Computer Vision bekannte Scale-Space-Techniken verbessern die Qualität der Merkmalsextraktion. Die Ergebnisse werden auch in einer VR-Umgebung präsentiert und helfen unseren Industriepartnern, das Design von Strömungsmaschinen, z.B. Wasserturbinen, zu verbessern.

ACKNOWLEDGEMENTS

I especially want to thank my advisor Prof. Markus Gross and my co-advisor Dr. Ronald Peikert for their invaluable advise and guidance to this thesis. Their enthusiasm and comprehensive scientific and technical knowledge were always a great source of motivation and encouragement to me. It was a great time at the Computer Graphics Laboratory which led to new and inspiring insights and experiences, both professionally and personally.

I'm also very grateful to all of the people at the ETH Zurich for providing such a convenient, friendly and stimulating environment. My thanks goes to Daniel Bielser, Milena Brendle, Oliver Bröker, Manuela Cavegn, Nurhan Cetin, Oscar Chinellato, Daniel Cotting, Prof. Walter Gander, Bruno Heidelberger, Prof. Hans Hinterberger, Andreas Hubeli, Richard Keiser, Nicky Kern, Oliver Knoll, Rolf Koch, Sathya Krishnarmurthy, Doo Young Kwon, Edouard Lamboray, Claudia Lorch, Reto Lütolf, Matthias Müller, Martin Näf, Prof. Kai Nagel, Christoph Niederberger, Mark Pauly, Brian Raney, Martin Roth, Samuel Hans Martin Roth, Prof. Bernt Schiele, Christian Sigg, Thomas Sprenger, Oliver Staadt, Matthias Teschner, Julia Vogel, Michael Waschbüsch, Tim Weyrich, Martin Wicke, Stephan Würmlin, Matthias Zwicker.

In addition, I have to thank the employees from our industry partners VA Tech Hydro and Sulzer Hydro (the former Escher Wyss) for providing the industrial datasets which were used during this Ph.D. study. My special thanks goes to Mirjam Sick, Felix Muggli, Etienne Parkinson and Andreas Sebestyen for their friendly contact and advice.

Special thanks goes to Daniela Hall, Mie Sato and Filip Sadlo for their support of some parts of the implementation and for many enjoyable and fruitful discussions on scale-space theory, vortex hulls, FrameMaker and other important issues.

This work grew out of a long collaboration between the Computer Graphics Laboratory (CGL) of the ETH Zürich with our industry partners VA Tech Hydro (Zürich) and Sulzer Markets and Technology (Winterthur). The research project was funded by the Swiss Commission for Technology and Innovation (KTI) and by VA Tech/Sulzer under the project number 4917.1 KTS (October 2000 - September 2003).

CONTENTS

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

During the recent years, visualisation on digital computers has become an important part of scientific and engineering work, especially in application fields like medicine, the natural sciences, automotive and aircraft industry. In parallel to the continuously increasing computing power available, the amount of data to be processed has grown rapidly. As in the case of the internet, there is a growing gap between the quantity of information that is theoretically available and the ability to efficiently process and handle this information. The extraction of the essential contents of the information and its reduction to a reasonable degree thus becomes a more and more important task.

This also holds for the field of *flow visualisation*, where the underlying industrial datasets mostly are based on large computational grids originating from CFD simulations (Computational Fluid Dynamics). A grid of this type often contains millions of cells and nodes. The data stored on these structures usually are vector and scalar fields, often given for several hundred time steps in the case of unsteady flow. As a consequence of this, the raw data consume significant disk space, and it is for a human being not possible to inspect and analyse every detail of the flow.

Nevertheless there is a way out of this dilemma, since most of the flow details are not needed for understanding the general flow behaviour. The designers of industrial water turbines, for example, are mainly interested in special flow structures like *vortices* since these can cause undesired vibrations and resonance, which lead to reduced efficiency, enhanced material abrasion, and in the worst case to machine damage. The essential parts of the flow, which are called *features*, can be extracted from the flow data using special criteria and algorithms.

This dissertation investigates the essential features and behaviour of unsteady flow in the context of scientific visualisation and industrial application. The focus is on new feature-, region- and integration-based methods for selective visualisation of flow given in terms of vector and scalar fields on unstructured three-dimensional grids. The thesis also treats the fairly new idea to apply *scale-space* techniques to the field of flow visualisation.

Originally proposed in the context of image processing, scale-space techniques have been used for structured grids in the field of computer vision. However, their potential for flow visualisation was previously not exploited, maybe due to the major changes in implementation required for the unstructured grids mainly used in this field.

The goal of this thesis is to benefit from scale-space methods to improve the quality of *feature extraction* and *tracking*, which is performed in terms of *vortex core line* detection. Additionally, *vortex hulls* for unstructured grids are presented to enhance the visual perception of vortex sizes and shapes. Furthermore, a modified *particle tracer* is proposed which allows for concentrating on selected regions-of-interest (ROIs), thereby reducing occlusion problems, visual artifacts and storage requirements. Some of the results are also presented in a *virtual reality* environment and help our industry partners improve the design of turbomachinery parts, e.g. of water turbines.

The remainder of this chapter is structured as follows: Section 1.1 sets the context of this thesis by describing the flow visualisation process. Section 1.2 motivates the approaches presented in this thesis, whereas Section 1.3 shortly discusses some related work done in this area before. Our major contributions to the field are listed in Section 1.4, and Section 1.5 concludes this chapter with an outline of the dissertation.

1.1 FLOW VISUALISATION PIPELINE

The overall visualisation process can be seen as a pipeline consisting of four major phases: data input, filtering, mapping, rendering [HM90]. Figure 1.1 illustrates the process from gaining scientific flow data until rendering of the graphical results on a computer display. The following sections will discuss the individual stages of the pipeline in more detail.

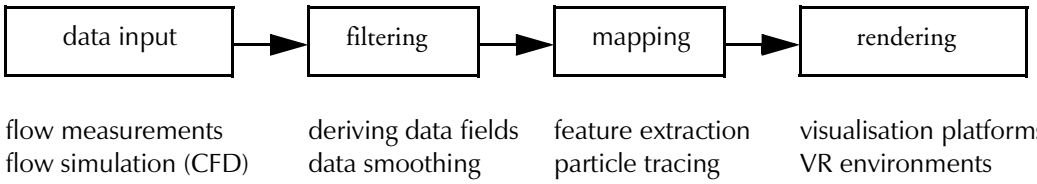


FIGURE 1.1 The flow visualisation pipeline with its four major phases.

1.1.1 Data Input

There are two basically different sources for the input data of the visualisation process, a physical and a virtual one. The first is to really establish a physical flow by constructing real-world models (e.g. a turbo-machine in a wind channel) and to measure its characteristic data (such as velocity and pressure), using sensors mounted at the test object (such as wing tips of an airplane or the draft tube of a water turbine). The alternative way is to do Computational Fluid Dynamics (CFD), i.e. to numerically simulate the flow on a digital computer by solving *Navier-Stokes* equations. In both cases, the results are produced for a predefined grid in the computational domain and stored as datasets on disk.

1.1.2 Filtering

The filtering step is in a sense the *preprocessing* of the actual visualisation (whereas from the CFD viewpoint, the visualisation step is often called *CFD postprocessing*, see [Bun89]). Filters convert a CFD dataset to another one by modifying its contents. Typical examples are: calculating derived fields from the input data fields (e.g. vorticity from velocity), building 2D slices from 3D data, or smoothing the input data using Gaussian filters.

1.1.3 Mapping

Mapping is the central part of the visualisation process. Its task is to produce geometrical primitives (e.g. points, lines or triangle meshes) which depict the essential information of the flow data. Typical examples are the computation of isosurfaces and contour plots from scalar fields, as well as feature extraction and particle tracing. Often the result data must be postprocessed (e.g. by computing connected components of triangle meshes).

1.1.4 Rendering

The rendering phase concludes the visualisation process by converting the 3D geometric primitives to a 2D image consisting of coloured pixels. Common visualisation platforms such as *AVS 5*, *AVS Express* or *Covise* offer sophisticated renderers for a variety of purposes, e.g. the *Covise* system includes a conventional renderer as well as a special renderer supporting virtual environments (see Chapter 8).

1.2 MOTIVATION

This dissertation mostly deals with the filtering stage and mapping stage of the flow visualisation pipeline. It particularly focuses on methods for selective visualisation of flow features like vortices, vortex hulls and particle movements in time-dependent CFD data given on unstructured three-dimensional grids.

As mentioned above, automatic extraction of features is a promising strategy to cope with the large amount of data produced by such time-dependent CFD simulations. The computational time for this type of simulation is typically in the order of days, which justifies the time spent on postprocessing the data by extracting features in a batch run. In fact, automatic feature extraction can achieve substantial data reductions while still permitting the visualisation of the essential parts of the flow. It can be used for visually browsing the results or in conjunction with other visualisation methods. However, the implementation of this strategy leads to the following problems:

1. Many flow features are of fractal nature, that means that there is no unique definition of their feature size. Depending on the observation scale, different sets of features can be observed. Usually, the scale is implicitly defined when an extraction method is designed. Instead, it would be preferable to let the user specify the scale interactively while viewing the result data.

2. Most methods require numerical computation of spatial derivatives, which is known to cause a roughening of the data. Smoothing the data can reduce this effect, but it is not trivial to find an appropriate smoothing kernel when dealing with irregular grids and highly varying cell sizes, as often occur in CFD datasets.

3. When features are extracted from time-dependent data, animating them can cause popping effects with features suddenly appearing or disappearing. These artifacts can be alleviated by incorporating temporal in addition to spatial smoothing.

As we will see in Chapter 4, *scale-space* techniques can successfully be applied to the field of computer vision, where feature extraction has already been practised for a long time. These techniques smooth the input data using Gaussian kernels, the standard deviation σ of which can be any positive real number. The hereby defined *scale axis* plus the spatial axes span the scale-space. Features thus not only have spatial extents but also a certain scale extent. They can therefore be searched and found in scale-space. This technique has the potential to solve the problems mentioned above also for the flow visualisation field.

The idea is now to combine scale-space methods with feature extraction in order to *track* features along time or, alternatively, along scale. Whereas the first scenario is obvious, the second one should help identify fragmented features while still keeping their positional accuracy. Furthermore, the quality of the results should be improved, because estimating derivatives of the flow fields needs no extra filtering due to the presmoothed data at higher scale levels. Furthermore, subsetting of the resulting vortex core lines should be automated and need fewer heuristic parameters than in previous implementations (see Chapter 5).

The concept of feature extraction is part of the more general task of *selective visualisation* of flow. A variety of vortex detection criteria can be found in literature, which comprise the definition of line-type features like vortex core lines and surface-type features like vortex hulls. Also, a stream of mass-less particles in a flow can help identify its regions-of-interest. All these paradigms can be applied to steady and time-dependent flow likewise.

Besides the feature tracking system discussed above, we want to design a method for adding vortex hulls to the extracted vortex core lines, which in contrast to other approaches is capable to operate on unstructured grids (see Chapter 6). The idea is to combine the advantages of line-type features (clear separability of different vortices) with those of region-type features (better notion of vortex shape and size). The new method should permit the choice of different scalar fields and thresholds to define the boundary of the vortex, as well as filtering and fairing operations to smooth the resulting tubes.

Furthermore, we want a modified particle tracer which reduces the number of particles to be processed by concentrating on certain regions-of-interest (see Chapter 7). A cell classification scheme and a special particle seeding scheme will help keeping the particle density constant, while blending operations will allow smooth transitions at the boundaries.

In addition to conventional rendering and displaying techniques, *virtual environments* are becoming more and more popular in science and industry. VR technologies enhance the visual and acoustic perception by use of new input and output devices, such as laser swords, 3D glasses, head trackers, large projection screens and stereo sound. They provide a way to "immerse" into a 3-dimensional graphical scene and thus to better explore and understand the structure of the visualised data.

The field of visualisation can also benefit from these techniques and is therefore supported by VR development packages from current visualisation platforms. We will in Chapter 8 present a framework which allows to interactively explore results from our visualisation methods in such a virtual environment. Since the filtering and mapping stage of the flow visualisation pipeline takes significant time, the approach is to store the results to disk and transferring them to the VR system, which can render and display them at comfortable frame rates.

1.3 RELATED WORK

This section summarises some important work which was done in the context of flow visualisation prior to this dissertation. More detailed descriptions of the referred papers will be given in the appropriate chapters later in this thesis.

The general field of visualisation was first defined in 1987 by McCormick, DeFanti and Brown in their groundbreaking NFS report [MDB87], which stated the necessity of and showed the direction for further research. The vast application field of flow visualisation was described in detail in 1989 by Buning [Bun89] and later by Post, Laramee et al. in several articles [HPvW94, PvW94, PVH+03, LHD+04].

Multi-scale and multi-resolution data representations had already been used during the 1970s [Kli71], but without solving the problem of relating features of different scales to each other. The foundations of the scale-space theory were then established in the 1980s by Witkin [Wit83, WT83] and Koenderink [Koe84]. Originally restricted to image processing and vision, the theory was in the 1990s extended by Florack, Romeny et al. [FtHRKV92] and Lindeberg [Lin94], who gave a thorough compendium of scale-space theory and application fields at that time.

Nonlinear scale-spaces based on anisotropic diffusion were used by Perona and Malik [PM87] for edge detection and by Diewald et al. [DPR00] for visual exploration of vector fields. Our goals are, beyond visual exploration, to *algorithmically* extract features as geometric objects and to improve this process by exploiting the multi-scale nature of the features. For this purpose, we will use isotropic linear scale-spaces, since they better fit to our requirements.

Luerig, Westermann et al. [LGE97, WE97] performed feature extraction from volumetric data in scale-spaces based on wavelets. However, to keep the spatial resolution also at higher scale levels, we will not construct a multi-resolution pyramid. Also, such a pyramid would yield a too coarse sampling along the scale axis. For the purpose of feature tracking, it is preferable to have no prescribed sampling.

Different vortex detection methods were proposed by Levy et al. [LDS90], Sujudi/Haimes [SH95b, SH95c], Banks/Singer [BS94, SB94], Miura/Kida [MK96], and Strawn/Kenwright/Ahmad [SKA98, SKA99]. Based upon their line-type definitions of a vortex, Roth and Peikert [RP99, Rot00] proposed the parallel vectors operator, a more general scheme which all of the above vortex criteria can be embedded in by formulating them in terms of two vector fields. However, all these approaches only solve the problem for *steady* flow fields in 3D. Our goal is to also *track* vortices over time or scale using a 4D extension (published in [BP02b]) of the parallel vectors algorithm.

Feature tracking was performed e.g. by Silver et al. [SZF+91, SW96] and Reinders et al. [RPS99, Rei01]. Most of the existing tracking methods operate on features extracted from 3D grid cells in single datasets. In contrast, our approach is to extend the grid cells by one dimension and to extract the features from these lifted cells, which are in our case 4D hypercubes. This reflects the same lifting principle as the Marching Cube algorithm uses for isosurface extraction in 3D, a problem which was originally regarded as tracking 2D contour plots along the z axis of the 3D grid. An advantage of our feature tracking method is that it needs no heuristics like spatial overlap [SW96] or shape attributes [RPS99], while it can still handle features moving by more than one grid cell per dataset.

Vortex hulls have been treated by Zabusky [ZBP+91], Banks and Singer [BS94], Sadarjoen [Sad99], Roth [Rot00] and recently by Garth et al. [GTS+04]. Most approaches regard the vortex hull as a deformable model (see Terzopoulos and Fleischer [TF88]) which has a centerline and is extended until a certain condition is met, for instance when the pressure becomes too large. However, many implementations are based on rectilinear or structured grids. We will also stick to the deformable model principle but develop a method working on unstructured grids and for termination criteria based on different scalar fields. Our extension principle is more sophisticated than the one previously presented in [BP02a] and therefore needs less computational time.

Particle tracing is actually a well-known standard technique (see the description by Sadarjoen et al. [SvWHP97]). Nevertheless, it is hard to find better techniques for time-dependent vector fields. One of the key problems is to maintain the particle density over time. For streamline integration, streamline placement techniques were introduced by Turk and Banks [TB96]. Our approach [BP02a] takes advantage of the mass conservation property of physical flow fields and uses a quasi-random particle seeding scheme (see [SH95, SMA00]) in conjunction with a buffer cell mechanism. The restriction to a certain region-of-interest surrounded by a ring of buffer cells saves significant computational effort. Based upon two scalar thresholds, a smooth blending of particles entering and leaving the ROI is achieved.

For our VR application, we chose the *Covise* visualisation platform developed at the University of Stuttgart [HLR04] and now being distributed by the spin-off company *VisEnSo* [VIS04]. The VR renderer contained in this package can be enhanced by user-programmed plugins [VIS03], which enabled us to transfer our visualisation results into a virtual environment for interactive exploration and enhanced visual perception.

1.4 CONTRIBUTIONS

The goal of this thesis is to investigate the essential features and behaviour of unsteady flow in the context of flow visualisation and industrial applications. The focus is on new feature-based, region-based and integration-based methods for selective visualisation of 3D flow given in terms of vector and scalar fields on unstructured three-dimensional grids. Scale-space techniques known from computer vision improve the quality of the feature extraction. The results are also presented in a virtual reality environment and help our industry partners improve the design of turbomachinery parts, e.g. of water turbines. The main contributions of this thesis are:

- an application of scale-space techniques to the field of flow visualisation,
- a computational scheme for constructing the scale-space on unstructured CFD grids,
- an algorithm for extracting and tracking vortex core lines over time and scale,
- a scalar-field based method for constructing vortex hulls around core lines,
- a particle tracer for selective visualisation of regions-of-interest in flow fields,
- a framework for interactive visualisation of flow features in virtual reality.

1.5 OUTLINE

The remaining chapters of this thesis are organised as follows:

- **Chapter 2** gives a brief overview on the fundamentals of visualisation in general and of flow visualisation in particular. After a presentation of several different ways to categorise visualisation, the most important grid and data types used in flow visualisation are discussed. The definitions given here are fundamental for the flow visualisation techniques described in the following chapters.

- **Chapter 3** gives a short overview on several basic techniques for flow visualisation, most of which were used as ingredients for the algorithms and methods presented in the later chapters of the thesis. A few representatives of the flow visualisation categories mentioned in the previous chapter are discussed, namely of the direct, integration-based, region- and feature-based flow visualisation techniques.

- **Chapter 4** gives an overview of some important data representations dealing with different levels of resolution or scale. It presents the scale-space representation and its special properties, along with some application examples from image processing and computer vision. The use of scale-space methods for flow visualisation is also motivated. At the end of the chapter, several methods for computing the scale-space representation on different grid types are discussed and compared, and a newly developed method for computing the scale-space on unstructured grids is presented.

- **Chapter 5** recapitulates the parallel vectors operator, a general vortex core line extraction algorithm introduced by Roth and Peikert [RP99], and describes the adaptations and simplifications to be made in the context of a scale-space analysis. It is then shown how a 4D extension of this algorithm is able to track vortices in either the temporal or in the scale domain. Finally, both numerical and visual results are given.

- **Chapter 6** presents an implementation of vortex hulls for vortex core lines on unstructured grids, which allows to select from different hull shapes and scalar fields, to define suitable thresholds, and to smooth the surfaces of the constructed tubes. The combination of line-type features (vortex core lines) and region-type features (vortex hulls) leads to plastic illustrations of vortex extents and still clearly separable features.

- **Chapter 7** explores techniques for visualising selected flow structures in time-dependent data by use of a modified, sophisticated particle tracer. A particle seeding scheme based on quasi-random numbers is proposed, which minimises visual artifacts such as clusters or patterns. By constraining the visualisation to a region of interest, occlusion problems are reduced and storage efficiency is gained. The primary industrial application is the visualisation of the *vortex rope*, a rotating helical structure which builds up in the draft tube of a water turbine. In two related applications, the cavitation regions near the runner blades of a Kaplan turbine and a water pump are visualised.

- **Chapter 8** presents a framework for transferring results of the vortex core extraction and vortex hull construction to a virtual environment. A file system based approach is used to decouple the rendering phase from the computationally expensive data acquisition and feature extraction procedures. As a consequence of this, the VR application is able to efficiently load and handle the vortex result data. The user can thus navigate through the result datasets at comfortable frame rates and interactively select, mark and remove vortex structures from the screen.

- **Chapter 9** concludes the dissertation with a summary of the main contributions, a discussion of advantages and drawbacks of the presented approaches, and some ideas for future research.
- **Appendix A** contains coloured versions of some result images shown in the previous chapters, especially concerning feature extraction and tracking, vortex hulls, the VR application, and selective particle tracing.
- **Appendix B** contains all literature references of this thesis.
- **Appendix C** contains some more detailed explanations on how the vortex hulls of Chapter 6 were constructed. The focus is on the problem of intersecting a given ray with a quadrangle face of a hexahedral cell in 3D space, which can be planar or non-planar. Due to the discrete nature of the underlying unstructured grids and due to noise contained in some of the raw industrial datasets we investigated, a number of nontrivial numerical problems had to be solved in addition to the theoretical procedure.
- **Appendix D** contains a curriculum vitae of the author of this thesis.

CHAPTER 2

BASICS OF FLOW VISUALISATION

Flow visualisation has been an important and interesting part of visualisation for decades. In this chapter, we give a brief overview on the fundamentals of visualisation in general and of flow visualisation in particular. After presenting several different ways to categorise visualisation, we will discuss the most important grid and data types used in flow visualisation. The definitions given here are fundamental for the flow visualisation techniques described in the following chapters.

2.1 CLASSIFICATION

Although visualisation has a long history, its meaning for scientific work was not fully recognised until the late 1980s. In their pioneering NSF report on visualisation in scientific computing [MDB87], McCormick, DeFanti and Brown gave a general definition of the subject and pointed out the need and direction for further research. Since the general field of visualisation, as well as that of flow visualisation, offers a wide range of methods and applications, no trivial or even canonical classification of the subject exists [HPvW94]. Also, the special conditions of any application field have a strong influence on the suitability of each visualisation method, and thus on the preferred choice. In this section, we will present several possible criteria for a categorisation of visualisation.

2.1.1 Application field

The perhaps most intuitive criterion is the application field where data are to be visualised. A coarse differentiation can be made between *information visualisation* and *scientific visualisaton*. The first one mostly deals with discrete and higher-dimensional data, which is often stored in tables or relational databases and does not have a well-defined metrics. Typical methods of information visualisation include chart diagrams, parallel coordinates, scatter plots and perspective walls. A wide selection of business applications are available which cover the most important needs of this area.

This thesis concentrates on the other type, scientific visualisation, and one of its most common subareas, flow visualisation. Typical applications of flow visualisation can be found in (but are not restricted to) aerodynamics, automotive industry, geology, medicine, meteorology, and turbomachinery design.

2.1.2 Gaining data for visualisation

In this chapter, we use the term "flow visualisation" for what is actually *computational* flow visualisation. This means that the datasets to be visualised were created by numerical simulations based on physical models, e.g. using the *Navier-Stokes* equations. The simulated flow data (especially the velocity and pressure data) are stored in datasets and serve as a basis for various computational methods to visualise the flow. In this thesis, we will mainly refer to this type of gaining and processing the information contained in a flow.

However, it should be kept in mind that flow visualisation can also be done the "good old way" using flow experiments and empiricism. One example is the use of test rigs for reduced scale models of existing turbines, as is done by our industry partner VA Tech Hydro, the former Escher Wyss (a size comparison of test model and real counterpart of a Francis runner is shown in Figure 2.1). The flow behaviour within these models can for instance be observed and recorded using high-speed cameras and stroboscope lighting. Another possibility is to inject dye into a flow and to photographically record the movement and distribution of the coloured water particles.

.

FIGURE 2.1 Original and 1:10 scale test rig model of a Francis runner (image courtesy of VA Tech Hydro, Zurich). See also Colour Figure A.1 on page 121.

Although numerical simulation and computational visualisation give better insight into the flow behaviour and help reduce the developing costs of technical devices (such as machine parts of a water turbine), *experimental* and empirical flow visualisation are still indispensable for proving the theoretically computed results. For cost reasons, such flow

experiments are only performed for turbomachinery designs which have in advance been optimised using computational visualisation methods.

2.1.3 Level of data abstraction

Visualisation methods can also be categorised according to the effort they require in comparison to other steps of the visualisation pipeline. Post, Laramee et al. distinguish between three fundamentally different types of visualisation methods, namely *direct*, *integration-based*, and *feature-based* visualisation methods [PVH+03, LHD+04]. Generally, it can be stated that the computational complexity of a method rises with its level of data abstraction (see Figure 2.2). On the other hand, a more sophisticated visualisation method leads to more physically meaningful results, which reduces the amount of data to proceed with.

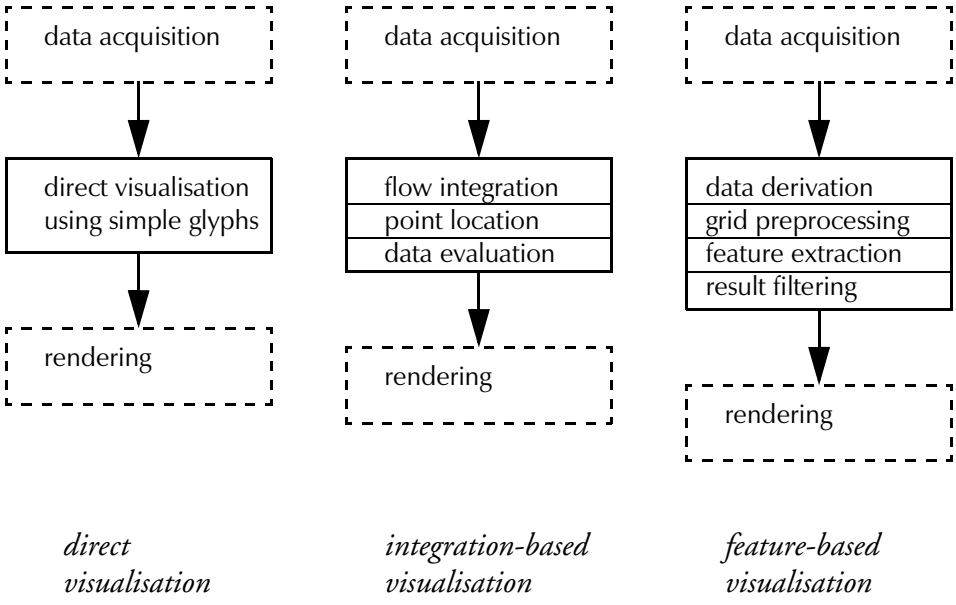


FIGURE 2.2 The actual flow visualisation step in the visualisation pipeline.

In the simplest case, the user is interested in a quick, intuitive, and *direct* visualisation of the data. This can often be achieved by relatively fast and simple methods, e.g. the velocity data of a flow can easily be drawn as an arrow plot if the flow is 2-dimensional (for 3 dimensions, the case is more difficult to handle due to perception and occlusion problems). A similar method is the use of hedgehog plots as suggested by Klassen and Harrington [KH91] and implemented in *The Visualisation Toolkit (VTK)* by Schroeder, Martin and Lorensen [SML03]. For direct visualisation of 3-dimensional flows, volume rendering is a good technique to cope with the increased occlusion problems. In medical applications, which mostly use Cartesian grids, volume rendering is very common. However, it is not trivial to adapt these methods to unstructured grids, which are mostly used in flow visualisation. A possible solution is to resample the unstructured grid to a Cartesian one, as is suggested by Westermann [Wes01].

More abstract than direct visualisation are *integration-based* methods, since they use integral objects as a counterpart to the derivative nature of simulation-based flow data (e.g. velocity gained from a CFD simulation). Using integration methods, it is possible to reconstruct visualisation-relevant data (such as the position of a particle, or streamlines) from the simulation data.

The third type is the *feature-based* approach, which performs an additional abstraction step by extracting certain phenomena (such as vortices) or topological information (like critical points) from the original flow data. The extraction step is in general computationally expensive, but this drawback is compensated by an enormous reduction of the amount of data. Since the original data are not needed for the final visualisation of the features, a data reduction rate of up to 1:10000 is possible ([Ken98]). In Chapter 5, we will present feature-based visualisation methods for extracting vortices from a flow.

In addition to the three cases mentioned above, we can also define a fourth type, namely *region-based* visualisation. In contrast to many feature-based methods which rely on locally defined criteria (such as critical points), one can also define a region-of-interest (ROI) by a more global criterion. For example, it is possible to set a threshold for a scalar field defined on the total computational domain, which leads to isosurfaces as resulting structures of the visualisation step. In Chapter 7, we will deal with integration-based as well as region-based visualisation methods, and present a modified particle tracer which combines techniques of both types.

2.1.4 Spatial dimensionality of grid and data

In most application fields of visualisation, 1-, 2- or 3-dimensional grids are common. For example, image processing assumes a 2D uniform data structure inherent in digital images, and uses the integer row and column index for addressing a certain pixel. These indices can also be interpreted as x- and y-coordinates determining the physical location of the point on a Cartesian grid (see Section 2.2).

For medical applications (and also in the field of vision), often 3D "images" are used, which are actually stacks of 2D images. A typical example is a computer tomography (CT), magneto-resonance (MR) or positron-emission (PET) dataset [MCS02], which mostly consists of more than 100 two-dimensional *slices* (each slice representing a cross-section through the body of a patient [EW92]). By regarding the direction perpendicular to the slices as the z-dimension of the Cartesian grid, the stack of slices builds a volumetric dataset which can be explored for structures such as bone, tissue and vessels.

For turbomachinery design, 3-dimensional unstructured grids (created using CAD tools) are predominant, for the possibility of representing arbitrary shapes of machine parts (such as the runner and draft tube of a water turbine). The data defined on these grids are at least 3- or even higher-dimensional (see Section 2.3).

2.1.5 Temporal dimensionality of data

Another important point is whether the flow is steady or unsteady (time-dependent). In the latter case, the data are often given for several hundreds of time steps, leading to serious storage and processing issues. Solutions often use special techniques to cope with the amount of data, such as out-of-core rendering [USM97] and rendering of compressed data [YMC00]. In Chapter 5, we will present a method for tracking vortex core lines over time, which uses a sliding activity window to access the datasets for different points in time.

2.2 CELL AND GRID TYPES

Flow data are in general not given as continuous analytic functions (except for artificially constructed theoretical examples) but as discrete datasets, which serve as an input for the visualisation pipeline. The datasets mostly originate from CFD simulations based on Lagrangian-type or Eulerian-type representations of the Navier-Stokes equation. Lagrangian methods comprise *smooth particle hydrodynamics (SPH)* mainly used in astrophysics [GM77], as well as *vortex methods* (see Cottet and Koumoutsakos [CK00]) based upon vorticity of moving fluid elements. In contrast, Eulerian methods deal with velocity and pressure data at fixed points on a *grid*. We will in the following only treat visualisation methods based on Eulerian flow fields given on such computational grids.

A variety of different grid types has been established, mostly due to the needs of specific application fields (see Nielson et al. [NSR90, NHM97]). Common to all grid types is that they store data (vectors or scalar values) at distinct point locations in physical space, which are called *nodes*. Neighbourhoods are defined through *edges*, every edge being a line segment directly connecting two nodes.

There exist also alternative point representations like *point clouds*, which have been successfully investigated and implemented at the ETH Zurich within the scope of the *PointShop3D* project by Pauly et al. (see the PointShop3D website [PZK+03] and [PG01, PGK02, PKG03, PKKG03, Pau03]). The point cloud representation stores only isolated points without connecting edges, so their is no a-priori defined neighbourhood of a certain point (see Figure 2.3 left). When neighbourhood information is needed for a sample point of the cloud (e.g. for estimating its surface normal), a neighbourhood can still be defined ad hoc, for instance by choosing the k cloud points which lie closest to the sample point. Alternatively, a point cloud can be transformed into a triangular *mesh* (which is a special case of a grid) using surface reconstruction methods (see Figure 2.3 right).

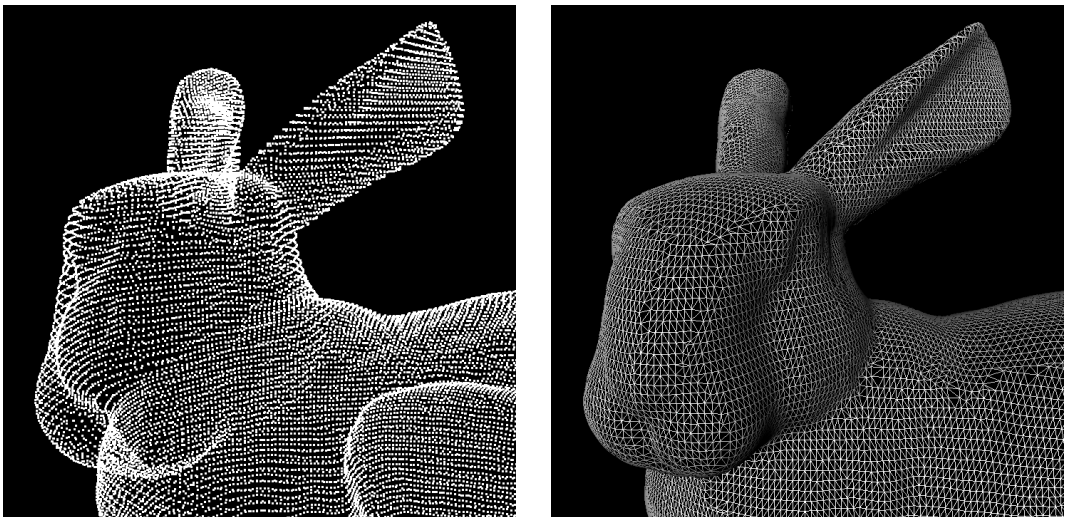


FIGURE 2.3 Point cloud versus grid (in this case, a triangular mesh) for the Stanford bunny model.

The smallest organisational unit of a grid is a called a *cell*. Usually, a cell contains a small number of nodes with their connecting edges. As the nodes are connected by edges, so the edges build the boundary of the *faces*, provided that the cells are 3-dimensional. In 2-space,

polygonal meshes are predominant, the cells mostly being triangles or quadrangles. In 3-space, mainly tetrahedral and hexahedral cells are used (see Figure 2.4).

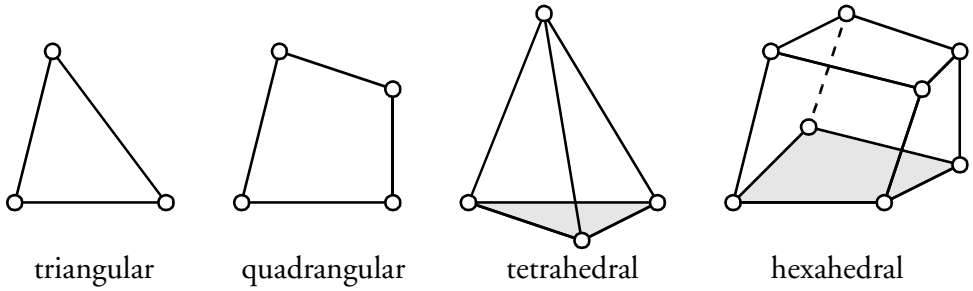


FIGURE 2.4 Different cell types used in visualisation grids (bottom faces are shaded).

A coarse classification of grids divides them into *structured* and *unstructured* grids. The fundamental difference between them is that structured grids consist of only one cell type, and that their nodes are arranged in an array. Therefore an arbitrary node index directly yields the indices of the neighbour nodes by simply incrementing or decrementing the index. As a further consequence, the number of neighbours of any inner node is constant (e.g. 4 neighbours in a 2D uniform grid with rectangle cells). Structured grids thus *implicitly* contain their connectivity (neighbourhood) information, which reduces the storage requirements as well as computational times and theoretical complexity of algorithms working on them.

The simplest and computationally easiest to handle case is of a structured grid is a *uniform* grid, which can be *Cartesian* or *skew-angled*. In the uniform case, the points are equally distributed along every dimension of the grid, thus a simple linear transformation exists between the node indices (i, j, k) and the physical node coordinates (x, y, z). It is therefore not necessary to explicitly store the physical coordinates of the grid nodes - the grid extents and constant distances between two successive nodes are sufficient for accessing all grid information and data.

Slightly more difficult to handle is a non-uniform but still structured grid (also called *curvilinear* grid), which is topologically equal to uniform grids, but the points are not equidistant. It thus requires to store a *node list* containing the coordinates for every grid node. However, the connectivity information is still implicitly available, as in the uniform case.

The class of *unstructured grids* is characterised by the fact that they do not possess a regular topology. Also, the cells can be of varying types (i.e. tetrahedral and hexahedral cells mixed within a 3-dimensional unstructured grid). Even if all cells are of the same type, the number of neighbouring nodes can vary also for the inner nodes, as can clearly be seen in the lower right picture of Figure 2.5.

There exist also some hybrid grid types, for example *block-structured* grids. These are unstructured compositions of structured sub-grids (as shown in Figure 2.5, bottom left). In the following chapters, we will only treat the case of *pure unstructured* grids, which lack any type of implicit structure. They build the most general type of unstructured grids and therefore also comprise the block-structured ones.

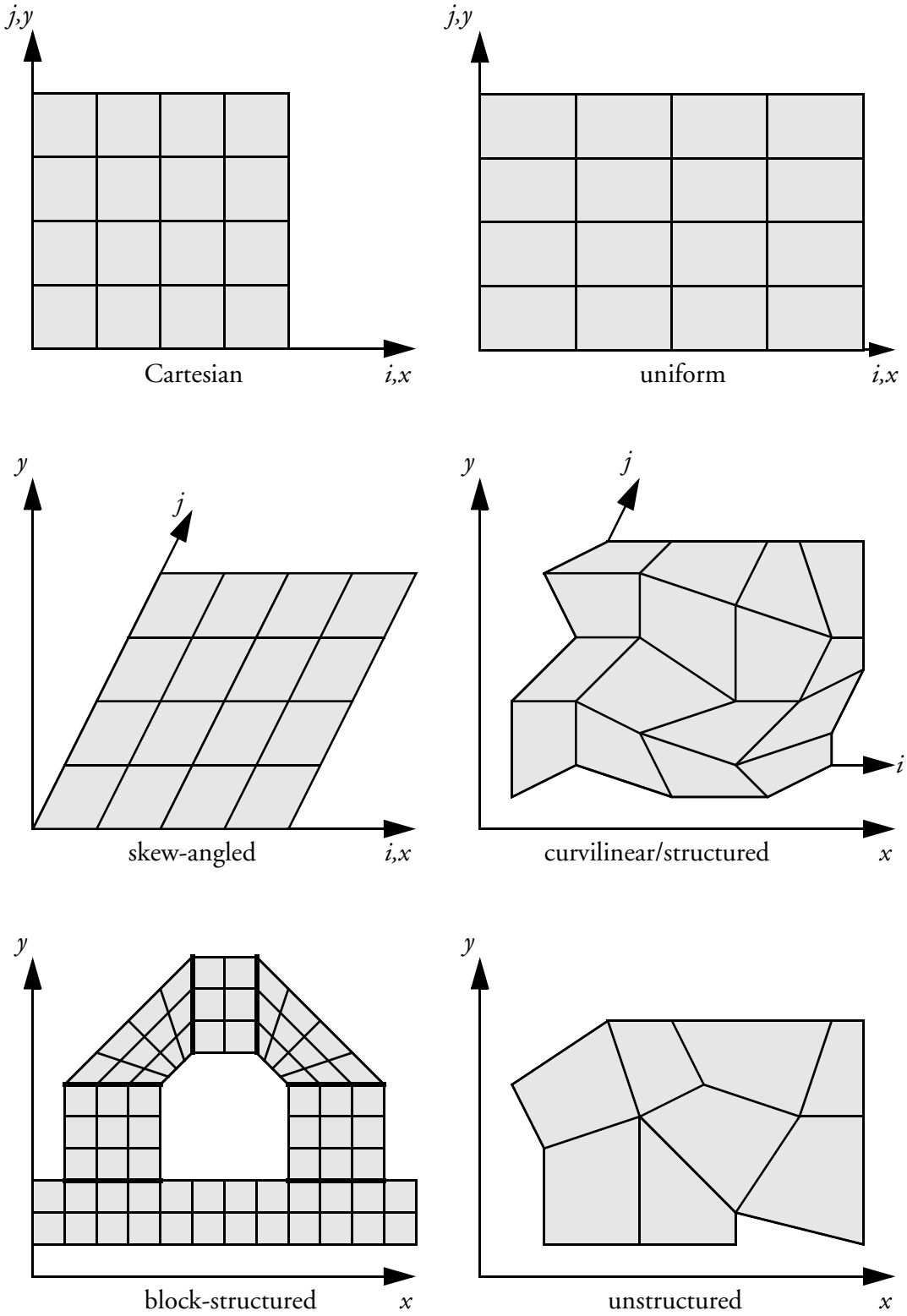


FIGURE 2.5 Different grid types used in visualisation (2D representation, cells are shaded).

2.3 DATA FIELDS

A CFD (Computational Fluid Dynamics) dataset is usually defined on a 2- or 3-dimensional grid and contains an n-dimensional data vector of real numbers for every grid node. The sources for these n-tuples are also called *data channels*. Some of the values are just scalar, some of them are defined as components of vectors, such as the velocity of the flow. In the following, we give a short overview of the most important types of vector fields and scalar fields originating and derived from CFD datasets. We will refer to these definitions when explaining flow visualisation techniques in the next chapter.

2.3.1 Vector Fields

Since CFD datasets are generally computed using physical models like the Navier-Stokes equations, the most common vector field stored for each node is the *velocity field*. Until further notice, we will assume that the flow is *steady*, i.e. the CFD dataset is given for only one point of time and the flow can be regarded as constant. Furthermore, we will assume a discrete 3-dimensional computational grid. The flow fields are then defined in a continuous 3-dimensional Cartesian xyz coordinate system, regardless of the fact whether the underlying grid is structured or unstructured. The *velocity* $\boldsymbol{v}$ of the flow at an arbitrary location (x, y, z) in computational space is defined component-wise as

$$\boldsymbol{v}(x, y, z) = \begin{bmatrix} v_x(x, y, z) \\ v_y(x, y, z) \\ v_z(x, y, z) \end{bmatrix} \qquad (x, y, z \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^3) \quad , \tag{2.1}$$

where each component marks the flow velocity in the corresponding spatial direction. If the point (x, y, z) is one of the grid nodes, its velocity and other data fields are directly given in the dataset. However, if the point (x, y, z) lies in the interior of a grid cell, an interpolation scheme is necessary to evaluate the data fields at this point. On 2-dimensional grids, linear interpolation is common for triangular cells and bilinear interpolation for quadrangle cells. On 3-dimensional grids, mostly linear interpolation is used for tetrahedral cells and trilinear interpolation for hexahedral cells. In the following, we will only write the vector and scalar field variables and omit the (x, y, z) coordinates indicating that the data are defined as a continuous field rather than only on the grid nodes.

Each of the three velocity field components (v_x, v_y, v_z) defines a scalar field, the gradient ∇ of which can be computed as the vector of its first derivatives in every spatial direction. The *velocity gradient* tensor $\nabla\boldsymbol{v}$ is defined as the Jacobian of the velocity, that is the combination of the gradients for every of the three velocity field components:

$$\text{grad}\ \boldsymbol{v} = \nabla\boldsymbol{v} = \begin{bmatrix} \text{grad}\ v_x \\ \text{grad}\ v_y \\ \text{grad}\ v_z \end{bmatrix} = \begin{bmatrix} (\nabla v_x)^T \\ (\nabla v_y)^T \\ (\nabla v_z)^T \end{bmatrix} = \begin{bmatrix} v_{x,x} & v_{x,y} & v_{x,z} \\ v_{y,x} & v_{y,y} & v_{y,z} \\ v_{z,x} & v_{z,y} & v_{z,z} \end{bmatrix}. \tag{2.2}$$

Note that this is not the Hessian matrix of the second derivatives of the velocity field - the notation $v_{x,x}$ means that the v_x component was once derived along the x direction, so it is only a first derivative.

The computation of spatial derivatives is not trivial for a data field which is not analytically given, as is the case on a discrete grid. A discretised differentiation operator is necessary, the complexity of which strongly depends on the grid type. For a uniform grid, a simple operator like the *symmetric difference* operator [Ste93] is sufficient, which is stored in a 3×3 matrix if the grid is 3-dimensional.

For an unstructured grid, however, symmetric differences are no suitable way since the number of neighbouring nodes is not constant for every node, and the distances between the nodes also differ. One possibility is to use a weighted difference operator on a k-neighbourhood of each node. Such a scheme can also be applied when computing higher derivatives such as the Laplacian, as was proposed by Taubin [Tau95] and refined by Desbrun et. al. [DMSB99]. They use an *umbrella operator* for smoothing triangular surface meshes, which we will apply to the postprocessing of vortex hulls later in Chapter 6.

Alternatively, a polynomial can be fit into the neighbourhood of each node to compute its spatial derivatives. Since this method is computationally quite expensive, we chose a least-square-fit among the one-neighbourhood of a node. This method is relatively easy to implement since the one-neighbours are directly accessible through the edge information of the grid. In Chapter 5, we will see that the accuracy of the method is sufficient even for higher-order derivatives, provided that the underlying datasets could be presmoothed.

The *vorticity* vector ω indicates the local rotation of the flow and is defined as the *curl* of the velocity vector. Its length is a measure for the rotational speed, whereas its direction shows the rotational axis (assuming that the rotation is right-handed). For the vorticity computation, six out of the nine components of the velocity Jacobian are needed:

$$\omega = \nabla \times \boldsymbol{v} = \begin{bmatrix} v_{z,y} - v_{y,z} \\ v_{x,z} - v_{z,x} \\ v_{y,x} - v_{x,y} \end{bmatrix}. \tag{2.3}$$

The velocity Jacobian is also contained in the *acceleration* $\boldsymbol{a}$ which each particle in the flow experiences:

$$\boldsymbol{a} = (\nabla \boldsymbol{v})\boldsymbol{v} + \frac{\partial \boldsymbol{v}}{\partial t} = \begin{bmatrix} v_{x,x} & v_{x,y} & v_{x,z} \\ v_{y,x} & v_{y,y} & v_{y,z} \\ v_{z,x} & v_{z,y} & v_{z,z} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \frac{\partial \boldsymbol{v}}{\partial t}. \tag{2.4}$$

In contrast to the velocity, the pressure p is no vector but only a scalar field, thus the *pressure gradient* ∇p w.r.t. the three spatial directions is a vector written as

$$\text{grad } p = \nabla p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T. \tag{2.5}$$

The *pressure Hessian* matrix $\nabla(\nabla p)$ contains the second derivatives of the flow pressure:

$$\nabla(\nabla p) = \begin{bmatrix} p_{xx} & p_{xy} & p_{xz} \\ p_{yx} & p_{yy} & p_{yz} \\ p_{zx} & p_{zy} & p_{zz} \end{bmatrix}. \tag{2.6}$$

It is also possible to multiply the Hessian and the gradient of the pressure, resulting in a vector field which we will call the *pressure Hessian times pressure gradient* $(\nabla(\nabla p))(\nabla p)$. This vector field will be needed for the Miura/Kida definition [MK96] of a vortex (see Chapter 3):

$$(\nabla(\nabla p))(\nabla p) = \begin{bmatrix} p_{xx} & p_{xy} & p_{xz} \\ p_{yx} & p_{yy} & p_{yz} \\ p_{zx} & p_{zy} & p_{zz} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}. \tag{2.7}$$

2.3.2 Scalar Fields

The most common scalar field contained in CFD datasets is the *pressure field*. For any node of the CFD grid, the pressure has been measured or computed by a flow simulation, as has been for the velocity. In combination with a suitable interpolation scheme, one can define the *pressure* p for an arbitrary point location (x, y, z):

$$p(x, y, z) \qquad (x, y, z \in \mathbb{R}, p \in \mathbb{R}) \quad . \tag{2.8}$$

Another important scalar field is the *helicity* h, which is the inner product of two vectors, namely the velocity and the vorticity (which itself is derived from the velocity):

$$h = \omega \bullet \boldsymbol{v} = \begin{bmatrix} v_{z,y} - v_{y,z} \\ v_{x,z} - v_{z,x} \\ v_{y,x} - v_{x,y} \end{bmatrix} \bullet \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}. \tag{2.9}$$

There is also a different version called the *normalised helicity* $\bar{h}$, based upon the normalised velocity and vorticity $(\hat{\boldsymbol{v}}, \hat{\omega})$ rather than the original vectors. Since the dot product of two normalised vectors is equal to the cosine of the angle enclosed by them, the normalised helicity field indicates for every location "how parallel" the velocity and vorticity are:

$$\bar{h} = \hat{\omega} \bullet \hat{\boldsymbol{v}} = \frac{\omega \bullet \boldsymbol{v}}{|\omega| \cdot |\boldsymbol{v}|} = \cos\angle(\omega, \boldsymbol{v}). \tag{2.10}$$

Roth and Peikert [Rot00] define two additional scalars which are useful for estimating the relevance of a detected vortex structure (namely a *vortex core line*, see Section 3.4.2). They presented the *parallel vectors operator*, which postulates that two certain vectors should be parallel or antiparallel on such a vortex core line. For a point on a vortex core line computed by the parallel vectors operator, the *feature quality* is defined using the angle between the velocity and the tangent to the core line at this point (see Section 5.2.5 for further explanations).

The *vortex strength* is an indicator of how fast the flow locally rotates around a certain point, e.g. on a vortex core line. For a 2-dimensional flow, a vortex occurs as a critical point in the 2D plane, forming a spiral pattern. In this case, the 2×2 matrix of the velocity gradient (which is the Jacobian of the 2D velocity field) has two conjugate complex eigenvalues, the absolute value of the imaginary part of which is the vortex strength.

Since the flow is swirling around the vortex core line also in 3D, a spiral pattern is to be expected in the 3D case, too. But the spiral will occur in a plane which is perpendicular to the core line (see Figure 2.6). The core line direction and the velocity at the core line are almost equal (the feature quality is near one on the core line), so the velocity direction can be chosen for an approximation of the rotational axis.

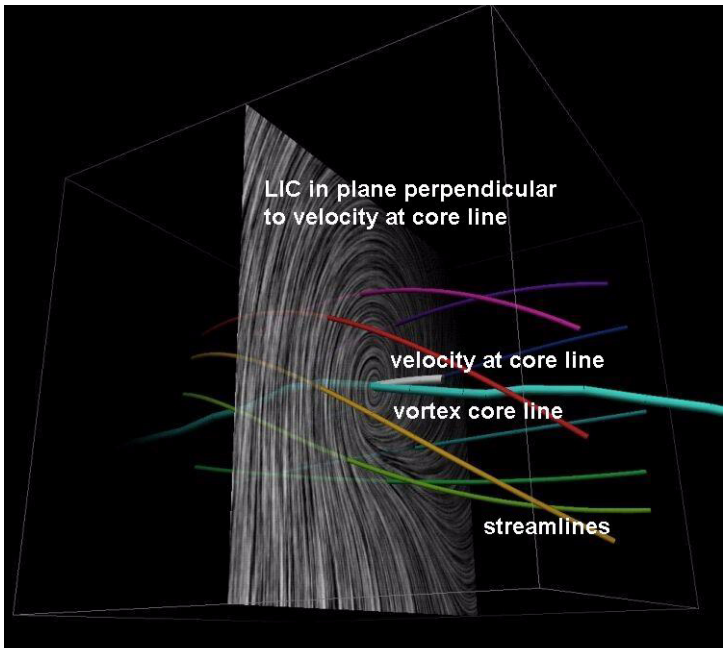


FIGURE 2.6 Definition of vortex strength in a plane perpendicular to the core line (vortex visualised by streamlines and LIC method, see Section 3.2). Image courtesy of ETH Zurich.

The velocity can now be projected to a plane perpendicular to the velocity direction by using the velocity gradient (Jacobian) of the 3D domain. If $\boldsymbol{f}$ and $\boldsymbol{g}$ are normalised basis vectors of the perpendicular plane, and

$$\tilde{\boldsymbol{f}} = (\nabla \boldsymbol{v})\boldsymbol{f} = \begin{bmatrix} v_{x,x} & v_{x,y} & v_{x,z} \\ v_{y,x} & v_{y,y} & v_{y,z} \\ v_{z,x} & v_{z,y} & v_{z,z} \end{bmatrix} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \tag{2.11}$$

$$\tilde{\boldsymbol{g}} = (\nabla \boldsymbol{v})\boldsymbol{g} = \begin{bmatrix} v_{x,x} & v_{x,y} & v_{x,z} \\ v_{y,x} & v_{y,y} & v_{y,z} \\ v_{z,x} & v_{z,y} & v_{z,z} \end{bmatrix} \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix}, \tag{2.12}$$

then the projected velocity matrix computes to

$$A = \begin{bmatrix} \boldsymbol{f} \bullet \tilde{\boldsymbol{f}} & \boldsymbol{f} \bullet \tilde{\boldsymbol{g}} \\ \boldsymbol{g} \bullet \tilde{\boldsymbol{f}} & \boldsymbol{g} \bullet \tilde{\boldsymbol{g}} \end{bmatrix}, \tag{2.13}$$

and the absolute value of the imaginary part of its eigenvalues yields the vortex strength.

At first glance, the vortex strength is only locally defined in the vicinity of a vortex core line. However, since the definition only depends on the velocity field and its gradient, it can be applied to every point of the computational domain, so the strength field is globally available for evaluation. The definition also makes sense outside the vortex regions, for if the eigenvalues of the projected velocity matrix are real numbers, the vortex strength is set to zero.

In this chapter, we gave a synopsis of basic flow visualisation terms, such as the most common grid types and data fields. Based on the definitions given here, we will in the next chapter describe a few basic computational techniques for flow visualisation, which were part of the algorithms developed for this thesis and presented in Chapter 5 to Chapter 7.

CHAPTER 3

FLOW VISUALISATION TECHNIQUES

In this chapter, we will give a short overview on several basic techniques for flow visualisation, most of which were used as ingredients for the algorithms and methods presented in the later chapters of this thesis. We will discuss here a few representatives of the flow visualisation categories mentioned in Section 2.1.3, namely of the direct, integration-based, region- and feature-based flow visualisation techniques.

3.1 DIRECT TECHNIQUES

As described in Section 2.1.3, direct visualisation techniques are computationally inexpensive, since they merely just evaluate the flow data and use simple glyphs for depicting the flow behaviour. Among the most common representatives of direct techniques are *arrow plots* (also called *hedgehog plots*) and *colour coding* of scalar data (like pressure, temperature or velocity magnitude of the flow). Both of them can also be combined and are mostly performed in two spatial dimensions.

3.1.1 Arrow and Hedgehog Plots

The probably most direct and intuitive visualisation method, at least for 2-dimensional flows, is to draw for every grid node an arrow indicating the velocity vector at this location [BCE92]. The arrow direction and the arrowhead indicate the direction of the flow velocity at the current node. The length of the arrow can be used for displaying the magnitude of the velocity vector, i.e. the speed of the flow.

However, this leads to very different arrow lengths among the grid nodes, which can disturb the visual perception: points of low velocity (e.g. near a *critical point* where the flow velocity is zero) might yield too short arrows which are hardly visible. On the other hand, points of high velocity might have too long arrows which cross each other, leading to a clutter of lines (see Figure 3.1, left).

To circumvent this problem, the arrow lengths can be kept constant by normalising the flow velocity vectors (Figure 3.1, right). Of course this leads to a loss of information, and at the latest when it comes to the visualisation of 3-dimensional flow, additional problems like occlusion arise. Furthermore, the projection from a 3D scene to a 2D image can also mislead the perception of the viewer. Therefore arrow plots are hardly used for 3D flow visualisation.

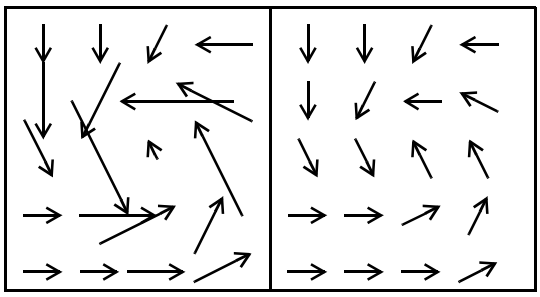


FIGURE 3.1 Arrow plots. Left: length proportional to velocity, right: constant length.

3.2 INTEGRATION-BASED TECHNIQUES

Also directly working on the physical flow data like velocity, these techniques differ from direct techniques in that they interpret the flow as a movement of particles rather than just evaluating the flow fields at fixed grid points. This requires the use of calculus methods, such as integrals for solving differential equations, and also interpolation methods for field evaluation, since the particles most of the time are located in the interior of grid cells.

3.2.1 Streamlines and Particle Tracing

To visualise the continuous nature of a (2- or 3-dimensional) flow field, *streamlines* can be computed based on the physical information contained in the flow [BCE92]. Let P be an arbitrary particle in a *steady* flow, which starts at time t_0 at the seed point $\boldsymbol{x_0}$. The initial condition of the particle movement is therefore

$$\boldsymbol{x}(t_0) = \boldsymbol{x_0} \qquad (t_0 \in \mathbb{R}, \boldsymbol{x}, \boldsymbol{x_0} \in \mathbb{R}^3) \quad , \tag{3.1}$$

and the movement of the particle fulfils the ordinary differential equation

$$\dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{v}(\boldsymbol{x}(t)) \qquad (t \in \mathbb{R}, \boldsymbol{x}, \boldsymbol{v} \in \mathbb{R}^3) \tag{3.2}$$

where the velocity of the particle depends on its current location in the constant flow field. The trajectory of the particle can therefore be integrated forward. This is mostly done by discretising the differential equation and applying explicit or implicit solution methods. A typical example is the Euler method, which is a first-order approximation since it is based upon a first-order Taylor series to discretise the differential equation.

A forward integration step is made by evaluating the velocity at the current particle location, computing the displacement of the particle during one time step, and updating

the location of the particle. By inverting the sign of the time step, is also possible to integrate backward the motion of the particle.

More accurate results can be obtained by decreasing the time step (which leads to rapidly increasing computation times) or by the use of higher-order methods such as the 2nd order Heun method or the classic 4th order Runge-Kutta method [Sch97]. Alternatively, the differential equation can be solved by implicit rather than explicit methods. This requires in every step the solution of a linear equation system but allows greater time steps, which also leads to more numerical stability.

While integration in *physical space* is straightforward, evaluating the velocity field is nontrivial for non-uniform or even unstructured grids, since it requires the identification of the cell the particle is currently located in, and of its local coordinates within that cell. The stencil walk algorithm by Buning [Bun89], for example, starts at the midpoint of a cell for iteratively searching the point position in *computational space*, using Newton's method (see also the book chapter on particle tracing of Sadarjoen et al. [SvWHP97]).

Streamlines are a very natural and obvious method to depict the continuous nature of a flow (see Figure 3.2). However, they mostly tend to get unevenly distributed even when being started at equidistant seed points. To cope with this problem, methods were developed for optimised automatic streamline placement, such as the methods by Turk/Banks [TB96], Jobard/Lefer [JL97], and Telea/vanWijk [TvW99].

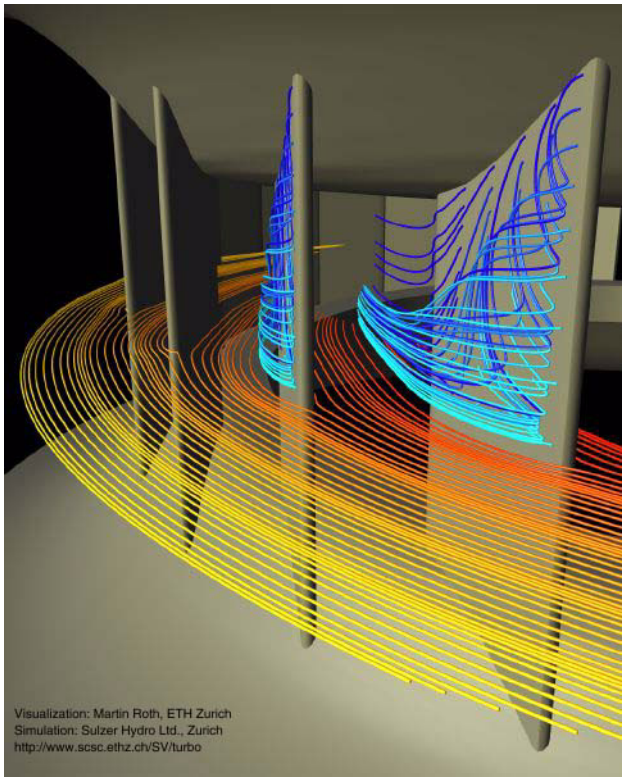


FIGURE 3.2 Streamlines indicating a vortex at the stay vanes of a water turbine (image courtesy of VA Tech Hydro Zurich and of ETH Zurich).

3.2.2 Path Lines and Streaklines

For *time-dependent* flow fields, it is also possible to compute *path lines* and *streaklines*, in a similar manner to streamlines. The difference for path lines is that for unsteady flow, the velocity field is updated at every time step, so the velocity evaluation is not only depending on the current particle location $\boldsymbol{x}(t)$ but also on the current integration time t:

$$\dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{v}(\boldsymbol{x}(t), t) \qquad (t \in \mathbb{R}, \boldsymbol{x}, \boldsymbol{v} \in \mathbb{R}^3) \tag{3.3}$$

The difference for streaklines is that for unsteady flow, several different particles are started at the same seed point but consecutively at different points in time.

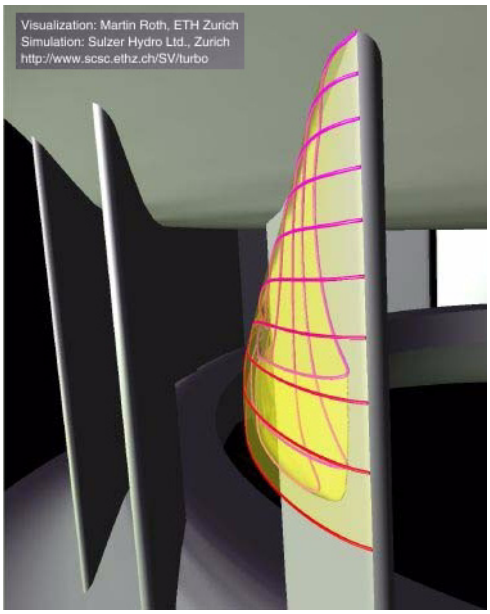


FIGURE 3.3 Stream surfaces indicating a vortex at the stay vanes of a water turbine (image courtesy of VA Tech Hydro Zurich and of ETH Zurich).

Streamlines from different seed points can be combined to *stream surfaces* by connecting sets of particle locations of the same time to polylines (see Figure 3.3). They are described in more detail in the work of Hultquist [Hul90, Hul92], in the article by Post and van Wijk [PvW94] and in the paper of Garth et al. [GTS+04].

3.2.3 Line Integral Convolution

A further alternative is the use of *Line Integral Convolution (LIC)*, which was presented by Cabral and Leedom [CL93] and improved by Stalling and Hege [SH95a, BSH97]. It combines integration-based with texture-based methods by smoothing a noise image along streamlines, resulting in a picture where the streamlines appear as smears (see Figure 3.4).

The original LIC algorithm works as follows: a streamline can be uniformly reparametrised using the arc length s of its curve rather than the integration time t. As a consequence of this, the ordinary differential equation (3.2) transforms to

$$\boldsymbol{x}'(s) = \frac{d\boldsymbol{x}(s)}{ds} = \frac{\boldsymbol{v}(\boldsymbol{x}(s))}{|\boldsymbol{v}(\boldsymbol{x}(s))|} \qquad (s \in \mathbb{R}, \boldsymbol{x}, \boldsymbol{v} \in \mathbb{R}^2) \tag{3.4}$$

where the velocity is normalised, leading to equidistant sample points on the computed streamline. For a given streamline $\boldsymbol{x}(s)$ and position $\boldsymbol{x_0} = \boldsymbol{x}(s_0)$, the intensity value I of a pixel in the result image is then calculated by the convolution integral

$$I(\boldsymbol{x_0}) = \int_{s_0-L}^{s_0+L} k(s-s_0)T(\boldsymbol{x}(s))ds \qquad (s, s_0 \in \mathbb{R}, \boldsymbol{x}, \boldsymbol{x_0} \in \mathbb{R}^2) \tag{3.5}$$

where k is a one-dimensional scalar weighting function (a filter of length $2L$ applied along the streamline, starting from the pixel position) and T is a 2D texture image (usually white noise) of the same resolution as the result image. Since the result image is a grey-level one, colours can be used to display additional information like the velocity magnitude.

Line Integral Convolution is mostly applied for 2D images, for the same reasons as arrow plots. In the example image, a 3D dataset was underlying and the 3D perception problem was solved by taking a cross-section of the grid and mapping the third velocity component onto that plane using the colour of the streamlines. However, there are also 3D LICs possible using volume rendering techniques, as were suggested by Interrante and Grosch [IG97, IG98].

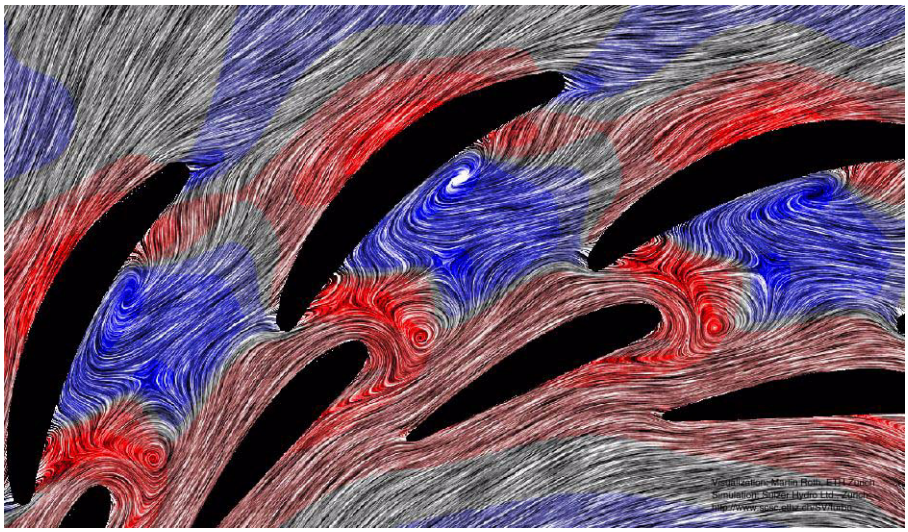


FIGURE 3.4 Line Integral Convolution image for guide and stay vanes of a water turbine (image courtesy of VA Tech Hydro Zurich and of ETH Zurich).

3.3 REGION-BASED TECHNIQUES

A completely different approach to visualise a flow behaviour is to restrict the interest to certain regions of the flow. Flow-relevant features like vortices can then be defined as connected sets of 3D points in the computational domain. This set of points often represents a closed surface bounding a volume surrounding the feature, e.g. a hull surrounding a vortex core line (see Chapter 6).

3.3.1 Isosurfaces and Vortex Hulls

Such a *region-of-interest (ROI)* of the flow can, for instance, be limited by an *isosurface* of one of the scalar fields mentioned in Section 2.3.2. A common choice for the scalar field is the absolute value of the flow vorticity. One can assume that in the interior of a vortex, the vorticity magnitude is high and must lie above a predefined scalar threshold (see the article of Zabusky et al. [ZBP+91]):

$$|\nabla \times \boldsymbol{v}| \geq \omega_{thresh} \tag{3.6}$$

Rather than taking the vorticity magnitude to define the boundary isosurface of the vortex, one can also use helicity (which is derived from vorticity and velocity, see Section 2.3.2):

$$(\nabla \times \boldsymbol{v}) \bullet \boldsymbol{v} \geq h_{thresh} \tag{3.7}$$

or alternatively the normalised helicity:

$$\frac{(\nabla \times \boldsymbol{v}) \bullet \boldsymbol{v}}{|\nabla \times \boldsymbol{v}| \cdot |\boldsymbol{v}|} \geq h_{thresh} \tag{3.8}$$

Another variant is to assume that there is a pressure minimum in the interior of a vortex:

$$p \leq p_{thresh} \tag{3.9}$$

Computationally more expensive is the so-called *lambda*$_2$ method [JH95], which for a certain location decomposes the Jacobian matrix J into its symmetric part S and antisymmetric part Ω and computes the three real eigenvalues $\lambda_1, \lambda_2, \lambda_3$ of the symmetric matrix $S^2 + \Omega^2$. A vortex is regarded as a region where at least two eigenvalues are negative, so if $\lambda_1 \leq \lambda_2 \leq \lambda_3$, the boundary isosurface should enclose all points with

$$\lambda_2 \leq 0 \tag{3.10}$$

However, all of these purely isosurface-based methods have in common that they are not suitable for reliably detecting the vortices in a typical turbomachinery flow, since the vortices of these are in general weak structures and not isolated from each other very well, as was shown by Roth and Peikert [RP96, Rot00] in the context of curved vortices in a bent flow (see Figure 3.5). Furthermore, the scalar threshold criterion is often met at the boundaries of the grid, so the isosurface method tends to produce *false positives* (e.g. vorticity is often maximal at the grid boundary due to strong shear, see Figure 3.6).

Therefore a combination of scalar thresholding with *feature*-based techniques is more promising. One possibility is to firstly extract line-type features, such as *vortex core lines* (see Section 3.4.2 and Chapter 5), and then build growing surfaces surrounding the features [BP02], such as *vortex hulls*. We developed a suitable implementation of such vortex hulls for unstructured grids, which will be described in detail in Chapter 6 of this thesis.

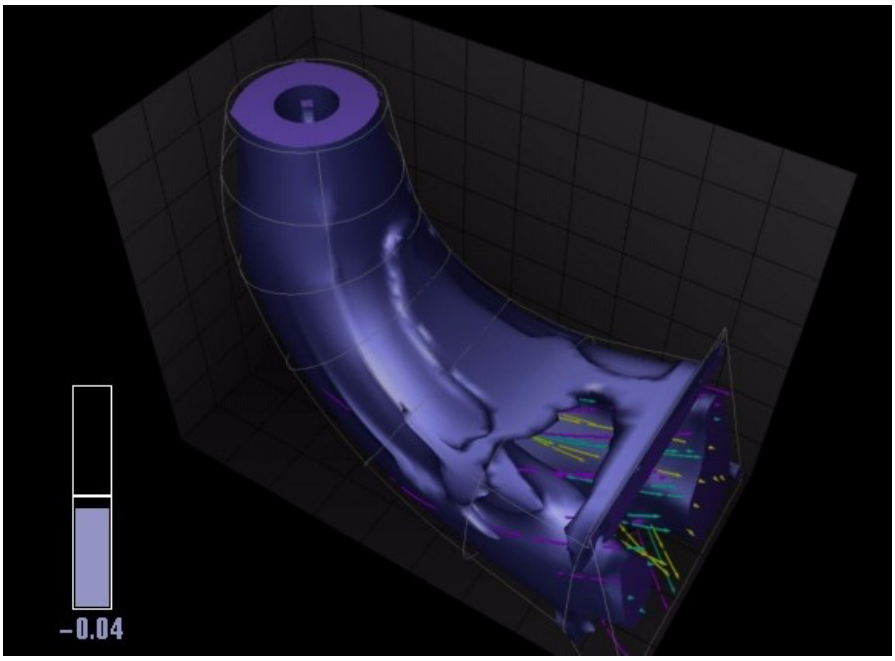


FIGURE 3.5 *Lambda*$_2$ isosurfaces in a Francis turbine showing connected components (image courtesy of VA Tech Hydro Zurich and of ETH Zurich).

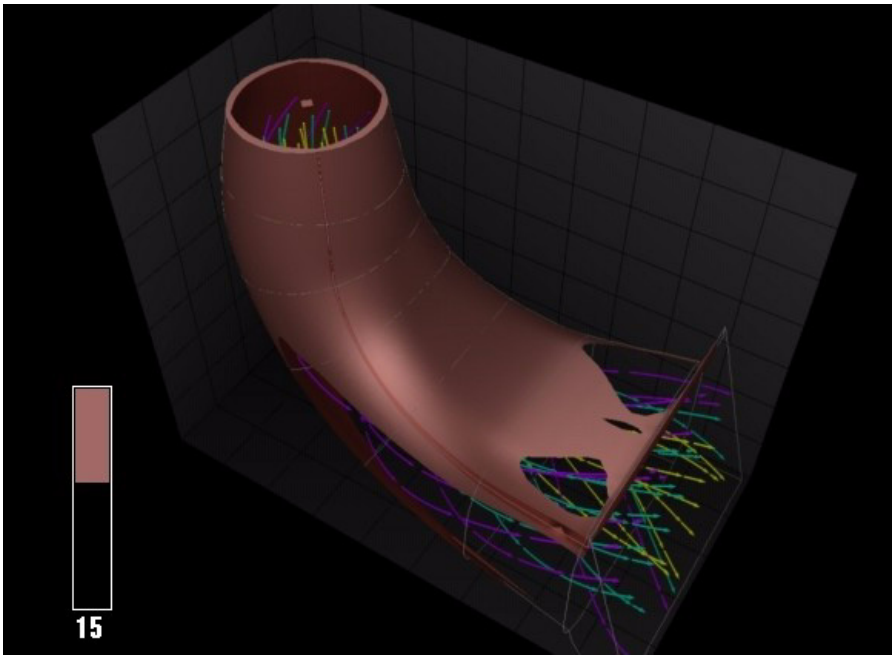


FIGURE 3.6 Vorticity isosurface in a Francis turbine showing false positive at grid boundary (image courtesy of VA Tech Hydro Zurich and of ETH Zurich).

3.4 FEATURE-BASED TECHNIQUES

As previously explained in Section 2.1.3, there are several good reasons for visualising flow fields on a more abstract level, namely by extracting *features* of the flow. Such features can be described in a topological way, for instance using 0-dimensional *point-type* features or 1-dimensional *line-type* features. These have been defined in literature using various criteria, mostly based upon the vector and scalar fields contained in the flow data. We will in the following give some explanations to both techniques and later in Chapter 5 concentrate on line-type features.

3.4.1 Vector Field Topology and Critical Points

If the underlying flow field is a 2-dimensional vector field, e.g. containing the velocities at the grid points, the vector field can be linearised or regarded as linear in each cell (e.g. by subdivision of quadrangular cells into triangular ones and subsequent linear interpolation of the vectors within the triangular cells, see also Section 5.2.4).

Helman and Hesselink [HH89, HH91] introduced the concept of *topological analysis* of 2D linear vector fields. They detected and classified the *critical points* of a flow, which are the isolated points where the velocity field is zero (provided that $\det(A) \neq 0$, see below). The key is to compute the *eigenvalues* and *eigenvectors* of the velocity gradient (see Section 2.3.1) at these points. We assume a linear vector field

$$\boldsymbol{v}(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b} \qquad (\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^2, A \in \mathbb{R}^{2\times 2}) \qquad \text{, where} \tag{3.11}$$

$$A = \begin{bmatrix} v_{x,x} & v_{x,y} \\ v_{y,x} & v_{y,y} \end{bmatrix} = \operatorname{grad} \boldsymbol{v} = \nabla \boldsymbol{v} \tag{3.12}$$

is the Jacobian of the velocity. Solving its *characteristic equation*

$$\det(A - \lambda I) = \begin{vmatrix} v_{x,x} - \lambda & v_{x,y} \\ v_{y,x} & v_{y,y} - \lambda \end{vmatrix} = 0, \tag{3.13}$$

$$\lambda^2 - (v_{x,x} + v_{y,y}) \cdot \lambda + (v_{x,x} \cdot v_{y,y} - v_{x,y} \cdot v_{y,x}) = 0 \text{, or simply} \tag{3.14}$$

$$\lambda^2 - \operatorname{trace}(A) \cdot \lambda + \det(A) = 0 \tag{3.15}$$

yields the eigenvalues

$$\lambda_{1,2} = \frac{\operatorname{trace}(A) \pm \sqrt{\operatorname{discr}(A)}}{2} \tag{3.16}$$

where the *discriminant* of the characteristic equation is

$$\operatorname{discr}(A) = \operatorname{trace}^2(A) - 4 \cdot \det(A). \tag{3.17}$$

We can now distinguish between the cases shown in Table 3.1, depending on the trace and determinant of the velocity Jacobian, and on the discriminant of its characteristic equation. The trace of the velocity Jacobian is equal to the flow *divergence*.

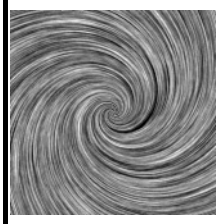
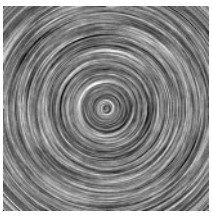
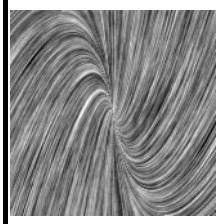
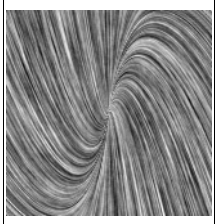
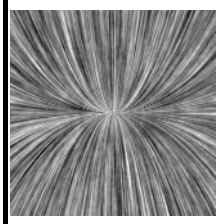
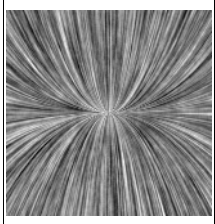
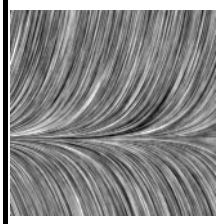
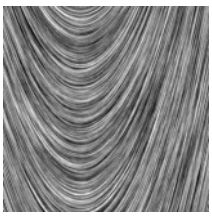
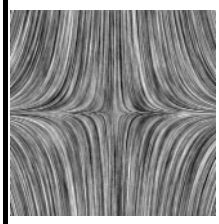
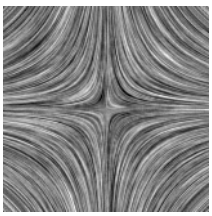
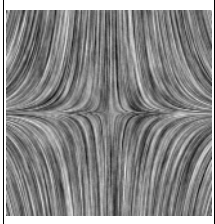
		$\text{trace}(A) > 0$	$\text{trace}(A) = 0$	$\text{trace}(A) < 0$
$\text{discr}(A) < 0$	$\det(A) > 0$	focus source 	center 	focus sink 
$\text{discr}(A) = 0$	$\det(A) > 0$	node focus source 		node focus sink 
$\text{discr}(A) > 0$	$\det(A) > 0$	node source 		node sink 
$\text{discr}(A) > 0$	$\det(A) = 0$ (not a critical point)	line source 	shear 	line sink 
$\text{discr}(A) > 0$	$\det(A) < 0$	saddle source 	saddle div.-free 	saddle sink 

TABLE 3.1 Different cases of velocity gradient eigenanalysis and vector field topology (table according to Peikert [Pei03], images courtesy of ETH Zurich).

The topology near a critical point can be of various shapes. For complex eigenvalues, a *focus* indicates a vortex spiralling around a rotational center (see also the definition of vortex strength in Section 2.3.2). For real eigenvalues, we obtain a *node focus*, *node*, *line* or *saddle*, the real eigenvectors then being asymptotes of the flow near the critical points. Based upon the classification of the critical points, it is possible to construct a topological *skeleton* of separatrices connecting them, and thus to subdivide the domain into topologically homogeneous regions.

Vector field topology has widely been used for 2-dimensional flows (e.g. by Peikert [Pei03]) and also been extended to 3-dimensional flows (e.g. by Chong, Perry and Cantwell [CPC90, PC87, PC92], by Tobak and Peake [TP82] and by Soria and Cantwell [SC92]). However, high-resolution datasets of turbulent 3D flows can contain a large number of critical points, cluttering the flow topology image. To cope with this, de Leeuw and van Liere [dLvL99] proposed a *multilevel* topology analysis, removing structures of varying scales from the flow topology. A similar effect could be achieved by using a scale-space representation of the flow data. Furthermore, line-type features are less prone to cluttering than isolated points. We will pursue both of these approaches in Chapter 4 and Chapter 5.

3.4.2 Vortex Core Lines and Parallel Vectors Operator

A line-type vortex definition assumes that the flow spirals around a (straight or bent) central axis, which is called *vortex core line*. The main advantage of vortex core lines is that due to their thin shape, different vortices can clearly be separated from each other, which also makes *feature tracking* an easier task. Furthermore, a line-type feature has a clearly defined extent (since its thickness is mathematically zero, no termination criterion for the object boundary is necessary, as is for region-type methods like isosurfaces). And finally, core lines are more stable for they lie in the centre of the vortex, where the vortex is strongest.

A variety of vortex core line definitions can be found in literature, mostly based on the velocity or pressure field of the flow and their derivatives. Since some of the criteria were expressed by complex algorithms, the original versions are difficult to compare to each other. Remarkably, it is yet possible to reduce them to a common framework, which is a relatively simple computational scheme.

This scheme was discovered by Roth and Peikert, who published it in 1999 as the *parallel vectors operator* [RP99]. In general, the concept means that any vortex core line defined by one of the mentioned vortex criteria can be computed as a set of points where two given vector fields $\boldsymbol{u}$ and $\boldsymbol{w}$ are parallel, or as a subset thereof:

$$\boldsymbol{u} \parallel \boldsymbol{w} \qquad (\boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^3) \tag{3.18}$$

This condition can also be written as

$$\boldsymbol{u} = \lambda \boldsymbol{w} \qquad (\lambda \in \mathbb{R} \cup \{-\infty, \infty\}) \tag{3.19}$$

where the magnitude of the parameter λ denotes the aspect ratio of the two vectors and its sign indicates whether they are parallel or antiparallel.

We will in the next sections describe five existing vortex core line definitions and show how they can be mathematically expressed by parallel vectors, as suggested by Roth [Rot00]. At the end of this chapter, we will give a systematic classification of the methods, their underlying vector fields and their embedding into the parallel vectors scheme.

3.4.3 Vortex detection method by Levy, Degani and Seginer

Levy, Degani and Seginer define a vortex by means of the normalised helicity (see Section 2.3.2). Since this scalar is a dot product of two normalised vectors, namely the normalised velocity $\hat{\boldsymbol{v}}$ and normalised vorticity $\hat{\omega}$ (see Figure 3.7), it is equal to the cosine of the angle enclosed by these two vectors, and thus it lies in the range of -1 to +1:

$$-1 \leq \frac{\omega \bullet \boldsymbol{v}}{|\omega| \cdot |\boldsymbol{v}|} \leq 1 \tag{3.20}$$

In their pioneering paper [LDS90], Levy et al. expect the normalised helicity to tend to one of these extremal values when approaching a vortex. The border case is equal to the assumption that the velocity and vorticity vector are parallel or antiparallel (i.e. they build an angle of 0 degrees or 180 degrees, respectively):

$$\boldsymbol{v} \parallel \omega \tag{3.21}$$

In spite of the fact that this criterion is sufficient for a line-type feature definition, Levy et al. use an algorithmic approach which computes slices with contour plots of normalised helicity. After manually locating sectional extrema of normalised helicity on these slices, they integrate streamlines starting at the extremum points to detect the vortices, assuming that the vortex core is a streamline itself.

This method, however, mixes the *local* definition of a normalised helicity extremum with the *global* definition of a streamline, which is not proper for general vector fields. A further drawback is the fact that the extraction of vortex core lines is not fully automated in their algorithm.

In contrast to the original paper, we will restrict ourselves to the parallel vectors criterion of Equation 3.21 when referring to the Levy method. This approach not only is mathematically simpler but also has the advantage of including *critical points* into the vortex definition. Since velocity or vorticity can be zero at distinct locations even in a non-degenerate flow field, the pure normalised helicity definition of Equation 3.20 fails in these cases because of division by zero when normalising the two vectors. By expressing the vortex criterion in terms of *original* velocity and vorticity, the parallel vectors operator avoids that problem.

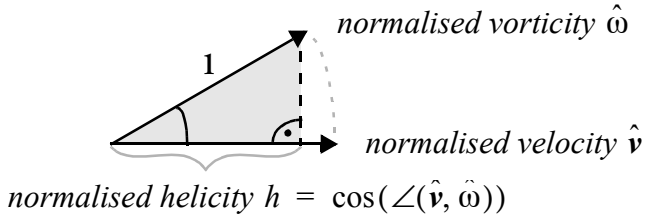


FIGURE 3.7 Principle of the Levy method (sketch according to [Rot00]).

3.4.4 Vortex detection method by Sujudi and Haimes

Sujudi and Haimes give an algorithmic definition of a vortex. In a technical report and a paper [SH95b, SH95c], they assume a tetrahedral grid for the computational domain of the flow. The velocity field can thus be linearly interpolated within every grid cell, and the velocity gradient is constant within every cell. The algorithm loops through all grid cells and computes this 3×3 Jacobian matrix and its eigenvalues for every cell. The case of three real eigenvalues is rejected since a vortex only appears where complex eigenvalues exist, analogously to the 2-dimensional case (see Peikert's summary [Pei03]). In the case of only one real eigenvalue, its eigenvector is computed and the velocities for all four nodes of the current tetrahedron are projected onto a plane perpendicular to this real eigenvector. The linear field of these projected velocities has a straight line of zeroes. Intersecting this line with the tetrahedral cell potentially yields two intersection points on the triangular faces, which limit a line segment indicating a piece of a vortex core line (see Figure 3.8).

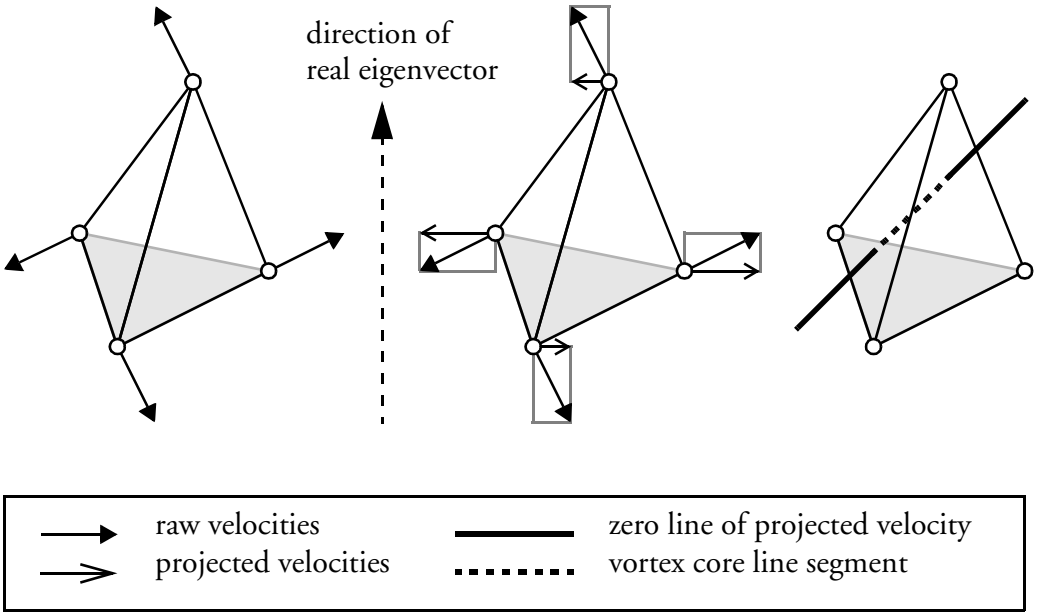


FIGURE 3.8 Principle of the Sujudi/Haimes method (sketch according to [SH95b] and [Rot00]).

This rather complex algorithm can be simplified as follows: As is evident from Figure 3.8, the condition that on the vortex core line the *projected* velocity is zero is equivalent to the condition that the *raw* velocity is parallel or antiparallel to the real eigenvector. This, however, means that the raw velocity $\boldsymbol{v}$ is *itself* a real eigenvector of the original Jacobian J:

$$J\boldsymbol{v} = \lambda\boldsymbol{v} \tag{3.22}$$

And this again means that the velocity vector is parallel to the product of Jacobian and velocity vector, which is the *acceleration* of a particle in the case of a steady flow field:

$$\boldsymbol{v} \parallel (\nabla\boldsymbol{v})\boldsymbol{v} \tag{3.23}$$

Similar to the Levy case, the Sujudi/Haimes vortex criterion can be formulated as a problem of finding parallels for two vector fields.

3.4.5 Vortex detection method by Banks and Singer

Banks and Singer were the first researchers to present a vortex method which explicitly constructs vortex cores as line-type features. In a paper [BS94] and in a technical report [SB94], they propose a *predictor-corrector* scheme for extending a vortex core line. Once a point has been found on the vortex core line, the algorithm performs a small integration step to predict the next point on the vortex core line (see Figure 3.9). This integration step reminds of the Euler method for streamline integration but the difference is that the Banks/Singer method uses the vorticity rather than the velocity field for the integration.

The vorticity is then evaluated at the predicted point, and a plane containing the predicted point and perpendicular to its vorticity vector is established. The assumption is now that the vortex core is a region of low pressure p, thus the pressure field projected to the plane has a local minimum, which marks the next point on the vortex core line. The local pressure minimum is thus computed and its location is used as the corrected point, provided that the corrected point is not too far away from the predicted one. Otherwise the algorithm stops and assumes that the end of the vortex core line has been reached.

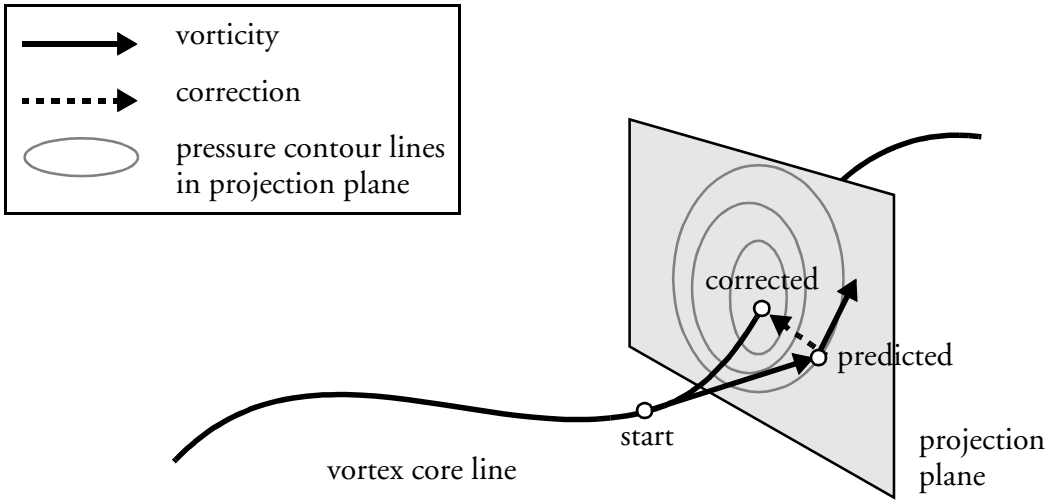


FIGURE 3.9 Principle of the Banks/Singer method (sketch according to [SB94]).

The Banks/Singer method is obviously based upon the vorticity field and the pressure field of the flow. To be more precise, it steps along the vorticity and searches for local minima of pressure, assuming that the vorticity variation is small compared to the pressure variation. For decreasing step sizes, the method thus computes points where the vorticity is parallel to the pressure gradient:

$$\nabla p \parallel \nabla \times \boldsymbol{v} \tag{3.24}$$

Since the search for local minima of the (projected) pressure field is based on finding the zeroes of the (projected) pressure gradient, it also yields local maxima and saddle points. To restrict the results to local minima, a second criterion must be applied in addition to the parallel vectors operator, which excludes false positives. This can be achieved by checking the Hessian (i.e. the second derivatives) of the pressure. Points are only minima when the two eigenvalues of the (symmetric) pressure Hessian are both positive.

3.4.6 Vortex detection method by Miura and Kida

Miura and Kida proposed another method to find vortex core lines based on *minima of pressure*. In a paper [MK96], a technical report [MK97a] and several articles [MK97b, MK98a, MK98b, MK98c], they define a modified pressure, which is under certain conditions equal to the pressure. Their algorithm computes for every grid node the pressure Hessian matrix, its eigenvalues $\lambda_0, \lambda_1, \lambda_2$ and eigenvectors $\boldsymbol{e_0}, \boldsymbol{e_1}, \boldsymbol{e_2}$ (since the Hessian is symmetric, all three eigenvalues are real). The coordinate system is then transformed to another one spanned by the three eigenvectors. Within the new system, the pressure can be quadratically approximated w.r.t. the local coordinates ξ_0, ξ_1, ξ_2 by the scalar function

$$p(\xi_0, \xi_1, \xi_2) = p(c_0, c_1, c_2) + \frac{1}{2} \sum_{i=0}^{2} \lambda_i (\xi_i - c_i)^2 \tag{3.25}$$

which has a local extremum at the centre point $C(c_0, c_1, c_2)$. For a pressure minimum, λ_1 and λ_2 are both positive, and the graph of the pressure function is an upward opening paraboloid. On any plane parallel to the plane spanned by the eigenvectors $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$, the graph of the projected pressure p is a parabola and minimal on a straight line through C and aligned with the $\boldsymbol{e_0}$ direction (see Figure 3.10). The point on this straight line closest to the grid node $\boldsymbol{Q}$ is chosen for the next point $\boldsymbol{R}$ on the vortex core line, provided that the distance does not exceed a certain threshold depending on the cell size. At the end, the isolated core line points are connected to polylines using a nearest neighbour heuristic.

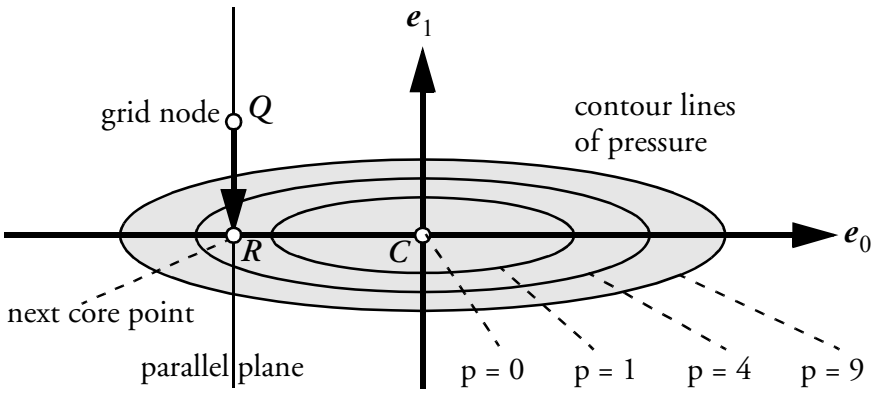


FIGURE 3.10 Principle of the Miura/Kida method (sketch according to [MK96] and [Rot00]).

This rather complex algorithm can again be expressed by a simpler formulation. Roth [Rot00] shows that by picking points on the predicted line of sectional pressure minima, the Miura/Kida method tracks exactly the *valley lines of pressure*. These are the places where the pressure gradient is an eigenvector of the pressure Hessian. At the vortex core line, the product of the pressure Hessian and the pressure gradient must therefore be parallel to the pressure gradient:

$$\nabla p \parallel (\nabla(\nabla p))(\nabla p) \tag{3.26}$$

Again, the vortex criterion could be expressed by the local parallelism of two vector fields.

3.4.7 Vortex detection method by Strawn, Kenwright and Ahmad

Similar to the Miura/Kida method which seeks local pressure minima, it is also possible to search for local extrema of other data fields of the flow. In the context of aeronautics, Strawn, Kenwright and Ahmad [SKA98, SKA99] suggested the computation of local *maxima of vorticity magnitude* to find the centre lines of vortices. Their implementation assumes a structured hexahedral grid where the vorticity, vorticity magnitude and its gradient have been precomputed for all nodes using central differences.

The algorithm loops through all interior grid cells and treats all six faces for every cell. For a certain face, a 4×4 nodal patch is defined containing the four corners of the face and 12 neighbouring nodes (see Figure 3.11). The vorticity magnitude of these 16 nodes is compared and the central face only considered a candidate if the node with greatest vorticity magnitude is a face corner. In this case, the exact location of the maximum on the face is determined by bilinear interpolation of the vorticity magnitude gradient on the face and computing the zeroes of the 2D gradient restricted to the face plane.

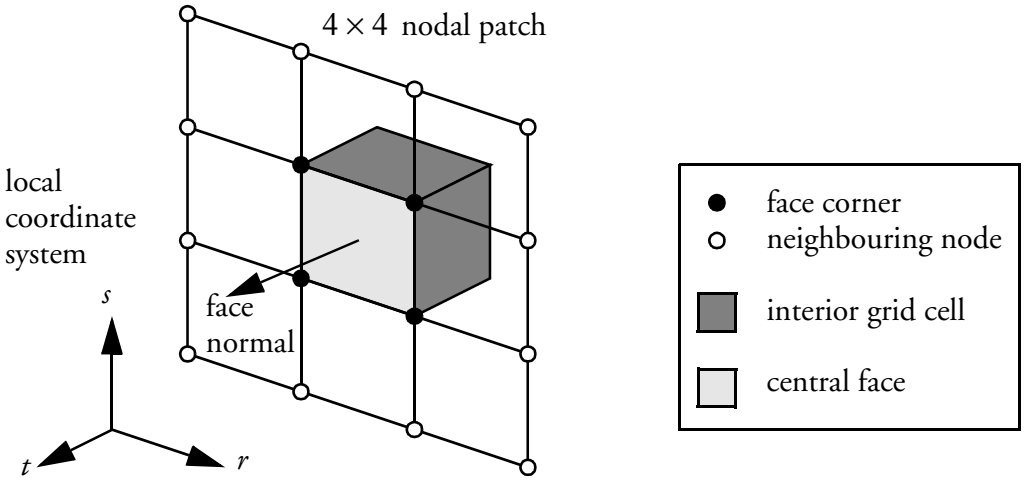


FIGURE 3.11 Principle of the Strawn/Kenwright/Ahmad method (sketch according to [SKA98]).

The use of cell faces for calculating the local maxima is an approximation. Actually, the goal is to find the *maximum lines* of vorticity in the flow, i.e. the set of points where the vorticity magnitude $|\omega|$ is maximal in a plane containing the point and perpendicular to the vorticity ω. Since for a maximum point, the 2D gradient of vorticity magnitude in this plane is zero, the 3D gradient is normal to the plane and thus parallel to the vorticity:

$$\omega \parallel \nabla|\omega| \tag{3.27}$$

Of course we can also maximise the square of vorticity rather than its magnitude to find the maximum points. And since $\nabla(\omega^2) = 2(\nabla\omega)^T\omega$, we get the vortex criterion

$$\omega \parallel (\nabla\omega)^T\omega \tag{3.28}$$

The vorticity must be parallel to the product of transposed vorticity gradient and vorticity.

3.4.8 Mapping of vortex criteria to Parallel Vectors Operator

As has been shown in the previous sections, the various vortex definitions seem at first glance to be completely different from each other, but can all be expressed using the parallel vectors operator scheme [RP99]. Table 3.2 summarises the different vortex criteria, the vector fields they are based on, and their assignment to the input fields of the parallel vectors operator.

Authors	1st vector field	symbol	2nd vector field	symbol
Sujudi/Haimes	velocity	$\boldsymbol{v}$	acceleration of fluid particle	$(\nabla \boldsymbol{v})\boldsymbol{v}$
Levy/Degani/Seginer	velocity	$\boldsymbol{v}$	vorticity	ω
Strawn/Kenwright/ Ahmad	transposed vorticity gradient times vorticity	$(\nabla\omega)^T\omega$	vorticity	ω
Banks/Singer	pressure gradient	∇p	vorticity	ω
Miura/Kida	pressure gradient	∇p	pressure Hessian times pressure gradient	$(\nabla(\nabla p))(\nabla p)$

TABLE 3.2 Different vortex criteria and their parallel vectors representation (table entries rearranged for better overview of common vector fields).

In Chapter 5, we will discuss algorithms for extracting and tracking vortex core lines, which were implemented using the parallel vectors operator. Due to the generality of the approach, there are additional vortex criteria in literature suitable for this formulation. It is thus relatively easy to implement additional vortex criteria by just setting up the required vector fields and using the existing parallel vectors framework.

CHAPTER 4

SCALE-SPACE TECHNIQUES

When doing physical measurements or observations of real-world objects, it is impossible to get a perfect representation of the sampled data because the "aperture" of the observation instrument (e.g. a human's eye or a photo camera) cannot be varied arbitrarily, thus the image resolution is limited to a certain range of scales. The necessity of regarding several levels-of-detail when investigating datasets gained from physical measurements led to the development of so-called *multi-scale* and *multi-resolution* representations. These data representations have widely been used for image processing since the beginning of the 1970s, but it is no trivial task to relate image structures across different levels of scale, nor to distinguish significant image features from noise.

A major breakthrough was the introduction of the *scale-space* theory by Witkin and Tenenbaum [Wit83, WT83] and by Koenderink [Koe84] in 1983/84. They define the *inner scale* as the smallest level-of-detail we can resolve and represent in our input data (e.g. one cone or rod of our retina, or one image pixel), the *outer scale* as the largest possible structure (e.g. our field of view, or the whole image).

In this chapter, we give an overview of some important data representations dealing with different levels of resolution or scale. After sketching the multi-resolution and multi-scale techniques, we present the scale-space representation and its special properties, along with some application examples from image processing and computer vision. The use of scale-space methods for flow visualisation is also motivated.

At the end of this chapter, several methods for computing the scale-space representation on different grid types are discussed and compared. One challenge in the field of flow visualisation is the use of unstructured grids originating from CFD simulations. As a first major contribution of this Ph.D. thesis, we will therefore present a method for computing the (linear isotropic) scale-space representation for unstructured grids.

4.1 MULTI-RESOLUTION AND MULTI-SCALE REPRESENTATIONS

There are several reasons for constructing multi-resolution and multi-scale data representations in the sciences. Unfortunately, both terms have often been used interchangeably in literature. To avoid any confusion, we will therefore strictly use the term multi-*resolution* if the representation is based on data with different spatial resolutions for different scale levels, and multi-*scale* if the data has got the same spatial resolution at every scale level.

What is common to both types of data representations is the need for several levels of detail or scale, respectively. For example, regard the fundamental problem of edge detection in image processing. Figure 4.1 shows a cross-section through an object boundary in a 2D image. The black dots represent the grey-level values sampled along the cross-section direction. In order to extract an edge, gradients have to be computed by discrete approximation of the first derivative of the grey-level function along the cross-section. The different slopes of the straight lines indicate that the result of the first derivative estimation strongly depends on the size of the support of the difference operator and thus on the scale of observation.

The dotted line, which marks the slope computed from two directly neighboured grey-level values, seems to be only a product of noise and thus to be useless. However, even the dashed line does not certainly mark the slope of the "edge", because it could also be noise superimposed to a bigger structure on a coarser scale. In this case, the continuous line might yield a better approximation for the slope of the edge. Since we have in general no a-priori knowledge about the size of the features of a given data signal, it is necessary to regard *all* levels of scale, or at least a wide range of different scales.

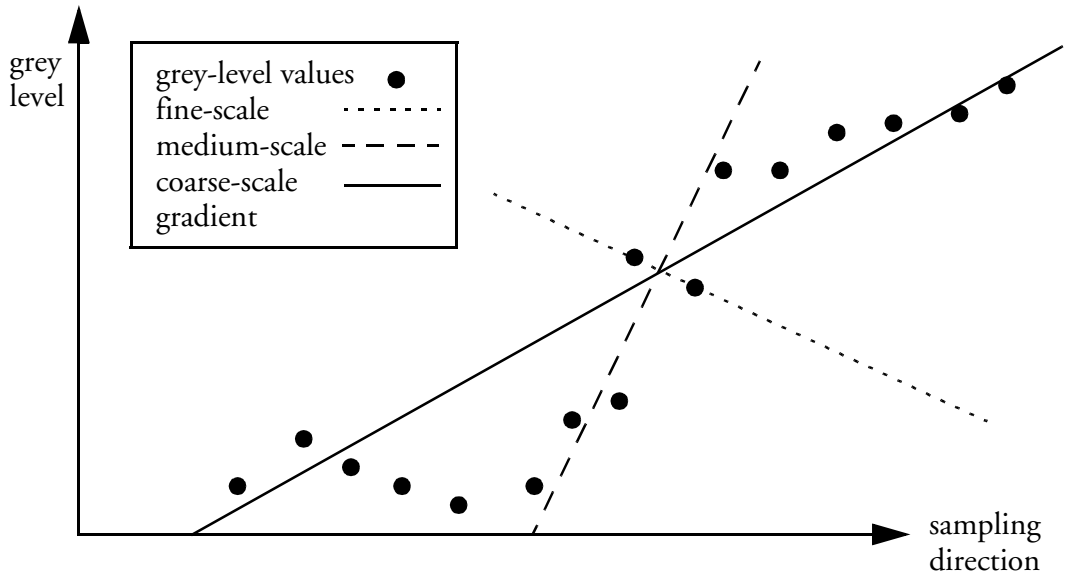


FIGURE 4.1 The basic scale problem, as it occurs in the task of edge detection (sketch according to Lindeberg [Lin94]).

4.1.1 Multi-resolution data representations

Multi-resolution techniques, such as *quad-trees* [Kli71], pyramids, *multi-grid* methods [Hac85] and wavelets, are especially useful for the task of data compression, which reduces storage requirements and transfer times on networks. A typical example is the *pyramid* of a 2D image, a concept which arose during the 1970s. Beginning with the original image, a set of images with exponentially decreasing size is constructed by successively smoothing and subsampling (see Figure 4.2). In the simplest case, the original image is a squared one with $2^n \times 2^n$ pixels. The size of every subsequent image decreases by a factor of 2 in every dimension, thus by 4 in the 2D image case. Every pixel of the smaller image is computed from a certain neighbourhood in the previous image, e.g. by averaging 4 pixels of the corresponding position in the larger image. In the one-dimensional case, this *reduce* operation can recursively be written as

$$f_{k+1}(x) = \sum_{n=-\infty}^{\infty} c(n) \cdot f_k(2x - n) \qquad (k = 0, 1, 2, \ldots) \tag{4.1}$$

where $f_0(x)$ represents the original image and $c(n)$ stands for a weighting function which is the filter mask of the discrete convolution. Rather than just averaging (which leads to severe aliasing problems), more sophisticated filter masks can be applied, like a *binomial filter* with weights (1/4, 1/2, 1/4) for every dimension. The binomial filter is a discrete approximation of a Gaussian filter, since its weights are all positive, symmetric to the centre of the filter and sum up to 1. Repeated application of such a discrete smoothing filter successively suppresses high image frequencies and therefore leads to a *low-pass pyramid*, which was proposed by Burt [Bur81] and Crowley [Cro81] at the beginning of the 1980s. By regarding the difference between two adjacent levels in a low-pass pyramid, one obtains a *bandpass pyramid*, which is also called *Laplacian pyramid* or *difference of low-pass pyramid* (DOLP). Such bandpass pyramids have widely been used for feature detection and data compression.

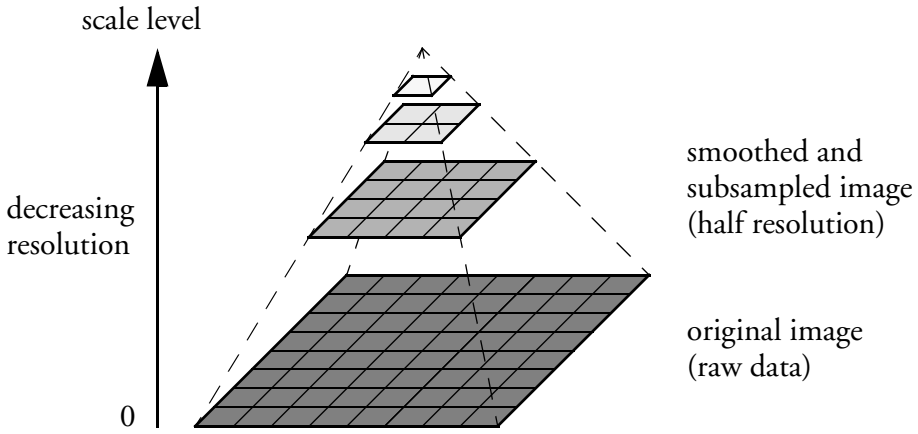


FIGURE 4.2 Multi-resolution (pyramid) representation of a two-dimensional image (sketch according to Lindeberg [Lin94]).

An advantage of bandpass pyramids is that they do not lose detail information at each subsampling step, but store the details in addition to the smoothed data. The original data can then be reconstructed from a coarser resolution level by an *expand* operation, which is the inverse of the reduce operation. A similar mechanism is part of the *wavelet* approach, which was introduced by Daubechies [Dau88] and for instance used in conjunction with volume rendering by Lippert [Lip98] and surface/volume compression by Staadt [Sta01]. In contrast to Laplacian pyramids, (orthogonal) wavelets provide a *critical sampling* of the data, i.e. there is no redundancy between data at different resolution levels.

The main advantage of pyramid representations is that the image size rapidly decreases, reducing both the memory requirements and the computational effort of the pyramid construction and subsequent algorithms. This allows for *progressive transmission*, which is especially important for transferring multimedia structures like images and movies over networks. Pyramids are constructed from an algorithmic process, which makes theoretical analysis more difficult in comparison to a representation based on purely analytic formulas. For our purposes, multi-resolution techniques would not be an optimal choice because pyramids correspond to a quite coarse quantisation along the scale dimension, which might not be sufficient for relating and tracking features across scales. Also, the loss of detail due to reduced spatial resolution is not always tolerable, as we will see in Section 4.5.

4.1.2 Multi-scale data representations

Multi-scale techniques define a one-parameter family of signals which also are derived from the original signal by a smoothing process, which successively suppresses fine-scale information (see Figure 4.3). In contrast to the multi-resolution approach, the resolution here is kept equal for all levels of scale, leading to a stack of images rather than a pyramid. Furthermore, the smoothing operation must not necessarily be formulated algorithmically but can also be expressed using mathematical formulas dealing with *continuous* functions, as we will see later. This makes it possible to overcome the discrete scale levels of the multi-resolution scheme by admitting real numbers for the scale parameter.

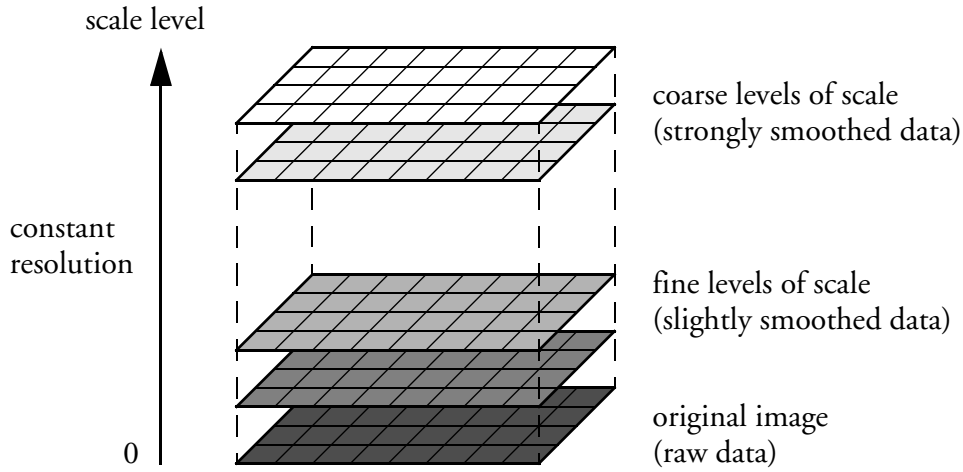


FIGURE 4.3 Multi-scale representation of a two-dimensional image (sketch according to Lindeberg [Lin94]).

The multi-scale approach not only provides a framework for accessing arbitrary levels of scale but also simplifies the implementation of many numerical algorithms for feature extraction, since these often require the numerical computation of derivatives. To cope with the roughening effect of these derivatives, some amount of smoothing is necessary. Since for the multi-scale representation the underlying data have already been smoothed, no smoothing functionality needs to be integrated into the feature extraction algorithms, easing their implementation and enhancement. In the next section, we will propose the scale-space concept as a multi-scale representation with unique and canonic properties, which are advantageous for many application fields.

4.2 SCALE-SPACE CHOICE AND PROPERTIES

In the previous section, we explained the basic structure of a multi-scale representation of 2D image data. This concept can also be extended to higher dimensions and other data structures. For reasons of simplicity, we will keep the term "image" until the introduction of specific flow visualisation needs in Section 4.5.

The question arises what type of filter is most appropriate for the smoothing operation to build a coarse-level image from its predecessor in the multi-scale image stack. A natural choice for the smoothing kernel would be a Gaussian filter. The crucial point was shown by Florack et al. [FtHRKV92]: The convolution of an n-dimensional data signal $f(\boldsymbol{x})$ with the n-dimensional Gaussian kernel

$$G(\boldsymbol{x}, \sigma) = \frac{1}{\sqrt{(2\pi\sigma^2)^n}} \cdot e^{-\frac{\boldsymbol{x}^2}{2\sigma^2}} \tag{4.2}$$

where

$$s = \frac{\sigma^2}{2} \tag{4.3}$$

holds for the scale level s in relation to the standard deviation σ, is the unique family of "aperture" functions to meet the following requirements:

- *Linearity:*
  Applying the smoothing function can be interchanged with adding two datasets or multiplying a dataset with a scalar. (This means that, for instance, if an original image has been smoothed and its grey level values shall later be stretched by a linear transformation, the computationally expensive convolution operation needs not to be done again - stretching the grey level values of the smoothed image results in the same image as smooting the image after stretching its grey level values.)
- *Shift invariance:* There is no preferred location of an image structure or feature.
- *Scale invariance:* There is no preferred size of an image structure or feature.
- *Isotropy:* There is no preferred direction for an image processing operator.
- *Semigroup (cascade smoothing) property:*
  The set of smoothing functions forms a commutative semigroup w.r.t composition (a semigroup is closed under the operation, is associative and has a zero element). This

means that an image at a coarse scale t needs not necessarily to be computed directly from the original image at scale zero with the large scale parameter t and its wide Gaussian kernel support. It can alternatively be computed from an already smoothed image at a medium scale level r by applying a Gaussian kernel for the reduced scale $s = t - r$, the smaller support of which makes the convolution computationally less expensive.

- *Non-creation and non-enhancement of local extrema:*
  New image structures like local extrema (for any order of derivative) are not created but can only disappear when increasing the scale level. Furthermore, existing local extrema cannot be enhanced but only diminished with increasing scale. Since the convolution with a Gaussian kernel can also be interpreted as solving the isotropic diffusion (heat) equation (see Section 4.6), a physical interpretation of this property can also be given: A hot spot on a finite domain (e.g. a 1-dimensional metal stick as in Figure 4.4) will not become hotter but cool down with increasing time, and a cold spot will warm up till it has reached the equilibrium temperature of the domain.

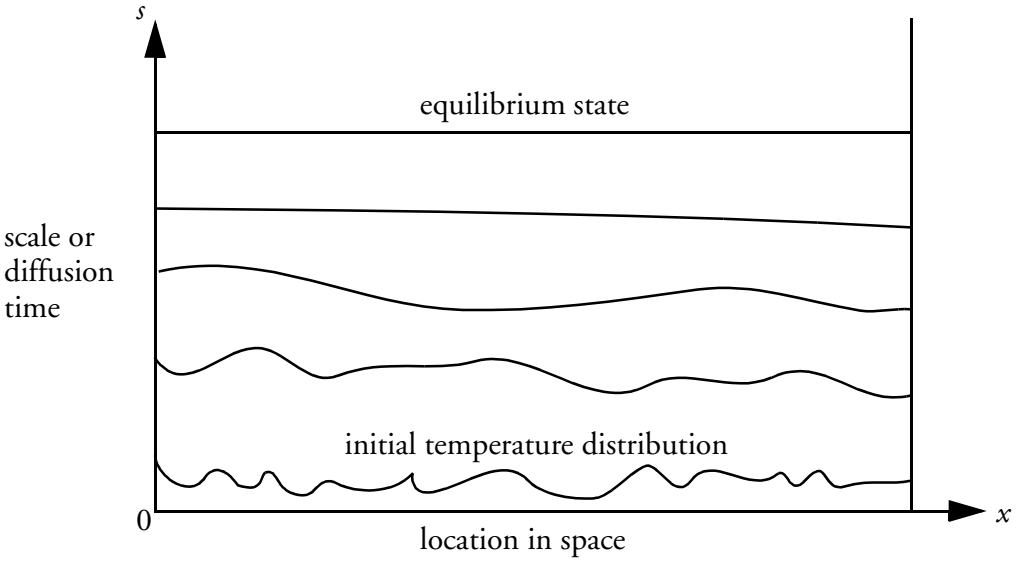


FIGURE 4.4 Non-creation of local extrema (application to a 1-dimensional signal) (sketch according to Lindeberg [Lin94]).

Based on these requirements, a *linear scale-space* representation can be defined by adding an additional scale axis (corresponding to the standard deviation of the Gaussian kernels) to the spatial axes of the computational domain. A formal definition of the linear scale-space will be given in Section 4.6.

Nonlinear scale-spaces can be obtained by relaxing some of the requirements. A famous example is Perona and Malik's scale-space based on *anisotropic* diffusion, which supports operators for edge detection [PM87]. Diewald, Preusser and Rumpf [DPR00] also use anisotropic diffusion to smooth images while preserving their edges. Their method is restricted to uniform grids. We will in the following restrict ourselves to the *isotropic linear* scale-space, since it better fits to our needs for flow feature detection (see Section 4.5).

4.3 RELATION OF SCALE-SPACE TO FOURIER THEORY

Although we regard the scale-space representation as a stack of images where every image has the same spatial resolution, we could in practice reduce the memory requirements by storing "smoother" images with less resolution. (In our case, we avoided this because for unstructured grids, it would have induced additional complexity, see Section 4.7). Let us compare the scale-space construction in the spatial domain and in the Fourier domain, there represented by frequencies. For reasons of simplicity, we restrict ourselves to 1-dimensional signals. We assume that the original signal $f(x)$ is continuous and *band-limited*, having a Fourier transform $F(\omega)$ with a maximum frequency ω_{max}, and that it has been sampled to a discrete signal. The sampled signal is not band-limited but has a periodic spectrum. However, we only need the "core spectrum" in the range of $[-\omega_{max},\omega_{max}]$ to reconstruct the signal, since the other frequencies are copies of the core spectrum and thus redundant. If the original signal was sampled above the *Nyquist frequency* ($\omega_{sample} \geq 2 \cdot \omega_{max}$) due to *Shannon's sampling theorem* [Sha49], these copies do not overlap. For an ideal reconstruction of the signal, we can use a box filter in the interval $[-\omega_{max},\omega_{max}]$. Due to the *convolution theorem*, a convolution of two signals in the spatial domain corresponds to a multiplication of the signals in the frequency domain. Multiplying the Fourier transform of the original signal with the rectangular box filter corresponds to convolving the original signal in the spatial domain with a *sinc* function of type

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x} , \tag{4.4}$$

because the *sinc* function is the Fourier transform of a *rect* function and vice versa (see Figure 4.5). Of course the *sinc* function deviates from the Gaussian used for constructing the scale-space. On the other hand, the shape of the *sinc* function is similar to the bell shape of the Gaussian kernel. Both filter kernels are axisymmetric, have infinite support and a maximum at the symmetry axis $x = 0$, and converge towards zero if $|x|$ tends towards infinity. The Fourier transform of a Gaussian kernel is again a Gaussian and thus a non-ideal low-pass filter (see Figure 4.6). The standard deviation σ_x of the spatial Gaussian is reciprocal to the standard deviation σ_ω of its counterpart in the frequency domain:

$$\sigma_x \cdot \sigma_\omega = 1 . \tag{4.5}$$

This *uncertainty relation* is closely related to the famous one from the field of quantum physics (Werner Heisenberg, 1927) dealing with location and momentum of a particle. In our case, the uncertainty relation says that the broader the Gaussian is in the spatial domain, the narrower it is in the frequency domain and vice versa (see Figure 4.6). This also matches our intuition because a smooth spatial function (wide Gaussian filter kernel, large σ_x) must have low frequencies (small σ_ω and thus a small ω_{max}).

In contrast to a box filter, the Fourier transform of a Gaussian has unlimited frequencies, but we can truncate it using the *three sigma rule*, which says that nearly all values (99.7 percent) occurring in a Gaussian distribution lie in the interval $[-3\sigma,3\sigma]$, where σ is the standard deviation. The truncation leads to a *band-limitation* of the smoothed signal to $\omega'_{max} = 3 \cdot \sigma_\omega = 3/\sigma_x$. According to the sampling theorem, we can then *subsample* the smoothed signal at a sampling rate of $2 \cdot \omega'_{max} = 6 \cdot \sigma_\omega = 6/\sigma_x$. The hereby induced error only causes a minimal *ringing* effect in the spatial domain. If we double the width σ_x of the Gaussian kernel for the next level in the scale-space, we can subsample the new, smoother signal by a factor of 2 in comparison to the previous signal.

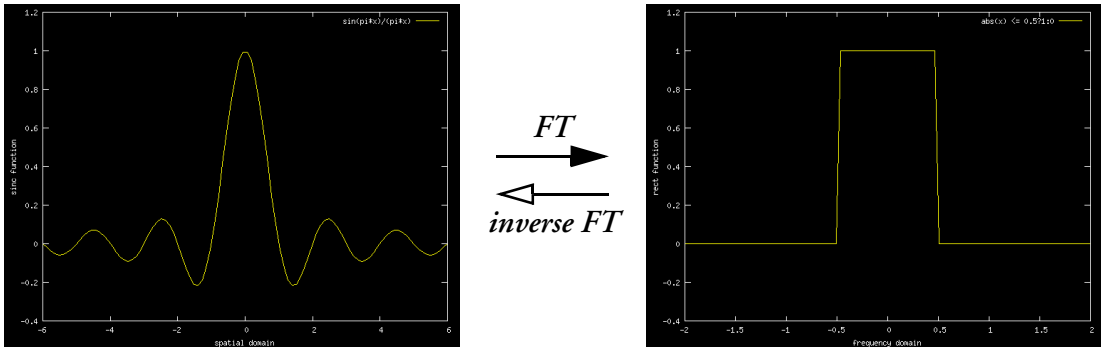


FIGURE 4.5 Correspondance of *sinc* and *rect* function in spatial and frequency domain. (Images created using Gnuplot version 4.0).

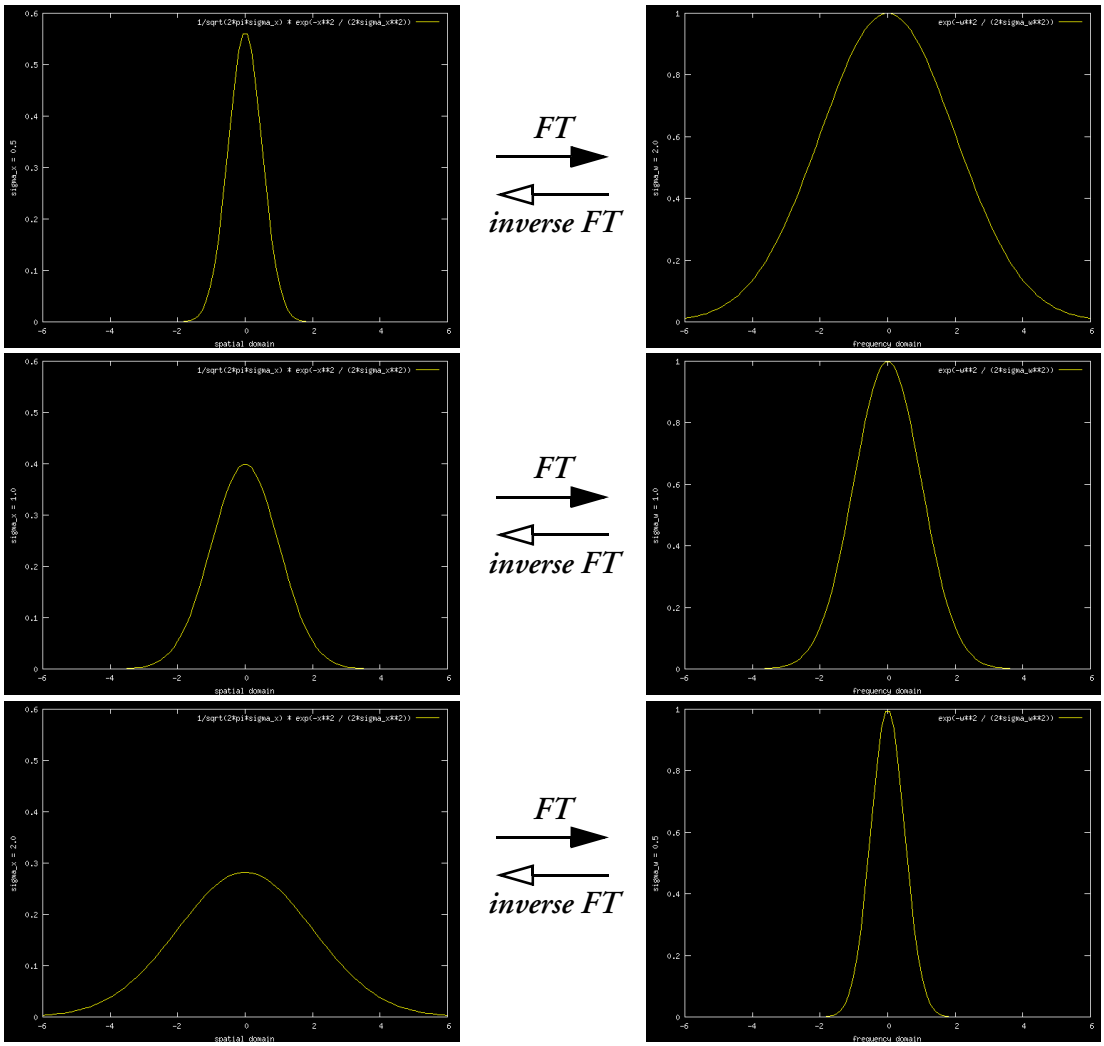


FIGURE 4.6 Correspondance of Gaussian functions in spatial and frequency domain. Left column: Gaussian in the spatial domain for $\sigma_x = 0.5, 1, 2$. Right column: Gaussian in frequency domain for $\sigma_\omega = 2, 1, 0.5$. (Images created using Gnuplot version 4.0).

4.4 APPLICATION TO IMAGE PROCESSING AND COMPUTER VISION

Scale-space representations have widely been used in image processing and computer vision for more than 20 years. They are especially useful for the important task of feature extraction. Since "feature" is only a general term for a variety of relevant objects contained in a (2D or 3D) image, features can be image structures of very different shape and size. It is therefore necessary to investigate a whole set of images at different levels of scale to extract, distinguish and track the features of interest.

Lindeberg [Lin94] shows that a measure for the "feature size" of a grey-level image structure can be computed from its scale-space representation after building the *scale-space primal sketch*. For a certain level of scale, a *grey-level blob* arises for each feature, mainly as a bright region on a dark background or vice versa. The blobs can therefore be output as a grey-level image. The observation now shows that when the scale level increases, a typical feature arises at a certain scale level and disappears at a higher scale level. Several features can merge into a bigger feature, as well as a feature can break up into several other features.

All these topology changes (*creation, annihilation, merge, split*) are called *blob events*. The difference of the creation scale and annihilation scale of a certain feature is a measure for its *scale extent*. Features have therefore a spatial as well as a scale extent (see Figure 4.7). Since the grey-level blob is an n-dimensional object (e.g. a 2D image region), the sum (integral) of the grey-level blobs along the 1-dimensional scale axis forms an $(n+1)$-dimensional volume which is called *scale-space blob*, the size of which can be regarded as a measure for the feature size.

Figure 4.8 shows a typical example of feature extraction in 2D images. The upper left picture is the original image, which corresponds to the raw data of the input signal $f(\boldsymbol{x})$ at scale level zero (see Equation 4.7). To the right of the original image, the grey-level blobs computed for this image are shown. The procedure has been repeated for increasing levels of scale, using Gaussian convolution for successively smoothing the original image.

As is clearly visible, the features of the original image (where the inner scale is just one pixel) have very small size and are merely noise. With increasing scale level, the feature size increases, too. The second and third pair of pictures, for example, emphasise the small structures like keys of the telephone console and pocket calculator. The noise is already strongly reduced in these images. This can also be verified in the corresponding grey-level blob images, which hardly contain single points anymore. Instead, the keys of the phone and calculator can clearly be identified.

In the fourth and fifth picture row, the keys of the two devices have melted to contiguous blocks for the observer's eye. Likewise, the small grey-level blobs of the single keys melt to bigger grey-level blobs - two new features have arisen out of many small ones. However, even these bigger features have a bounded scale extent, since their "lifetime" along the scale axis is limited. The sixth picture row shows that the medium-sized features (key blocks) have merged into just three big features (like the receiver and cable of the phone), whereas all small features have disappeared.

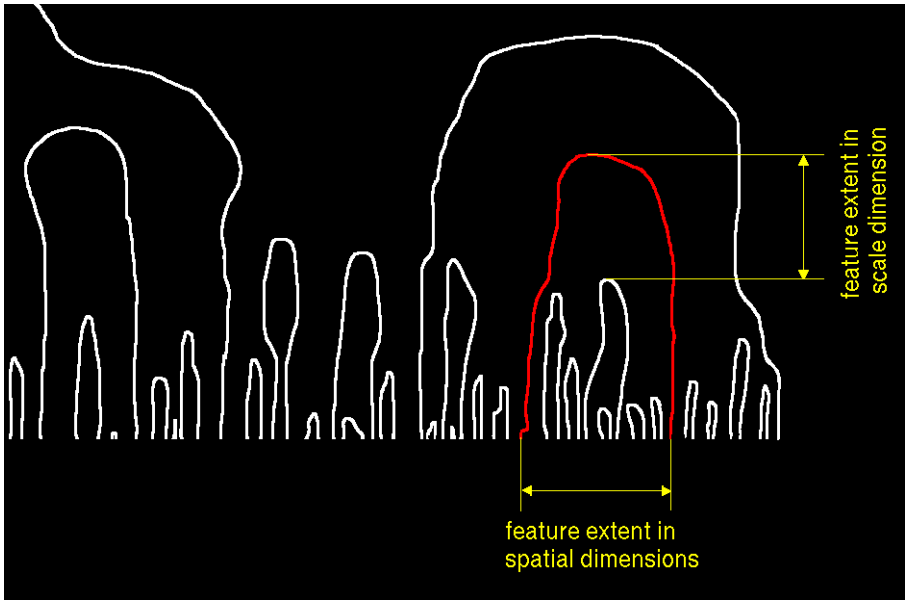


FIGURE 4.7 Feature extents in spatial and scale dimension. Indicated feature comprises smaller features and is contained by a bigger feature itself (image courtesy of T. Lindeberg [Lin94]).

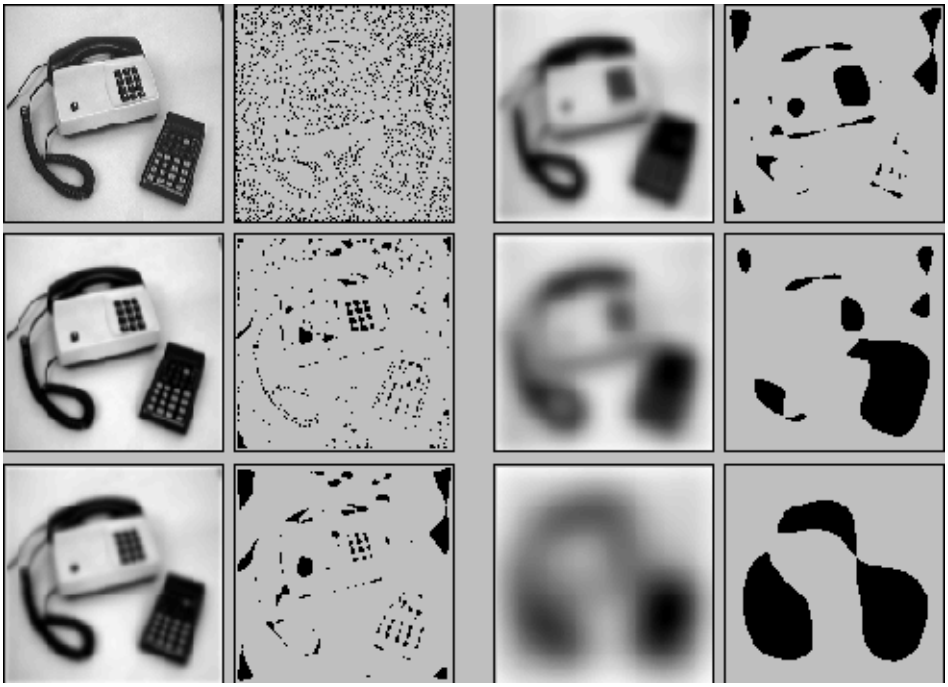


FIGURE 4.8 Gaussian smoothing and scale-space computation for images. Grey-level blobs at different scale levels indicate increasing feature sizes (image courtesy of T. Lindeberg [Lin94]).

4.5 APPLICATION TO FLOW VISUALISATION

As was mentioned in the motivation of this thesis (Section 1.2), automatic extraction of features is a promising strategy to cope with the large amount of data produced by time-dependent CFD simulations. The computational time for this type of simulations is typically in the order of days, which justifies the time spent on post-processing the data by extracting features in a batch run. Feature extraction can achieve data reductions of up to 1:10000 [Ken98] while still permitting the visualisation of essential parts of the flow. It can be used for visually browsing the results or in conjunction with other visualisation methods. However, the implementation of this strategy leads to the following problems:

1. Many flow features are of fractal nature, that means that there is no unique definition of their feature size. Depending on the observation scale, different sets of features can be observed. Usually, the scale is implicitly defined when an extraction method is designed. It would be preferable to let the user specify the scale interactively while viewing the result data.
2. Most methods require numerical computation of first- and second-order spatial derivatives, which causes the data to be roughened. Smoothing the data can reduce this effect, but it is not trivial to find an appropriate smoothing kernel when dealing with unstructured grids and highly varying cell sizes, which often occur in CFD datasets.
3. When features are extracted from time-dependent data, animating them can cause popping effects with features suddenly appearing or disappearing. These artifacts can be reduced by incorporating temporal in addition to spatial smoothing.

As we have seen in Section 4.4, scale-space techniques can successfully be applied to the field of computer vision, where feature extraction has already been practised for a long time. These techniques smooth the data using Gaussian kernels, the standard deviation σ of which can be any positive real number. The *scale axis* plus the spatial axes span the scale-space. Features thus not only have spatial extents but also a certain scale extent. They can therefore be searched and found in scale-space. This technique has the potential to solve the problems mentioned above for the flow visualisation field, too.

However, for flow visualisation applications, some additional adaptations must be made: Firstly, a discretised smoothing operation must be provided also for curvilinear and unstructured grids. And secondly, special attention has to be paid to boundary treatment, due to the relatively low resolutions in CFD datasets (often having interior boundaries) and the importance of the flow behaviour near material boundaries.

Many of the published "multi-scale" techniques are indeed multi-resolution, e.g. Luerig, Westermann et al. [LGE97, WE97] performed feature extraction from volumetric data in scale-spaces based on wavelets. However, to keep the spatial resolution, we do not construct a multi-resolution pyramid. Also, such a pyramid would yield a coarse sampling along the scale axis (see Section 4.1). Yet for the purpose of feature detection and tracking, it is preferable to have no prescribed sampling.

The idea of using the scale-space for visualisation of vector fields is not entirely new. Diewald et al. [DPR00] demonstrate the usefulness of anisotropic diffusion for the visualisation of vector fields. By successively smoothing the data, their scale-space can be visually explored. Our goals are, beyond visual exploration, to *algorithmically* extract features as geometric objects and to improve this process by exploiting the multi-scale nature of the features. And since we do not prefer certain spatial directions for searching flow features, we will use an isotropic version of the scale-space.

Scale-space analysis can enhance flow visualisation in many ways:

1. At larger scales, the set of features is reduced to fewer and clearer features. This is particularly useful for features defined by higher-order derivatives, such as the vortex criteria of Miura and Kida [MK97] and of Roth and Peikert [RP98].
2. It becomes possible to focus on features of a certain scale.
3. *Tracking* features over scale allows for visualising them with the positional accuracy of small scales and simultaneously deriving connectivity information from larger scales.
4. *Selective* visualisation can be done by picking an individual feature at a larger scale which is then tracked to a smaller scale and finally tracked over time.

All said above applies to flow features of any dimensionality. However, we will focus on 1D (line-type) features, for the reason that the CFD datasets computed by our industry partners are mostly visualised for the purpose of studying vortices. (Vortices reduce a machine's efficiency by binding energy. Also, an unstable vortex can interact with machine parts, producing undesired effects like material abrasion or resonance).

The following section presents a formal definition of the scale-space representation for *continuous* data signals. The last section then discusses several methods to numerically compute the scale-space for *discrete* structured 3-dimensional grids, plus their benefits and drawbacks. As a first major contribution of this Ph.D. thesis, we will also present a method to compute the linear isotropic scale-space for *unstructured* grids, which are most common in CFD datasets and flow visualisation.

4.6 SCALE-SPACE DEFINITION

The *linear isotropic scale-space* of a given n-dimensional physical space $X \subseteq \mathbb{R}^n$ and a continuous scalar data field $f(\boldsymbol{x})$ given on this domain is defined as the $(n+1)$-dimensional space $X \times \mathbb{R}_0^+$ with data

$$u(\boldsymbol{x}, s) = f(\boldsymbol{x}) \otimes G(\boldsymbol{x}, \sqrt{2s}) \tag{4.6}$$

and initial condition (original data signal)

$$u(\boldsymbol{x}, 0) = f(\boldsymbol{x}), \tag{4.7}$$

where $s = \sigma^2/2$ is the real-valued scale ($s \geq 0$). The (n-dimensional) Gaussian kernel of standard deviation σ computes to

$$G(\boldsymbol{x}, \sigma) = \frac{1}{\sqrt{(2\pi\sigma^2)^n}} \cdot e^{-\frac{\boldsymbol{x}^2}{2\sigma^2}}. \tag{4.8}$$

Of course the scale-space can be constructed from this definition in a straightforward manner by successively applying Gaussian kernels to the original data. However, convolving a scalar field $f(\boldsymbol{x})$ with a Gaussian of standard deviation σ is equivalent to solving the *isotropic diffusion equation* (or *heat equation*), which contains the first temporal derivative (written as dot-notation) and second spatial derivatives (Laplacian ∇^2) of the data signal:

$$\dot{u}(\boldsymbol{x}, s) = \nabla^2 u(\boldsymbol{x}, s) \tag{4.9}$$

for *diffusion time* $s = \sigma^2/2$ and initial condition

$$u(\boldsymbol{x}, 0) = f(\boldsymbol{x}). \tag{4.10}$$

Remark:

The reason for calling the diffusion time variable s rather than t is that we want to do scale-space analysis also for time-dependent data such as unsteady flow fields. We then have two orthogonal time axes, namely the "physical time" t and the "diffusion time" s which by definition of the scale-space (Equation 4.6) is equal to the scale parameter.

4.7 SCALE-SPACE COMPUTATION

The formal definition of Section 4.6 assumes a *continuous* data field defined in a continuous computational domain. To actually compute the scale-space for a *discrete* grid (as used in scientific visualisation) and for discrete scale levels, we need discretised versions of the convolution operator or of the diffusion equation, respectively. In this section, we will discuss such methods for the case of structured (e.g. uniform) and unstructured grids, as well as the computation of derivatives.

4.7.1 Structured Grids

On a structured grid, there are at least three basically different methods for smoothing data with a Gaussian filter kernel:

1. The obvious one is to actually compute the convolution with a discrete sampled Gaussian, which can be performed either in the spatial domain or in the frequency domain. For unstructured grids, this requires additional efforts, e.g. a uniform resampling and trilinear interpolation of the data, which increases the numerical imponderabilities.

2. The second method is to repeatedly apply for each dimension a binomial filter with weights (1/4, 1/2, 1/4). The weights of this *recursive filter* are (up to normalisation) the even rows of the Pascal triangle and thus converge to a Gaussian filter kernel. This method seems to be more adequate to the computation of a scale-space where a whole sequence of Gaussians is needed. However, the main problem with this approach is that the number of iterations is proportional to s and thus to σ^2.

3. The third method is to discretise the diffusion (heat) equation. While this amounts to numerically solving a partial differential equation, it has the advantage to work well also for large σ values, and furthermore to extend properly to unstructured grids. For the special case of a uniform grid, the spatial discretisation of the diffusion equation can be performed e.g. by *finite differences* to approximate the Laplacian of Equation 4.9. In the 1-dimensional case, the filter mask is of the form $(1, -2, 1)$. Higher-dimensional Laplace operators are built analogously, e.g. the 2-dimensional filter mask used for image processing is

$$\nabla^2 u = u_{xx} + u_{yy} \approx \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{4.11}$$

4.7.2 Unstructured Grids

The grids typically used in turbomachinery CFD consist of hexahedral cells and are either block-structured or unstructured. We will focus on the latter ones as they comprise the former ones. However, using a discretised smoothing kernel like the Gaussian one is not feasible for unstructured grids because of their irregular geometry. We must therefore discretise and solve the diffusion equation to compute the scale-space on unstructured grids.

For the spatial discretisation, one possibility is to approximate the Laplacian $\nabla^2 u(\boldsymbol{x}, s)$ of Equation 4.9. An example for a discrete Laplacian on unstructured grids is the *umbrella operator* proposed by Taubin [Tau95] and refined by Desbrun [DMSB99], which we used for the purpose of mesh fairing (Section 6.2.5), also treated by Guskov et al. [GSS99]. An alternative would be to use piecewise polynomial surfaces like Michelli splines [Mic94] for approximating the Laplacians at the grid nodes.

We decided to rather discretise the complete diffusion equation by applying the *finite element method* (FEM) pioneered by Zienkiewicz [ZT00] to the grid cells. This type of general hexahedral elements is usually referred to as *isoparametric* elements. They are unit cubes when expressed in local coordinates (r, s, t), see Figure 4.10. The basic procedure for a 1-dimensional scalar signal works as follows: We regard the scalar dataset

$$(u_0, u_1, \ldots, u_{N-1}) \tag{4.12}$$

defined on N nodes and interpolate it between the nodes using the linear basis functions

$$\Phi_i(x) = \left\{ \begin{array}{ccc} x-(i-1) & , & x \in [i-1,\, i] \\ (i+1)-x & , & x \in [i,\, i+1] \\ 0 & , & else \end{array} \right\} \qquad \forall i \in \{0, \ldots, N-1\}, \tag{4.13}$$

which are also called *hat functions* due to their triangular shape (see Figure 4.9). At any point location x in computational space, the hat functions sum up to one (*partition of unity*):

$$\sum_{i=0}^{N-1} \Phi_i(x) = 1 \qquad \forall x \in [0, N-1], \tag{4.14}$$

so they can be used as weighting functions for the scalar values of the grid nodes. The scalar value at location x can therefore be interpolated by the linear combination

$$u(x) = \sum_{i=0}^{N-1} u_i \cdot \Phi_i(x) \qquad (u, x, \Phi \in \mathbb{R}, x \in [0, N-1]). \tag{4.15}$$

Note that due to the small support of the hat functions, only direct neighbours of a certain node have influence on the scalar value near that node. Let us return to the continuous diffusion equation in the 3-dimensional case:

$$\dot{u}(\boldsymbol{x}) = \nabla^2 u(\boldsymbol{x}). \tag{4.16}$$

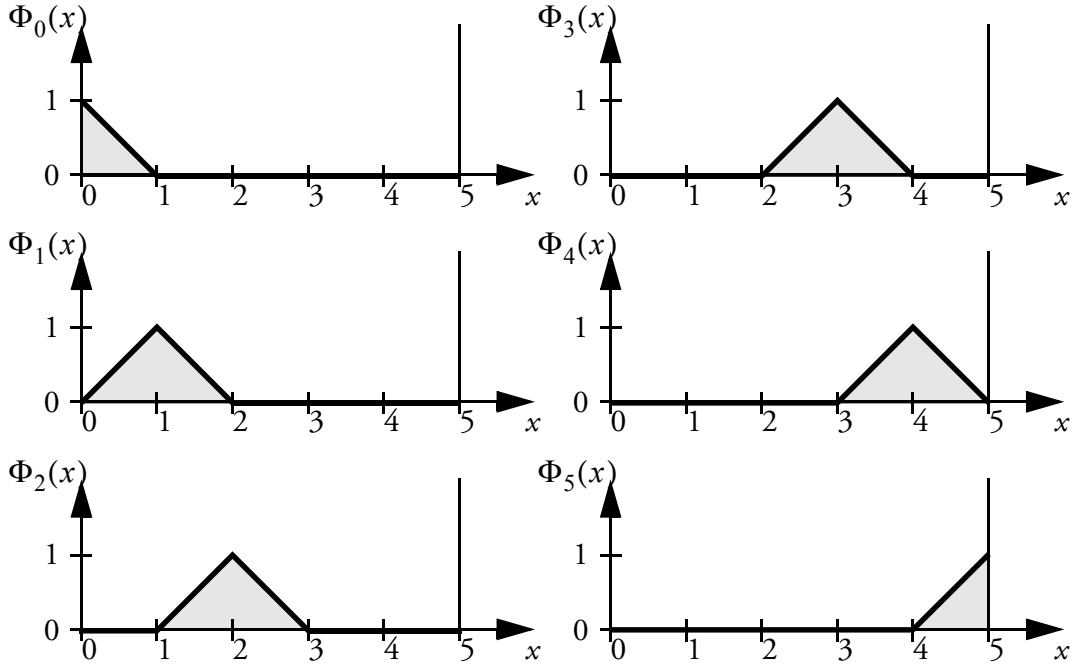


FIGURE 4.9 Hat functions as a simple example of finite elements (1D signal, N=6).

We can multiply this equation by any hat function $\Phi_j(\boldsymbol{x})$ as test function:

$$\dot{u}(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) = \nabla^2 u(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}), \tag{4.17}$$

and integrate over an arbitrary subset G of the computational domain:

$$\int_G \dot{u}(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x} = \int_G \nabla^2 u(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x}. \tag{4.18}$$

The diffusion equation can now be spatially discretised by inserting the interpolation function (Equation 4.15) into Equation 4.18:

$$\int_G \left(\sum_{i=0}^{N-1} \dot{u}_i \cdot \Phi_i(\boldsymbol{x}) \right) \Phi_j(\boldsymbol{x}) d\boldsymbol{x} = \int_G \left(\sum_{i=0}^{N-1} u_i \cdot \nabla^2 \Phi_i(\boldsymbol{x}) \right) \Phi_j(\boldsymbol{x}) d\boldsymbol{x}. \tag{4.19}$$

By swapping the sums and integrals on both sides, we get

$$\sum_{i=0}^{N-1} \dot{u}_i \int_G \Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x} = \sum_{i=0}^{N-1} u_i \int_G \nabla^2 \Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x}. \tag{4.20}$$

This can be written shorter using the abbreviations

$$a_{ij} = \int_G \Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x} \tag{4.21}$$

and

$$b_{ij} = \int_G \nabla^2 \Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x}, \tag{4.22}$$

resulting in the N equations

$$\sum_{i=0}^{N-1} a_{ij} \cdot \dot{u}_i = \sum_{i=0}^{N-1} b_{ij} \cdot u_i \qquad \forall j \in [0, N-1]. \tag{4.23}$$

The original *partial* differential equation (Equation 4.16) can therefore be solved by a system of *ordinary differential equations* (ODEs):

$$\boldsymbol{A}\dot{\boldsymbol{u}} = \boldsymbol{B}\boldsymbol{u}, \tag{4.24}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are the so-called *element mass matrix* and *element stiffness matrix*. These two matrices only depend on the grid geometry and therefore are constant throughout the smoothing process. As can easily be seen from Equation 4.21 and Equation 4.22, the matrix $\boldsymbol{A}$ is always symmetric, whereas $\boldsymbol{B}$ is in general not.

We decided to solve the ODE system using the *implicit* Euler method, since it is numerically superior to the explicit Euler and allows greater time steps τ. The spatially and temporally discretised diffusion equation is then

$$\boldsymbol{A}\frac{\boldsymbol{u}(t+\tau) - \boldsymbol{u}(t)}{\tau} = \boldsymbol{B}\boldsymbol{u}(t+\tau). \tag{4.25}$$

Multiplying by τ and rearranging the terms yields

$$(\boldsymbol{A} - \tau\boldsymbol{B})\boldsymbol{u}(t+\tau) = \boldsymbol{A}\boldsymbol{u}(t). \tag{4.26}$$

If the time step is kept constant throughout the computation, the matrix $\boldsymbol{A} - \tau\boldsymbol{B}$ can be precomputed like $\boldsymbol{A}$ and $\boldsymbol{B}$. Computing the smoothed scalar dataset $\boldsymbol{u}(t+\tau)$ from the previous dataset $\boldsymbol{u}(t)$ is equal to solving a linear equation system of type

$$\boldsymbol{M}\boldsymbol{x} = \boldsymbol{c} \qquad (\boldsymbol{M} \in \mathbb{R}^{N \times N}, \boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^N). \tag{4.27}$$

The question remains what boundary conditions the diffusion equation should have, and how this influences the properties of the matrices and thus the computational effort of the equation solving process.

An obvious choice for boundary conditions are *symmetric* boundary conditions, where the normal derivative is forced to be zero. This is the simplest form of a Neumann boundary condition. It says that diffusion can happen unrestrictedly along the boundary of the computational grid, but no diffusion is allowed across the boundary.

For symmetric boundary conditions, we can simplify the $\boldsymbol{B}$ matrix. Partial integration of the right side of Equation 4.22 yields

$$\int_G \nabla^2 \Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) d\boldsymbol{x} = \nabla\Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x}) - \int_G \nabla\Phi_i(\boldsymbol{x}) \bullet \nabla\Phi_j(\boldsymbol{x}) d\boldsymbol{x}. \tag{4.28}$$

The symmetric boundary condition makes the term $\nabla\Phi_i(\boldsymbol{x}) \cdot \Phi_j(\boldsymbol{x})$ zero, and the elements of the stiffness matrix can thus be computed as

$$b_{ij} = -\int_G \nabla\Phi_i(\boldsymbol{x}) \bullet \nabla\Phi_j(\boldsymbol{x}) d\boldsymbol{x}. \tag{4.29}$$

As a consequence of this, we get rid of the numerically unfavourable second derivatives. Even more important, the matrix $\boldsymbol{B}$ is now symmetric like the matrix $\boldsymbol{A}$. The choice of the symmetric boundary condition has therefore a positive effect on the computation: the storage requirements for the matrices are reduced, and more efficient equation solvers are available, which reduce the computational time (see Section 4.8 for detailed results).

The question remains how to compute the entries a_{ij} and b_{ij} of the element mass matrix and element stiffness matrix. Each element is defined for two nodes of the grid, whose global node numbers are i and j. However, a node is in general contained in several adjacent cells (see Figure 4.10, where node (000) and node (110) are shared by two cells lying on top of each other). Therefore we must loop over every cell of the grid, and for each node pair of the current cell we must compute the contribution of this cell to the matrix elements of this node pair.

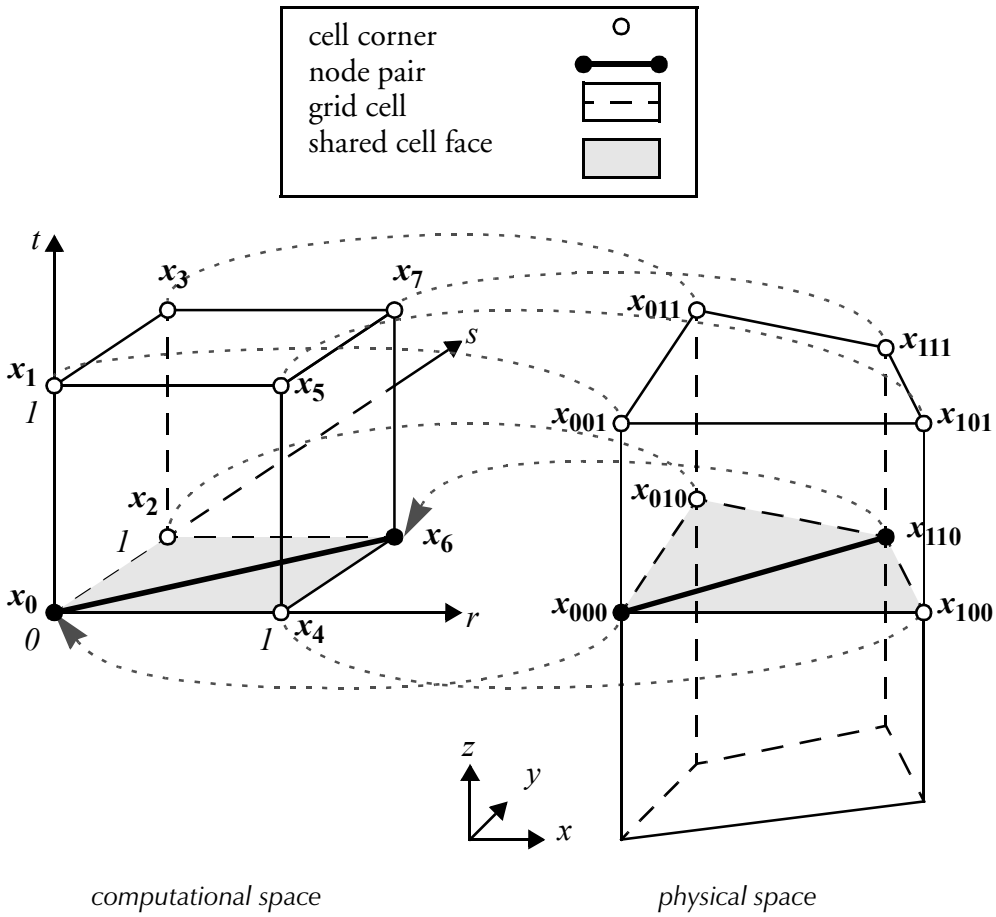


FIGURE 4.10 Sharing of nodes by adjacent grid cells, indexing scheme of the local cell corners and mapping of a grid cell from physical to computational space.

Since integration over the hexahedral elements would have been very complex in global coordinates, the actual computation of the integrals was done in local coordinates. Assume that the 8 corners of a hexahedral grid cell can be decimally indexed $(0, 1, \ldots, 7)$ or binary indexed $(000, 001, \ldots, 111)$, see Figure 4.10. Let their physical (x, y, z) coordinates be $(\boldsymbol{x}_{000}, \boldsymbol{x}_{001}, \ldots, \boldsymbol{x}_{111})$. For the coordinate transformation between computational and physical space in that cell we use the *trilinear interpolation*

$$\boldsymbol{x}(r, s, t) = \sum_{k=0}^{1} \sum_{l=0}^{1} \sum_{m=0}^{1} r^k s^l t^m \cdot (1-r)^{1-k} (1-s)^{1-l} (1-t)^{1-m} \cdot \boldsymbol{x}_{klm}. \quad (4.30)$$

There are 8 local hat functions defined in the cell domain:

$$\varphi_{klm}(r, s, t) = r^k s^l t^m \cdot (1-r)^{1-k} (1-s)^{1-l} (1-t)^{1-m}. \quad (4.31)$$

where $(k, l, m) \in \{0, 1\}^3$ and $(r, s, t) \in [0, 1]^3$. The contribution $\bar{a}_{i'j'}$ of the current cell to the element mass matrix for the two local cell corners i' and j' is then

$$\bar{a}_{i'j'} = \int_0^1 \int_0^1 \int_0^1 \varphi_{i'}(r, s, t) \cdot \varphi_{j'}(r, s, t) \cdot \det(\boldsymbol{J})\, dr\, ds\, dt, \quad (4.32)$$

where $\varphi_{i'}(r, s, t)$ and $\varphi_{j'}(r, s, t)$ are the hat functions for the local node numbers i', j' and

$$\boldsymbol{J} = \frac{\partial(x, y, z)}{\partial(r, s, t)} \quad (4.33)$$

is the Jacobian of the coordinate transformation function from Equation 4.30. The contribution $\bar{b}_{i'j'}$ of the current cell to the element stiffness matrix for the two local cell corners i' and j' is

$$\bar{b}_{i'j'} = -\int_0^1 \int_0^1 \int_0^1 (\boldsymbol{J}^{-1} \nabla' \varphi_{i'}(r, s, t)) \bullet (\boldsymbol{J}^{-1} \nabla' \varphi_{j'}(r, s, t)) \cdot \mathrm{Det}(\boldsymbol{J})\, dr\, ds\, dt, \quad (4.34)$$

where

$$\nabla' = \left(\frac{\partial}{\partial r}, \frac{\partial}{\partial s}, \frac{\partial}{\partial t} \right) \quad (4.35)$$

means the gradient of the hat function w.r.t. the local coordinates (r, s, t). We numerically compute the integrals of Equation 4.32 and Equation 4.34 using *Gauss quadrature*. In the end, our implementation loops through all grid cells and locally computes the FE integrals $\bar{a}_{i'j'}$ and $\bar{b}_{i'j'}$ between any two local nodes i' and j' of the current cell. The result for two certain nodes is assigned to their global node numbers i, j and added to the matrix entries a_{ij} and b_{ij} of the mass element matrix and mass stiffness matrix.

4.7.3 Computing Derivatives

Many feature extraction techniques, be it in computer vision or in scientific flow visualisation, require - in addition to the data fields - their first and sometimes second spatial derivatives. The need for smoothing increases with the order of the derivative, since the smoothing must compensate for the roughening effect of the numerical differentiation. A basic property of the Gaussian convolution operator is that it commutes with the differentiation operation:

$$\frac{\partial}{\partial \boldsymbol{x}}(u \otimes G) \;=\; \frac{\partial u}{\partial \boldsymbol{x}} \otimes G \;=\; u \otimes \frac{\partial G}{\partial \boldsymbol{x}}, \tag{4.36}$$

which provides us with three different ways to compute derivatives of the smoothed data. The left term corresponds to differentiating the smoothed data. This is our preferred approach because it is numerically clearly better than smoothing the differentiated data as reflected by the middle term. It also makes differentiation a simpler task which can be done with minimal stencils. Also, its runtime is shorter since we can do the time-consuming convolution as a preprocessing step and save its results to disk. This fact also argues against the use of sampled derivatives of the Gaussian which are then convolved with the data, according to the right term. This computing strategy makes particular sense in an environment where smoothing is generally done with sampled Gaussians. However, for the reasons mentioned above and in Section 4.7.1, we discarded this option.

Our actual implementation uses a *least squares fit* (LSF) based on 1-neighbourhoods for computing derivatives of presmoothed scalar and vector fields (e.g. the velocity and pressure gradient). To approximate the gradient of a scalar field s at a central node (x_0, y_0, z_0), the field is assumed to be linear in a local environment of the central node:

$$s(x, y, z) \;=\; d(x - x_0) + e(y - y_0) + f(z - z_0). \tag{4.37}$$

Estimating the scalar values of the K direct neighbours of the central node yields the errors

$$s(x_i, y_i, z_i) - s_i \;=\; d(x_i - x_0) + e(y_i - y_0) + f(z_i - z_0) - s_i \tag{4.38}$$

(for each neighbour, $i \in \{1, 2, \ldots, K\}$). The sum of the squared errors

$$\sum_{i=1}^{K} \left(d(x_i - x_0) + e(y_i - y_0) + f(z_i - z_0) - s_i\right)^2 \tag{4.39}$$

can be minimised in a straightforward manner by setting its gradient to zero, namely its derivatives w.r.t the parameters (d, e, f). This leads to a linear equation system of 3 *normal equations* with 3 unknowns, the solution (d, e, f) of which is the gradient of the linearised scalar field and thus an approximation for the gradient at the central node.

If the grid cells are non-degenerate, this method is numerically sufficient for the 3×3 equation systems occurring here. For degenerate grid cells with acute angles, it might be necessary to avoid the normal equations, e.g. using *Givens* transforms, *Householder* transforms or *singular value decompositions* (SVD) to solve the error equations (see [Sch97]).

The Jacobian of a 3D vector field is computed analogously by regarding the three vector components as scalar fields and repeating the above procedure for all of them.

4.8 RESULTS

In this section, we give performance results for the scale-space computation on two time-dependent industrial datasets [BP02], both discretised on unstructured hexahedral grids. The first dataset derives from a coupled unsteady simulation (by VA Tech Hydro) of the runner and draft tube of a Francis turbine. The second and third dataset originate from an unsteady simulation (by Sulzer Pumpen) of a mixed-flow pump. All three datasets will also be investigated w.r.t. feature extraction in Chapter 5. In all three grids, the hexahedral cells served as an FE discretisation. We list the CPU times for computing the two FE matrices and for smoothing the data in a single step, for three values of the standard deviation σ. All computations were done on an SGI Octane (640 MB main memory, MIPS R10000 CPU and R10010 FPU running at 250 MHz). We computed the FE matrix elements using Gauss quadrature ($3 \times 3 \times 3$ Gauss points) and solved the linear equation systems using the *f11jef* function from the *NAG* Fortran library [NAG01], applying the symmetric *Lanczos* method and preconditioning the systems by symmetric successive overrelaxation.

Setting up the matrices turned out to be quite expensive, taking several minutes per matrix (see Table 4.1). However, the matrices are valid for all datasets given on the same grid, since they contain purely geometric information. The smoothing times scaled less than linearly with σ. If an explicit Euler scheme were used, they would scale quadratically because of $s = \sigma^2/2$ and the limited step size. We tested the implicit method for strong smoothing ($\sigma = 0.032$) in a single step and in several intermediate steps, finding no visual impact regarding feature extraction (see Figure 5.15 and 5.16). Therefore the accuracy of the implicit Euler method was sufficient for large integration steps.

The previous sections showed the specific properties of the scale-space representation, its usefulness for flow visualisation, and how to compute a linear isotropic scale-space for unstructured grids as used in CFD. Based on these precomputed scale-spaces, we will in the next chapter present some known and newly developed feature extraction methods, and demonstrate how these algorithms can benefit from the scale-space framework.

turbine design	nodes	grid extent	sigma	matrix setup	smoothing	
				CPU[s]	CPU[s]	iterations
draft tube	654770	1.33 x 0.59 x 0.72	0.002	531.8	41.2	9
			0.008		88.5	23
			0.032		219.6	62
pump stator	310757	0.50 x 0.50 x 1.54	0.002	242.3	19.2	9
			0.008		31.8	17
			0.032		67.9	40
pump rotor	235223	0.33 x 0.33 x 0.19	0.002	185.4	18.1	12
			0.008		31.2	23
			0.032		61.1	48

TABLE 4.1 Performance analysis of scale-space computation.

CHAPTER 5

FEATURE EXTRACTION AND TRACKING

Scale-space analysis can be done in conjunction with virtually any scientific visualisation technique, in particular with locally operating methods (such as isosurfaces) and methods that include derivatives of the data fields. But like in the field of image processing (see [Lin94]), it is for flow visualisation most successfully combined with *feature extraction* techniques [BP02]. Features of interest can be of various types and dimensions, but since our main application is the extraction of vortex core lines from CFD datasets, we will in this chapter focus on the detection of line-type features.

This chapter recapitulates a general 3D vortex core line extraction algorithm introduced by Roth and Peikert [RP99] and describes the simplifications and improvements which we made in the context of a scale-space analysis. In particular, we take a glance onto the data structures, some techniques for speed-up of the computation, some mathematical background and heuristics for preparing the results.

It is then shown how a novel 4D extension of this algorithm is able to *track* vortices in either the temporal or the scale domain. In former approaches, features were mostly tracked *after* their extraction, using proximity-based and heuristic methods like spatial overlap [SW96] or shape attributes [RPS99] to relate them to each other. Our basic idea is, in contrast, to *combine* feature extraction and tracking within the same algorithmic process. This is achieved by *lifting* the computational grid as well as the resulting features by one dimension in order to shift the feature tracking from the postprocessing phase of the feature extraction to the feature extraction itself.

We will illustrate the lifting of unstructured 3-dimensional grid cells to 4-dimensional hypercubes as well as the lifting from 1-dimensional line-type features to 2-dimensional feature meshes. The most remarkable improvement of our novel tracking method is its ability to correctly carry the connectivity of the features over cell boundaries, even for fast-moving features which traverse more than one grid cell per time step. We will motivate the topological correctness of this approach and, at the end of this chapter, give both numerical and visual results of the methods presented here.

5.1 VORTEX CORE LINE EXTRACTION

The vortex core line extraction algorithm we used for our implementation is based on the *parallel vectors* operator introduced by Roth and Peikert [RP99]. It is recapitulated here mainly because it will serve as a basis for our vortex *tracking* algorithm later presented in this chapter. Furthermore, it contains some modifications in comparison to the original version, why it was re-implemented from scratch for this dissertation.

As described in Section 3.4.2, the parallel vectors operator allows for specifying a variety of vortex core line definitions, e.g. the vortex criteria given by Levy et al., by Sujudi/Haimes, by Banks/Singer, by Miura/Kida, and by Strawn et al. (see Section 3.4.3 to Section 3.4.7). In a mathematical sense, the parallel vectors algorithm computes the set of points where two given vector fields $\boldsymbol{v}$ and $\boldsymbol{w}$ are parallel or antiparallel, i.e. where there exists some scalar value $\lambda \in \mathbb{R} \cup \{-\infty, \infty\}$ such that $\boldsymbol{v} = \lambda \boldsymbol{w}$.

We implemented this algorithm on a cell-by-cell basis by scanning all grid faces for intersection points with the vortex core lines, which are then connected to line segments traversing the grid cells. By connecting the line segments of neighbouring cells to polylines, the final vortex core lines are obtained. The composing is based on the velocity field $\boldsymbol{u}$.

Note that in an abstract sense, one could argue that the parallel vectors method actually operates on *three* vector fields $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$. However, the vector field $\boldsymbol{u}$ only plays a secondary role, namely for postprocessing the resulting vortex core lines. Besides, one of the vector fields $\boldsymbol{v}$ and $\boldsymbol{w}$ often is the velocity field $\boldsymbol{u}$ itself (e.g. when using the Levy or Sujudi/Haimes criterion).

So far, our vortex extraction method corresponds with one of the implementations described in Roth's dissertation [Rot00]. But since we were working in the context of a scale-space analysis, we simplified the procedure in two ways:

1. Since the data are presmoothed, estimating derivatives (e.g. for setting up the velocity Jacobian or pressure gradient field) can be done with a simple computational scheme and no extra filtering. To be more precise, an ordinary least square fit of the velocity or pressure gradient based on a one-neighbourhood stencil was sufficient for our purposes (see Section 4.7.3).

2. The connecting of the lines can be done more automatically and with fewer "quality" parameters. In the case of vortex core lines, one parameter turned out to be sufficient, namely the signed length ratio λ of the two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ at the intersection points found on the grid faces. Since "insignificant" features are eliminated automatically when increasing the scale s, other parameters used in former implementations [RP99], like the deviation of the core line tangent from the velocity direction (feature quality, see Section 5.2.5), are no longer needed for *constructing* the vortex core lines. We will only store these attributes for an eventual postprocessing, namely the *filtering* of resulting vortex core lines for graphical output.

We will in the following explain some implementation details of the 3D version of the vortex core line extraction algorithm. Afterwards, we will extend this procedure from 3 to 4 dimensions and present a novel tracking algorithm based on a computational grid and features "lifted" by one dimension.

5.2 EXTRACTION OF VORTEX CORES IN 3D

Our vortex core line extraction algorithm basically consists of the following steps:

1. **Compute the connectivity structure** of the underlying unstructured grid:
   Store relevant cell <-> face <-> edge <-> node relations in suitable data structures.
2. **Set up the two vector fields** $\boldsymbol{v}$ and $\boldsymbol{w}$, depending on the chosen vortex criterion:
   Load the physical flow fields, build derived fields, assign them to $\boldsymbol{v}$ and $\boldsymbol{w}$.
3. **Mark the intersected edges** of the grid:
   Test for every edge whether the cross product $\boldsymbol{v} \times \boldsymbol{w}$ changes its sign (componentwise) between the two end nodes (for a later trivial reject test of the grid faces).
4. **Find the points of parallel vectors** on all faces of the grid:
   Traverse the grid cell by cell, traverse each cell face by face.
   On each previously untreated face, compute all points where $\boldsymbol{v} = \lambda \boldsymbol{w}$.
   Store all solution points in an attributed vertex list, together with attributes.
   For each cell, connect the solution points to line segments traversing the cell.
   Orient the line segments according to mean velocity at their endpoints.
5. **Compute the feature quality** and vortex strength of every found vertex
   (for eventual filtering before graphical output)
6. **Generate and filter a polyline list** containing the resulting vortex core lines:
   Loop through the vertex list, store contiguous segments as polylines.
   Filter the resulting polylines by 4 quality parameters (optional):
   minimal vortex strength, minimal feature quality,
   minimal vertices per polyline, maximal exceptions per polyline.
7. **Output the resulting geometry** to the graphics renderer.

We will now describe some implementation aspects of the vortex core line extraction algorithm in more detail. Each of the following subsections is numbered according to its corresponding step in the algorithm listed above.

5.2.1 Computation of the connectivity structure for the unstructured grid

The unstructured grid only contains the information which global nodes belong to a certain grid cell. It does not enumerate the edges and faces of the grid, so we build a connectivity data structure which stores

- for every (global) node number:
  - the smallest edge/face number of the edges/faces containing the node,
- for every (global) edge number:
  - the smaller and greater node number of its two endpoints,
- for every (global) face number:
  - the smallest node number of its four corners, and
  - the node number of its counterpart lying diametrically opposed on the face.

In the vortex extraction algorithm, we loop through all faces of a cell. This is achieved by taking four cell corners at once and identifying the global edge and face numbers using the information mentioned above and performing small searching loops. We found this strategy a good trade-off between main memory requirements and computation time.

5.2.2 Setup of the two vector fields

We assume that the main memory available for our implementation is sufficient to load all physically meaningful flow fields (velocity, vorticity, pressure) for a certain time step at once. The two vector fields $\boldsymbol{v}$ and $\boldsymbol{w}$ can thus be represented by two pointers onto the data structures containing the vector fields (like velocity and vorticity). Depending on the chosen vortex extraction method, the two pointers are simply set to the appropriate vector fields according to the mapping of Table 3.2 in Section 3.4.8.

5.2.3 Marking of the intersected edges

The trivial reject test is based on the vector cross product $\boldsymbol{v} \times \boldsymbol{w}$. If the two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are parallel, their cross product is the zero vector:

$$\boldsymbol{v} = \lambda \boldsymbol{w} \quad \Rightarrow \quad \boldsymbol{v} \times \boldsymbol{w} = (0, 0, 0) \tag{5.1}$$

so we expect a change of sign for the three components of the cross product between the endpoints of an intersected edge. We thus check every edge of the grid and store the sign change information in 3 bits per edge. The test for sign changes of the three components of $\boldsymbol{v} \times \boldsymbol{w}$ can be performed by simple comparison of the two end nodes (assuming a linear interpolation of $\boldsymbol{v} \times \boldsymbol{w}$ along the edge) or by additionally checking for double intersections (assuming a quadratic interpolation of $\boldsymbol{v} \times \boldsymbol{w}$ along the edge).

Later in stage 4.) we first check for a given face whether edges of this face are intersected. If not, we can immediately reject the face and continue with the next one. Since a vortex core line in general intersects only few cells and faces of the grid, this trivial reject saves significant computational time.

5.2.4 Finding the points of parallel vectors on all faces

The actual search for solution points takes place in three nested loops:

```
for each cell of the grid
  for each of the 6 faces of the current cell
    Subdivide the face into 4 triangles.
    for each of the 4 triangles of the current face
      Find the parallel points on that triangle using the eigenvector method.
    next triangle
  next face
next cell
```

FIGURE 5.1 How to find the solution points on all faces of the grid.

The inner loop computes the midpoint $\boldsymbol{c_4}$ of the quadrangle face $(\boldsymbol{c_0 c_1 c_2 c_3})$ by averaging its four corners:

$$\boldsymbol{c_4} = (\boldsymbol{c_0} + \boldsymbol{c_1} + \boldsymbol{c_2} + \boldsymbol{c_3})/4 \tag{5.2}$$

and subdivides the face into four triangles $(\boldsymbol{c_0 c_1 c_4})$, $(\boldsymbol{c_1 c_2 c_4})$, $(\boldsymbol{c_2 c_3 c_4})$, $(\boldsymbol{c_3 c_0 c_4})$, which are consistently oriented (see Figure 5.2, left). To avoid flipping triangle orientations, we assume *convex* quadrangles, which are common in CFD datasets (see [Gar90]).

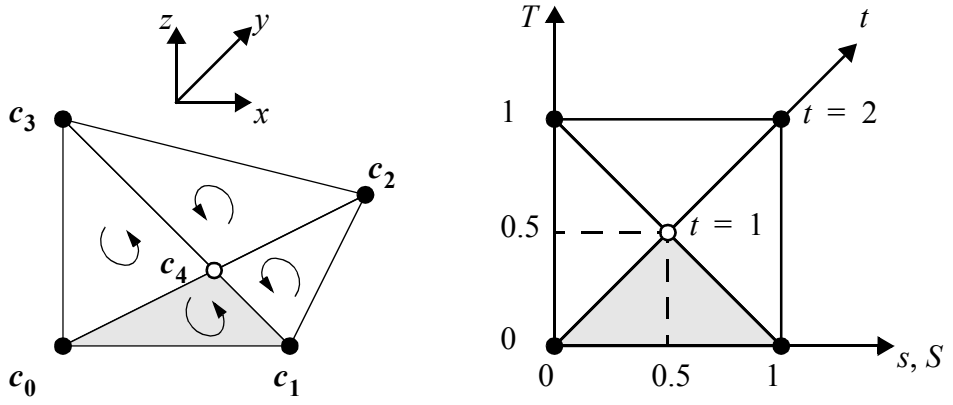


FIGURE 5.2 Subdivision of a cell face and conversion of local coordinates.

This triangulation allows for applying the analytic *eigenvector method* described by Roth [Rot00] to every triangle, rather than performing Newtonian iteration steps on the quadrangle face. The advantage of using triangles is that on a triangular face, the vector fields $\boldsymbol{v}$ and $\boldsymbol{w}$ can be *linearly* interpolated at any location (s, t):

$$\boldsymbol{v}(s,t) = \boldsymbol{v_0} + s(\boldsymbol{v_1} - \boldsymbol{v_0}) + t(\boldsymbol{v_2} - \boldsymbol{v_0}) = V \begin{bmatrix} 1 \\ s \\ t \end{bmatrix}. \tag{5.3}$$

$$\boldsymbol{w}(s,t) = \boldsymbol{w_0} + s(\boldsymbol{w_1} - \boldsymbol{w_0}) + t(\boldsymbol{w_2} - \boldsymbol{w_0}) = W \begin{bmatrix} 1 \\ s \\ t \end{bmatrix}, \tag{5.4}$$

where $0 \le s, t \le 1$ and $s + t \le 1$ holds for the local coordinates (s, t) of any point inside the triangle, $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{w_0}, \boldsymbol{w_1}, \boldsymbol{w_2}$ are the 3D vector data given at the three triangle corners (see Figure 5.3), and

$$V = \left[\boldsymbol{v_0}\ (\boldsymbol{v_1} - \boldsymbol{v_0})\ (\boldsymbol{v_2} - \boldsymbol{v_0})\right] = \begin{bmatrix} v_{0x}\ (v_{1x} - v_{0x})\ (v_{2x} - v_{0x}) \\ v_{0y}\ (v_{1y} - v_{0y})\ (v_{2y} - v_{0y}) \\ v_{0z}\ (v_{1z} - v_{0z})\ (v_{2z} - v_{0z}) \end{bmatrix} \tag{5.5}$$

$$W = \left[\boldsymbol{w_0}\ (\boldsymbol{w_1} - \boldsymbol{w_0})\ (\boldsymbol{w_2} - \boldsymbol{w_0})\right] = \begin{bmatrix} w_{0x}\ (w_{1x} - w_{0x})\ (w_{2x} - w_{0x}) \\ w_{0y}\ (w_{1y} - w_{0y})\ (w_{2y} - w_{0y}) \\ w_{0z}\ (w_{1z} - w_{0z})\ (w_{2z} - w_{0z}) \end{bmatrix} \tag{5.6}$$

are both 3×3 matrices containing differences of the data vectors at the corners.

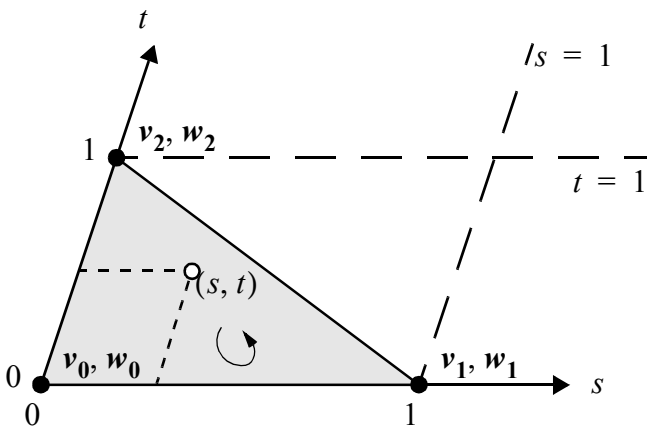


FIGURE 5.3 Linear interpolation of the vector fields $\boldsymbol{v}$ and $\boldsymbol{w}$ on a triangle.

The two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are parallel at the locations (s, t) where

$$\boldsymbol{v}(s, t) = \lambda \cdot \boldsymbol{w}(s, t) \tag{5.7}$$

or (by inserting Equation 5.3 and Equation 5.4 into Equation 5.7)

$$V \begin{bmatrix} 1 \\ s \\ t \end{bmatrix} = \lambda \cdot W \begin{bmatrix} 1 \\ s \\ t \end{bmatrix}. \tag{5.8}$$

Assume that the matrix W is regular. Then multiplying Equation 5.8 with the inverse matrix W^{-1} yields

$$W^{-1} V \begin{bmatrix} 1 \\ s \\ t \end{bmatrix} = \lambda \cdot W^{-1} W \begin{bmatrix} 1 \\ s \\ t \end{bmatrix} = \lambda \cdot \begin{bmatrix} 1 \\ s \\ t \end{bmatrix}, \tag{5.9}$$

which is an eigenvector problem for the 3×3 matrix $W^{-1}V$. Each resulting eigenvector $(1, s, t)$ contains the local coordinates (s, t) of one "parallel point" w.r.t. the triangle, and the corresponding eigenvalue λ is the signed length ratio of the two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ at this solution point.

In case the matrix W is singular, we can swap the roles of the two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$. Then multiplying Equation 5.8 with the inverse matrix V^{-1} yields

$$V^{-1} V \begin{bmatrix} 1 \\ s \\ t \end{bmatrix} = \begin{bmatrix} 1 \\ s \\ t \end{bmatrix} = \lambda \cdot V^{-1} W \begin{bmatrix} 1 \\ s \\ t \end{bmatrix}, \tag{5.10}$$

and dividing this equation by the scalar value λ results in

$$V^{-1}W\begin{bmatrix}1\\ s\\ t\end{bmatrix} = \frac{1}{\lambda}\cdot\begin{bmatrix}1\\ s\\ t\end{bmatrix} = \lambda'\cdot\begin{bmatrix}1\\ s\\ t\end{bmatrix}, \tag{5.11}$$

where $\lambda' = 1/\lambda$ is the reciprocal value of λ. So we must in this case solve the eigenvector problem for the 3×3 matrix $V^{-1}W$ and take the reciprocal value of each resulting eigenvalue λ' to get the length ratio λ of the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$.

For numerical reasons, it is in practice favourable to invert that matrix of V and W which is "more regular", i.e. whose determinant has the greater absolute value. In any case, the method computes the eigenvalues and eigenvectors for a 3×3 matrix, yielding up to 3 intersection points per triangle and thus a maximum of 12 intersection points on the quadrangle face.

As was said above, the eigenvector method determines the local coordinates (s, t) of any solution point w.r.t. the triangle where it has been found. Of course these must be converted to local coordinates (S, T) w.r.t. the quadrangle face, e.g. the coordinate transformation for the triangle $(\boldsymbol{c_0 c_1 c_4})$ is

$$S = s \tag{5.12}$$

$$T = t/2\,. \tag{5.13}$$

(see Figure 5.2, right). Having the local coordinates (S, T) w.r.t. the quadrangle face, all vector and scalar data at the solution points can be bilinearly interpolated from the values given at the four face corners.

In comparison to the original version [RP99], we changed the procedure for connecting the solution points. When all six faces of a cell have been treated, the vertices found in this cell are in pairs connected to line segments. The ambiguous situation of more than two vertices per cell is heuristically resolved by sorting the vertices by ascending λ and pairing vertices with consecutive λ values (see Figure 5.4, where the vertices with positive λ values and those with negative λ values are grouped together). If the number of vertices per cell is odd, it is possible to neglect one vertex or to repeat the computation for that cell without trivial rejecting any cell face.

In contrast to the former implementation, we stored each intersection point in an attributed *vertex list*. The list of vertex attributes comprises

- the computed eigenvalue λ (= signed length ratio of $\boldsymbol{v}$ and $\boldsymbol{w}$),
- the physical (x, y, z) coordinates of the vertex,
- the velocity $\boldsymbol{u}$ of the flow field at the vertex position,
- the time t for which the vertex has been found,
- the indices of the geometric grid cells where the vertex has been found (two different indices since a face in general belongs to two cells),
- the vortex strength (see Section 2.3.2) and feature quality (Section 5.2.5),
- some scalar values like helicity and pressure at the vertex,
- forward and backward pointers between the two vertices of a line segment (making the vertex chains to double-linked lists for easier traversal).

The orientation of a vortex core line segment is chosen consistently with the mean velocity at its vertices: for a "good quality" vortex, the angle between the line segment and its mean velocity should not exceed 90 degrees (see Figure 5.4 and also Section 5.2.5). Otherwise the line segment is flipped. Matching segments in adjacent cells are then connected to polylines, as long as the orientation of the vortex core line is conserved.

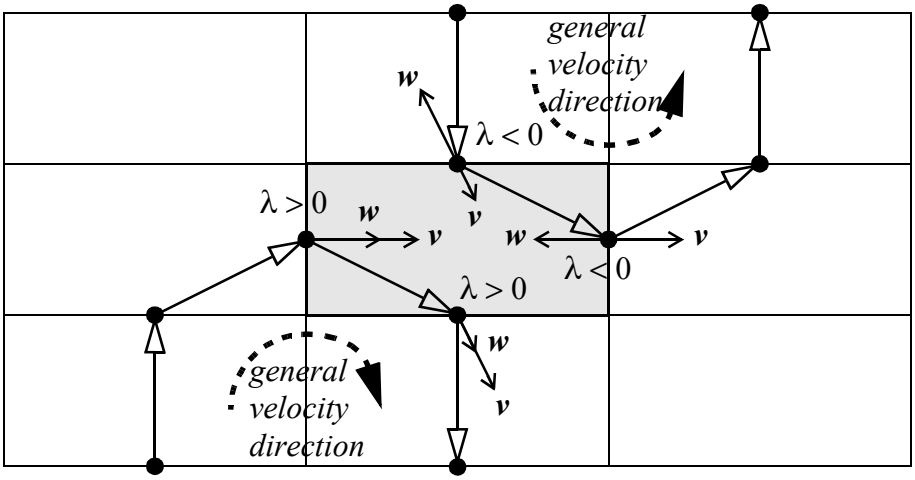


FIGURE 5.4 Composing and orienting the line segments within a grid cell (2D representation).

5.2.5 Computation of the feature quality at a vertex

The feature quality q of a vertex is defined as the absolute value of the cosine of the angle φ between the core line tangent $\boldsymbol{t}$ and the velocity $\boldsymbol{u}$ at the vertex position (see Figure 5.5). The core line tangent $\boldsymbol{t}$ at the vertex $\boldsymbol{v}_i$ can be approximated by the segment $\boldsymbol{s}$ connecting its predecessor vertex $\boldsymbol{v}_{i-1}$ and successor vertex $\boldsymbol{v}_{i+1}$ on the core line:

$$\boldsymbol{s} = \boldsymbol{v}_{i+1} - \boldsymbol{v}_{i-1} \tag{5.14}$$

$$q = |\cos\varphi| = |\cos\angle(\boldsymbol{u}, \boldsymbol{t})| \approx |\cos\angle(\boldsymbol{u}, \boldsymbol{s})| = \frac{|\boldsymbol{u} \bullet \boldsymbol{s}|}{|\boldsymbol{u}| \cdot |\boldsymbol{s}|}. \tag{5.15}$$

This approximation of the tangent is a weighted average, taking into account the different lengths of the core line segments, and thus being less vulnerable to noise at the core line. The feature quality is zero for a 90 degree angle and one for a 0 degree or 180 degree angle, thus being a measure of how well-aligned the core line direction is with the flow direction.

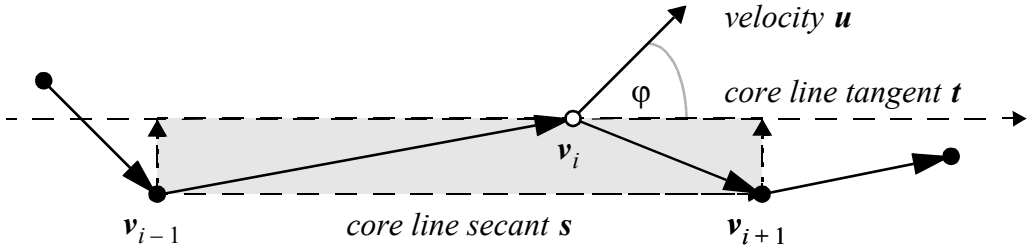


FIGURE 5.5 Definition of the feature quality of a vertex.

5.2.6 Filtering of the polylines

When viewing the resulting polylines, they should be filtered based on their stored vertex attributes. One possibility is to simply apply user-defined thresholds to the vertices, e.g. by demanding a minimal vortex strength or feature quality for every "valid" vertex. Better results can be obtained by a more sophisticated technique like *hysteresis* thresholding, allowing a certain number of exceptions between two valid vertices. An invalid vertex does not immediately terminate the vortex core line but increments a counter of consecutive invalids. The vortex core line is only truncated when the allowed number of consecutive exceptions is exceeded. If a valid vertex is found before this number is reached, the counter of invalid vertices is reset to zero and the vortex core line can be extended by further vertices (see Figure 5.6).

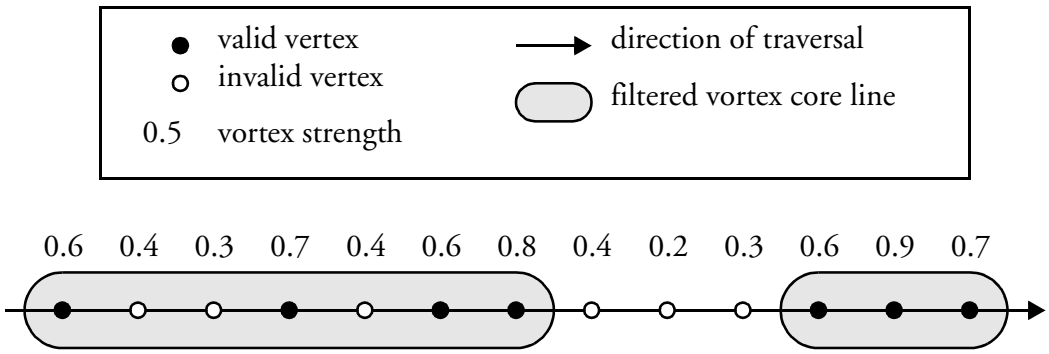


FIGURE 5.6 Filtering the polylines according to their vertex attributes (in this case, minimum vortex strength = 0.5, maximum number of exceptions = 2).

5.3 FEATURE TRACKING IN TIME AND SCALE

For various reasons, we need a way to *track* features from one timeframe to another. The obvious application is to track features along the time axis in time-dependent data. But it also makes sense to track features along the scale axis, this way exploring the information contained in the scale-space. An example for this is to interactively select an individual feature at a larger scale and then display it at a smaller scale, where its positions are more accurate but the feature might be broken into disjoint fragments. In both cases, we have the spatial dimensions plus an extra dimension, which we call the *tracking* dimension.

The two types of tracking can also be combined, as in the "user scenario" shown in Figure 5.7. Here, a feature selected at scale s_1 is tracked over scale until e.g. it breaks into fragments between the scales s_2 and s_3. The last scale s_2 where the feature was contiguous can be used as a starting point for a tracking over time rather than scale. At every time step, it is still possible to visualise the current feature in more detail at finer scales.

We will in the remainder of this chapter describe a 4D algorithm for *algorithmic* tracking of line-type features (e.g. vortices). Due to its complexity, the results take computational times between 10 seconds and 1 minute when tracked over several scales or times. For interactive exploration of features over time or scale but without explicit feature tracking, we developed a virtual reality application, which will later be described in Chapter 8.

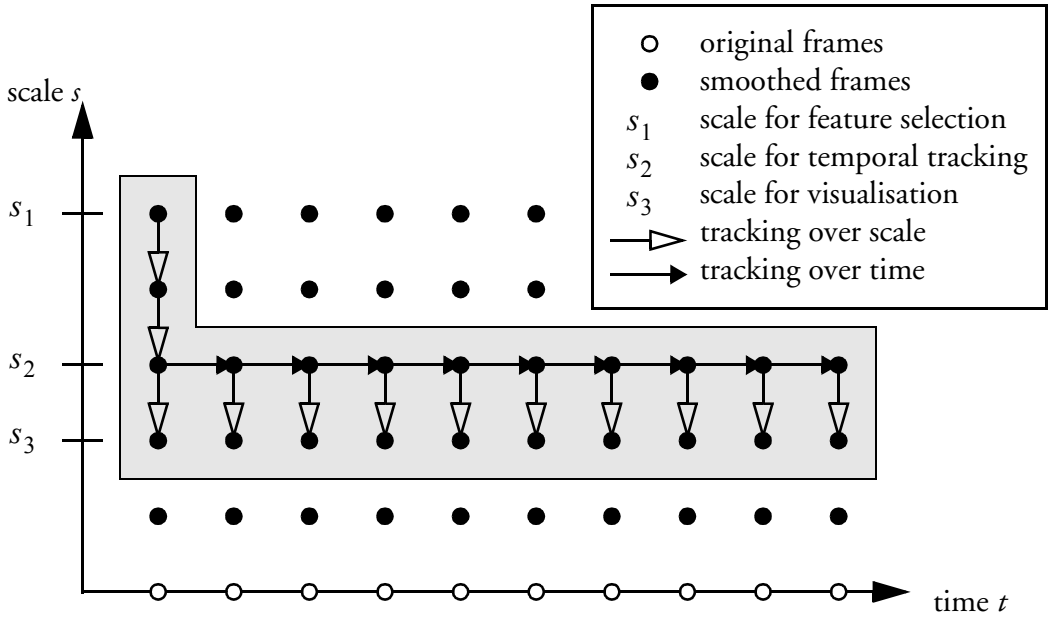


FIGURE 5.7 Example application of feature tracking in scale and time.

5.3.1 Lifting principle and hypercubes

Most of the existing tracking methods ([SZF+91, SW96], [RPS99, Rei01]) *a posteriori* try to relate features which have already been extracted from single timeframes. This approach can be compared to reconstructing an isosurface from given contour plots in a pile of slices. Before the *marching cubes* algorithm [LC87] was known, isosurface extraction had indeed been approached as a tracking problem with z being the tracking dimension.

The improvement brought by the MC algorithm was to "lift" already the contour extraction from 2 to 3 dimensions rather than lifting the results from 1 to 2 dimensions. Instead of working with 2D grid cells, the grid is extended by the tracking dimension and the contour extraction is done for 3D cells. The tracking therefore takes place *during* the feature extraction process. Such a lifting technique can basically be applied whenever a feature extraction is operating on a cell-by-cell basis. An example is the tracking of critical points by Tricoche et al. [TSH01]. In their paper, the tracking dimension is time, but it can be treated just like the z dimension in the marching cubes example. For a simpler notation, we will in the following use the term "time" for the tracking dimension, although the tracking can alternatively be performed over the scale dimension of the scale-space.

Of course, the features extracted from a lifted grid get an additional dimension, too. For example, 0-dimensional features such as critical points become 1-dimensional lines. The tracking is now simply achieved by taking slices of constant time. In an implementation, it is of course more appropriate to represent time as a vertex attribute rather than as an additional coordinate. Then, taking a slice of constant time t means to extract a *level set* for time t.

5.3.2 Structure of the lifted grid cells

Since we focus on vortex cores (or more generally on line-type features) in hexahedral grids, the lifting generates cells which topologically are 4D hypercubes. A *hypercube*, or tesseract, has 16 vertices, 32 edges, 24 faces and 8 boundary cubes. The detected features are also lifted, namely from 1D to 2D manifolds (see [BP02]). It is clear that degeneracies can cause feature dimensions to be other than expected. Implementations have to take this into account. However, this is not a problem of the lifting scheme, since the problem is already present in the underlying extraction method.

The 16 corners of a hypercube consist of the eight corners of a geometric grid cell at two consecutive times t_0 and t_1. The eight 3D boundary cubes of the 4D hypercube are depicted in Figure 5.8. Two of them are *purely spatial*, consisting of the geometric grid cell for time t_0 and for time t_1 (shown as cases a) and e)). The other six boundary cubes are *spatio-temporal*, each consisting of one geometric cell face for both times (shown as cases b) to d) and f) to h)).

The main advantage of the "lifted" vortex tracking method is that it needs no heuristics like spatial overlap [SW96] or shape attributes [RPS99] for relating the vortices across time. But it still can handle vortices moving by more than a grid cell per timeframe, the tracking being correct w.r.t. linear interpolation of time. We will demonstrate this in more detail in the next section.

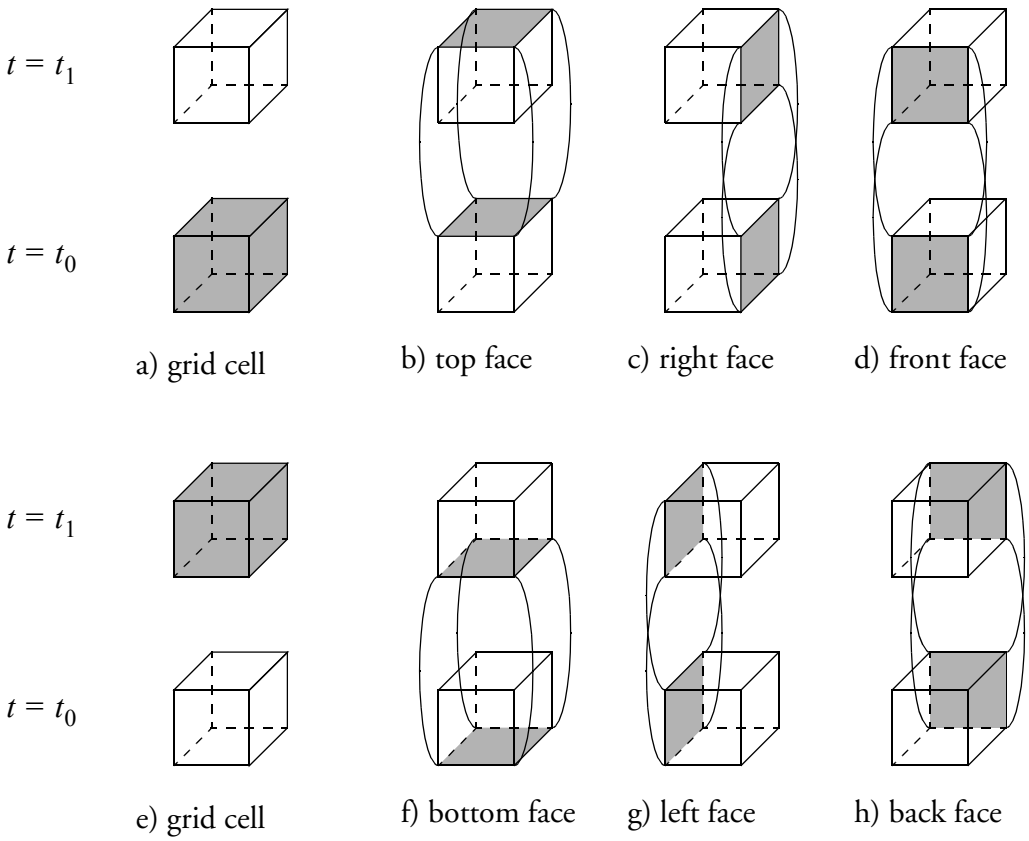


FIGURE 5.8 The eight 3D boundary cubes of a 4D hypercube (curved lines indicate the temporal edges of a spatio-temporal boundary cube).

5.3.3 Construction of the feature mesh

Let us have a look at Figure 5.9 for understanding how our algorithm tracks the vortex core lines over cell boundaries even though working on a pure cell-by-cell basis. The **left column** shows the temporal development of an example vortex core line as a film strip (Figure 5.9 a)-d)). The tracking takes place between two consecutive frames for time t_0 and time t_1. During this time span, parts of the fast-moving feature traverse more than one grid cell (e.g. the upper vertex of the core line moves from cell A to cell C, whereas the lower vertex stays within cell A).

If the vortex core line behaves due to above assumption, its upper vertex reaches the top edge shared by cells A and B at an intermediate time t'. Likewise, it reaches the top edge shared by cells B and C at an intermediate time t''. The *conventional segments* of the vortex core line for time t_0 and time t_1 (shown using black points) arise from the 3D extraction process applied to the *purely spatial* boundary cubes of the hypercubes explained in Figure 5.8. They will certainly appear in the resulting feature mesh (Figure 5.9 e)).

The **right column** of Figure 5.9 shows the computation of the *intermediate segments*. Intermediate vertices arise from the *spatio-temporal* boundary cubes of the hypercubes (see Figure 5.8) when a feature line intersects a cell edge between two consecutive timeframes. Figure 5.9 f) and g) illustrate the temporal development of the top and bottom face of cell A. The left stack marks exactly the spatio-temporal 3D boundary cube which consists of the top face of cell A for the time span between t_0 and t_1 (see Figure 5.8 b).

Applying the 3D extraction to this boundary cube results in two solution points. For time t_0, the upper vertex of the vortex core line (black point) is found. Since this upper vertex moves fast to the right and intersects the common top edge of cells A and B, it is also found on the right edge of the top face of cell A for the intermediate time t' (shown as a white point). Both solution points are projected to 3D neglecting the time which they were found for. For later times, we find no solution points since the feature has left cell A. Summarised over all times, we get an intermediate line segment (see last face of left stack, shown in dark grey). This line segment is added to the feature mesh.

The case of the *bottom* face of cell A (right stack) is different from the top face in that the lower vertex of the vortex core line moves more slowly and therefore does not leave cell A, nor does it intersect a cell edge. The 3D extraction thus yields only two solution points for this face, namely the two conventional vertices for times t_0 and t_1.

Figure 5.9 h) and i) illustrate the temporal development of the left and right face of cell B. For times before t', we find no solution point on the left face of cell B since the feature completely lies within cell A. Beginning with time t', the *intermediate vertex* from the top edge of the left face (also found for the top face of cell A) moves downward. It reaches its lowest position for time t_1, then being a *conventional vertex* on the vortex core line. The *right* face shows the same development somewhat retarded. Figure 5.9 j) and k) finally depict the top faces of cells B and C. The upper vertex of the feature completely traverses cell B and ends within cell C.

The projected line segments are stored in a segment list and form a closed polygon for every geometrical grid cell. These polygons are afterwards triangulated, filtered (see following Section 5.3.4) and build the resulting 2D feature mesh of the vortex tracking procedure. As we have demonstrated, the feature mesh contains all necessary topological information for relating the features over time, even when they are fast-moving. Furthermore, the mesh can also can be used for *event detection*, as we will see in Section 5.4.3.

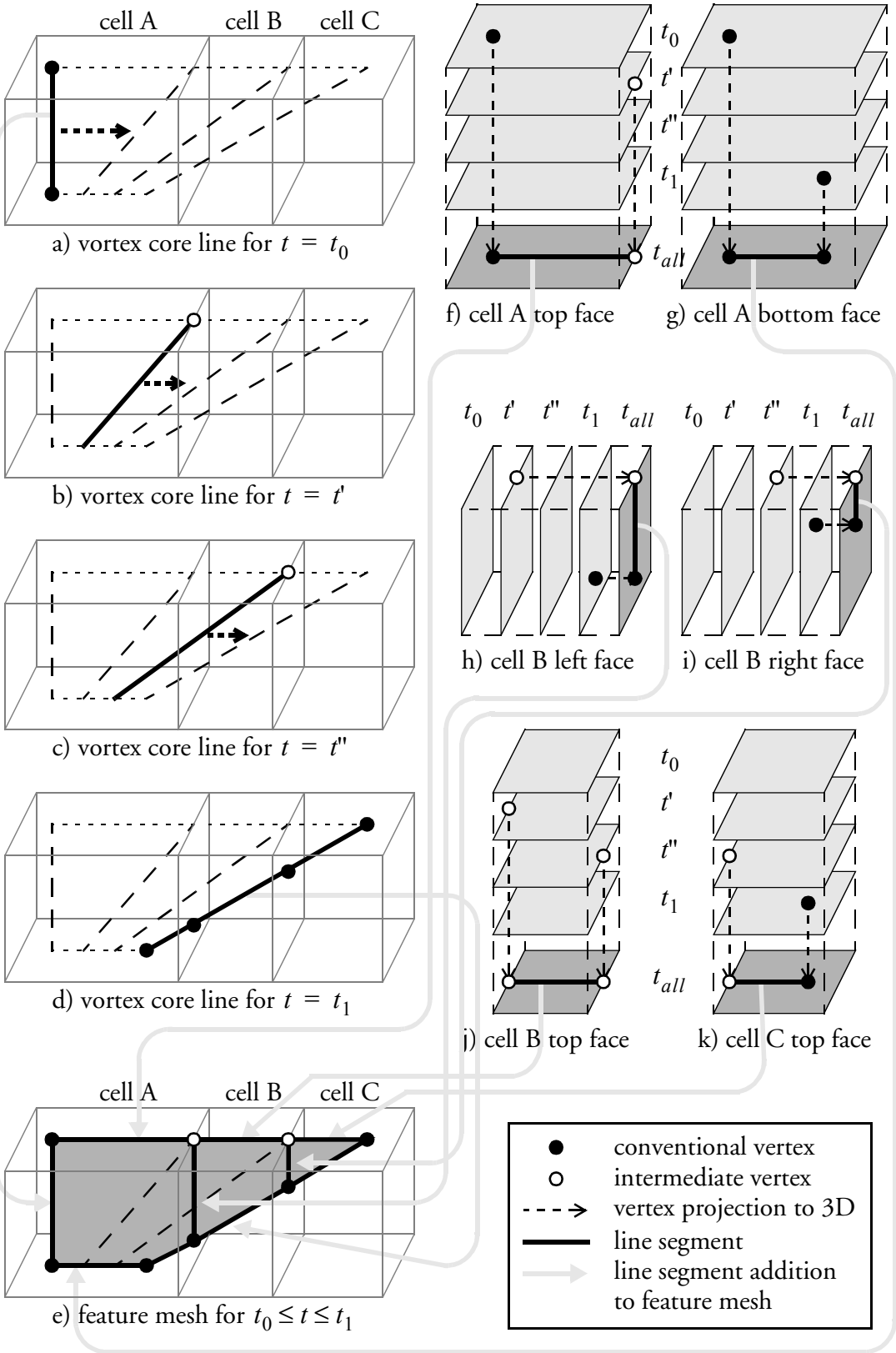


FIGURE 5.9 Construction of the 2D feature mesh from the 3D boundary cubes of the 4D hypercubes (left column: geometrical grid cells, right column: spatio-temporal boundary cubes).

5.3.4 Vortex tracking algorithm

The algorithmic problem can now be formulated as follows: Given two vector fields in 4D hyperspace (scale-space) spanned up by the physical space and the time axis:

$$\boldsymbol{v} : \mathbb{R}^4 \rightarrow \mathbb{R}^3 : (x, y, z, t) \rightarrow \boldsymbol{v}(x, y, z, t) \tag{5.16}$$

$$\boldsymbol{w} : \mathbb{R}^4 \rightarrow \mathbb{R}^3 : (x, y, z, t) \rightarrow \boldsymbol{w}(x, y, z, t) \tag{5.17}$$

and its values at the 16 corners of a hypercube, find the set of points on the 8 boundary cubes of the hypercube where the two vector fields are parallel, and output it as a triangle mesh in 3D space. The time for which a vertex was found should be stored as a vertex attribute, in addition to the other vertex attributes mentioned in Section 5.2.4.

Since all eight boundary cubes can easily be represented using the same data format as a standard grid cell, we can apply to each of them the same vortex core line extraction procedure as for the purely spatial 3D cells in Section 5.2. This leads to the algorithm shown in Figure 5.10, which loops over all eight boundary cubes for every hypercube.

```
Initialise a triangle list T.
Read the first timeframe (for time t_0).
for i = 1 to #timeframes - 1
  Read the next timeframe (for time t_i).
  for each grid cell
    Get the data values at all 16 corners of its hypercube.
    // = data values at the cell corners for times t_i-1 and t_i.
    for each of the 8 boundary cubes
      // Find line segments in the boundary cube:
      Apply the 3D procedure from Section 5.2.
      Project the line segments to 3D space
        and store the finding time as a vertex attribute.
    next boundary cube
    // The line segments form closed polygons.
    Triangulate the polygons and add the new triangles
      to the triangle list T.
  next grid cell
  Release the older timeframe (for time t_i-1).
next time step
```

FIGURE 5.10 The basic vortex tracking algorithm.

The following enhancements were necessary to upgrade the algorithm from Section 5.2:

- a file-based timeframe caching mechanism ("sliding activity window") which loads and releases the flow fields needed for the current two time steps,
- an extra loop over all timeframes / time steps where tracking shall take place,
- doubled vector and scalar fields (for treating two consecutive frames at a time),
- an extra loop over the 8 boundary cubes of a hypercube,
- lookup tables for assigning hypercube structures to geometrical cell structures (for all 16 hypercube corners: their time step and local node index in the grid cell),

- a triangle list, a filtering procedure for the triangles, and an additional output port for the triangle mesh to the graphics renderer, for depicting the 2D feature mesh resulting from the vortex tracking algorithm (see Figure 5.9).

Concerning the filtering of the triangle mesh, the minimum is to apply thresholds for the vertex attributes. Optional steps comprise the computation of connected components and/or the extraction of level sets of time.

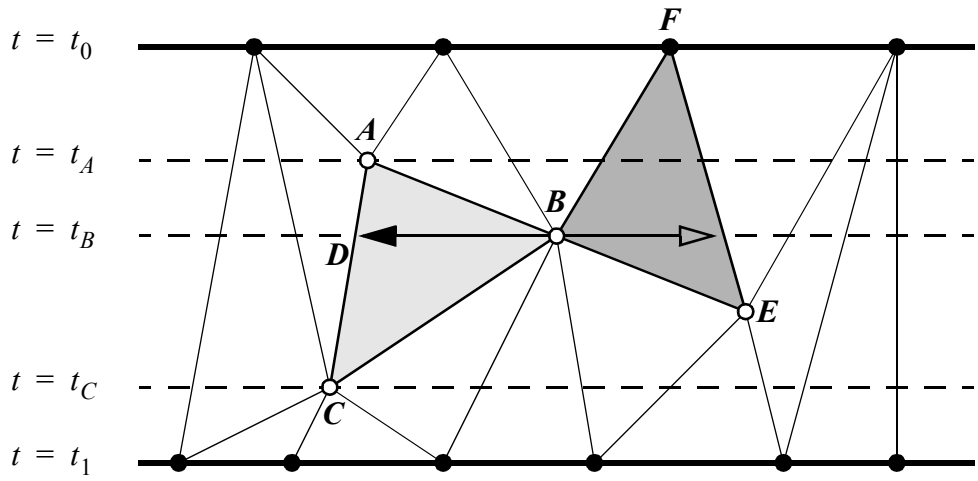


FIGURE 5.11 Filtering the triangle mesh by computing the feature quality of intermediate vertices.

One possibility for filtering a triangle is the computation of its feature quality. Section 5.2.5 explained how the feature quality for a vertex on a core line is computed (that means a vertex found for either time t_0 or time t_1, where frames are available). This definition must now be enhanced to vertices found for *intermediate* times ($t_0 < t < t_1$).

As for conventional vertices, the quality is defined as the cosine of the angle between a velocity vector and a direction vector. Since an intermediate vertex lies on a triangle *between* two core lines (see triangle (ABC) in Figure 5.11), we cannot use the core line segments for computing the "core line tangent" as described in Section 5.2.5. We therefore sort the vertices (A, B, C) of the triangle by ascending times so that $t_A < t_B < t_C$.

For vertex B with intermediate time t_B, we compute the *direction* vector $\overrightarrow{BD}$ which is the projection of the vertex B along the time isoline $t = t_B$ to the opposite triangle edge (AC). This direction replaces the "core line tangent" from Figure 5.5. The *velocity* vector is computed by averaging the velocity at the vertex B and at its counterpart D (linearly interpolated from the velocities at the vertices A and C). This vector replaces the velocity of a conventional vertex from Figure 5.5. Using the new direction and velocity definition, we can evaluate the cosine formula for computing the feature quality of vertex B.

Since the vertex B is the intermediate time vertex for *two* triangles, its quality must be computed for the triangle (BEF) lying in the opposite direction, too. If the cosines for both triangles have the same sign (apart from their opposite directions), the vertex B is marked as "valid", else as "invalid". At the end, a triangle of the feature mesh is marked as valid if all its three vertices are valid, otherwise the triangle is marked as invalid.

5.4 POSSIBLE EXTENSIONS

We will in the following discuss some possible extensions of our feature extraction and tracking methods, which were not implemented but might be interesting for future work.

5.4.1 Different cell types

So far, the 3D algorithm for feature extraction and the 4D algorithm for feature tracking assume that the underlying grid consists of only one type of grid cells, namely hexahedral ones. However, unstructured grids can contain a variety and even a mixture of different cell types. Besides of *hexahedral*, also *tetrahedral*, *pyramidal* or *prism* cells are common. The following Table 5.1 shows the properties of these cell types which are relevant for our feature extraction and tracking method:

cell type	number of 3D cell corners *(c)*	number of 2D cell faces *(f)*	type of faces	number of 4D hyper-corners *(2c)*	number of 3D boundary elements *(f+2)*	purely spatial boundary elements	spatio-temporal boundary elements
tetra-hedron	4	4	4 triangles	8	6	2 tetrahedra	4 prisms
pyramid	5	5	4 triangles 1 quadrangle	10	7	2 pyramids	4 prisms 1 hexahedron
prism	6	5	2 triangles 3 quadrangles	12	7	2 prisms	2 prisms 3 hexahedra
hexa-hedron	8	6	6 quadrangles	16	8	2 hexahedra	6 hexahedra

TABLE 5.1 Properties of different cell types and extension from 3 to 4 dimensions.

Since the faces of all these cell types are triangles or quadrangles, our 3D feature extraction method already contains the required methods to find solution points on the faces. As described in Section 5.2.4, we treat quadrangles by subdividing them into four triangles and applying the eigenvector method to the linearised vector fields on the triangles. For triangular faces as occur in tetrahedra, we can omit the subdivision and conversion of local coordinates. If we extract features for a pyramid or prism, we must for every face determine whether it is a triangle or a quadrangle.

Concerning the 4D feature tracking method, we similarly must for every cell regard the types and numbers of its boundary elements. As usual, we loop over all boundary elements of the current cell and apply the 3D feature extraction procedure. In summary, we only have to make a few slight changes to our algorithm, mainly in some simple additional conditions (*if...then...else* or *switch* statements), which do not consume considerable computational time.

5.4.2 Different feature definitions

Our main application in this thesis was the extraction of vortex core lines. However, our method is not restricted to these. In principle, we can apply it to any feature which fulfills the following requirements:

- the feature must be 1-dimensional (*line-type* feature)
- it can be calculated "face-wise", that means we can construct it from solution points found on the grid faces, which can later be connected to line segments
- we can calculate it on a cell-by-cell basis, that means all cells can be treated independently. As a consequence, the algorithm is suitable for *parallelisation*.
- the feature must be *locally* defined, that means for a certain grid cell, we do not need any information from its neighbouring cells or cells that are even more distant.

Besides of vortex core lines, also *separation* and *attachment* lines [Ken98], and also *ridge* and *valley* lines fulfill these requirements, since all of these can be reduced to the parallel vectors operator [RP99, Rot00]. However, integration-based features such as streamlines cannot be treated by our algorithm. Although they are line-type, they are *globally* defined over the whole domain, so they must be integrated from a certain startpoint through all the grid. For a calculation of a streamline within a cell, we first need the entry point, which depends on the calculation in the cells which the streamline has previously intersected. This sequential process would not allow for treating the cells independently, and also not for parallelisation.

All said above holds for the 3D feature extraction and for the 4D feature tracking likewise, since the latter one is based upon the former one.

5.4.3 Event detection

For this thesis, only the fundamental tracking of features was implemented, which comprises the development of *continuing* features without bifurcations. However, a feature can appear or disappear in the interior of the grid, and enter or leave the grid at its boundary. Furthermore, a feature can break into several features, or several features can join each other. Such *events* have been defined and investigated by Samtaney et al. [SSC94], Silver / Wang [SW97, SW99], and Reinders et al. [RPS99, RPS01] using feature tracking methods based upon *region correspondence* and *attribute correspondence*.

We will in the following indicate how our *implicit* feature tracking in the spatio-temporal domain could be extended to allow event detection, too. Most of the information required for this purpose is already implicitly contained in our two-dimensional triangulated feature mesh. As stated in Section 5.3.3 and Section 5.3.4, the triangles of this mesh are spanned by conventional vertices found for times where time steps are available from the CFD dataset, and intermediate vertices found for times between two time steps.

As Figure 5.11 depicts, we can determine the line-type features for a given intermediate time by computing isolines for this constant time from the feature mesh. Let us investigate how the different event types mentioned above can influence the shape of the feature mesh.

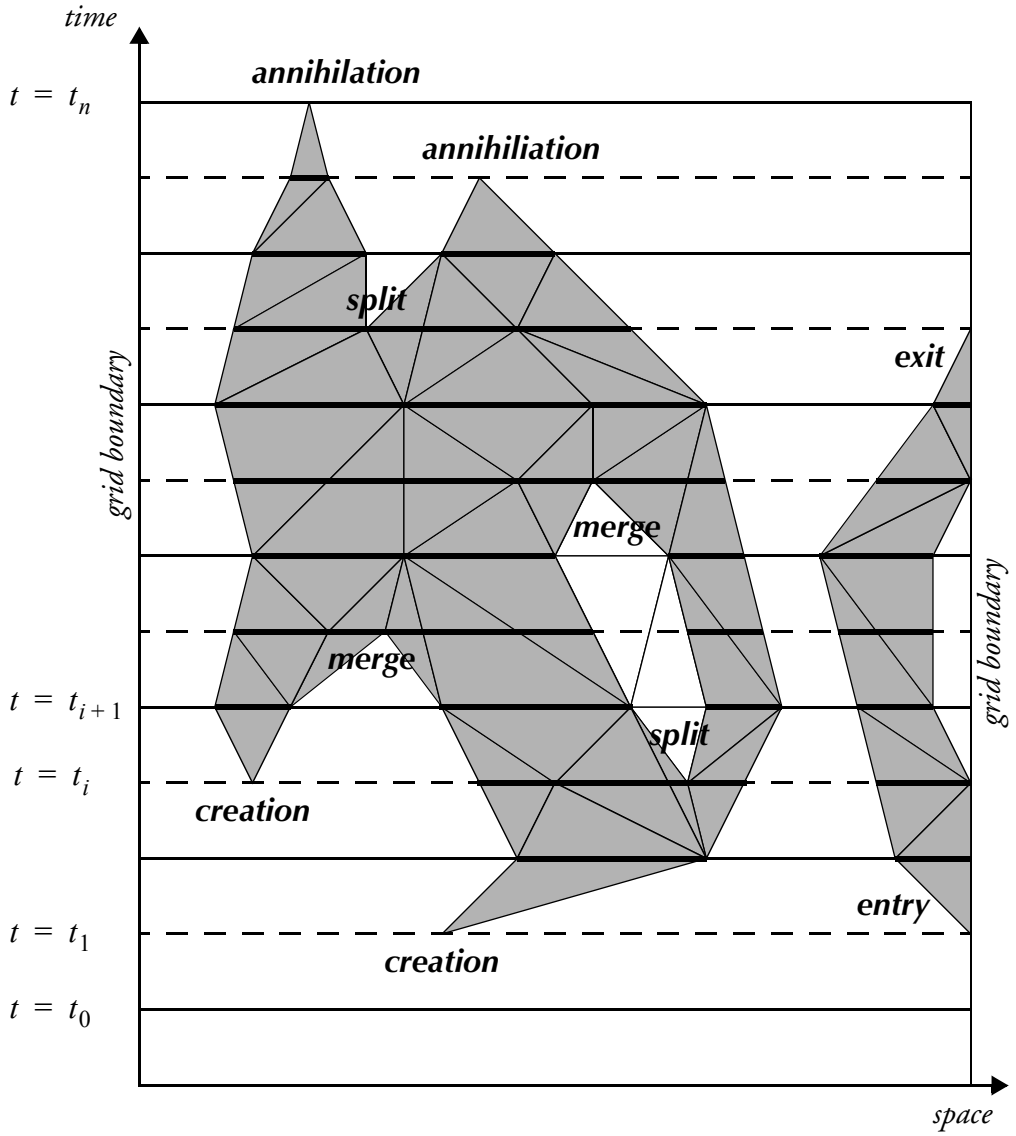


FIGURE 5.12 Event detection using the feature mesh.

- A *split* event will also split the feature mesh. If two split features later *merge*, a hole will occur in the mesh (see Figure 5.12). At the boundary of the mesh, splits and merges will show up as additional branches in the mesh topology. All in all, the topology of the mesh is significantly changed.
- If a feature appears, this *creation* event will open a new branch in a region of the mesh where no preceding triangles have been before, in contrast to a split event. Likewise, if a feature disappears, this *annihilation* event will terminate a branch of the mesh with no subsequent triangles in that region, in contrast to a merge event.
- An *entry* event is similar to a creation event in that a new branch occurs. The difference is, however, that some vertices of the corresponding mesh triangles were found on faces at the boundary rather than in the interior of the grid. As stated at the end of

Section 5.2.4, we store in the vertex list not only the time where the vertex was found, but also the global cell indices of the cells sharing the face where the vertex was found. A face at the grid boundary is characterised by having only one valid cell index (the second cell index is then set to "void"). Thus we can easily distinguish entry events from creation events. Similarly, we can define an *exit* event where a feature leaves the grid, and separate exit events from annihilation events.

Taking into account the considerations above, it makes sense to sort the triangles of the feature mesh by increasing minimum times of their vertices. For a complete event detection, we must loop over all times where (conventional or intermediate) vertices were found, so setting up a time list containing the times of all mesh vertices is also helpful.

The event detection can now take place similarly to the algorithm for rasterising a polygon known from computer graphics. We loop over all times of the timelist and extract for each time its time isoline by intersecting it with all *active triangles* of the triangle mesh. Trivial rejection of *inactive triangles* can be applied by comparing the current time with the minimum and maximum vertex time for each triangle. Active edges and triangles are exactly those whose vertices have times greater and smaller than the current time.

For each intersected triangle, we store the two intersection points as line segments in a segment list, as well as whether the intersected triangle is "valid" (see end of Section 5.3.4). When the feature lines have been extracted for all time steps, we make a second loop over all time steps, but this time we regard two successive time steps at once (just as we did when computing the feature mesh). If a triangle is "invalid", we have detected a hole in our feature mesh, which indicates an event, and the current feature line ends. If we find a new intersected and "valid" triangle, a new feature line begins. This way, we can determine the number and location of all feature lines for the current time.

Table 5.2 summarises the different event types, along with the circumstances which indicate their occurence.

event type	intersection of triangle with time isoline		triangle type	triangle flag	adjacent triangles
	$t = t_i$	$t = t_{i+1}$			
split	vertex	segment	interior	invalid	yes, both valid
merge	segment	vertex			
creation	vertex	segment	interior	valid	none
annihilation	segment	vertex			
entry	vertex	segment	boundary	valid	none
exit	segment	vertex			

TABLE 5.2 Mapping of triangle intersection types to event types.

5.5 RESULTS

In this section, we provide numerical and visual results based on two time-dependent industrial datasets. Both datasets are discretised on unstructured hexahedral grids.

The first dataset is a coupled unsteady simulation (by VA Tech Hydro) of the runner and draft tube of a Francis turbine. The main purpose of the simulation was to predict the *vortex rope*, a helical vortex structure having a precession rate of roughly one third of the runner frequency. The computation was done for the original and for an optimised runner geometry. The optimisation improved the flow behavior, but its less articulate vortex rope is more difficult to extract. Figure 5.14 shows the vortex rope in the original design. The vortex core was extracted based on the definitions by Miura and Kida [MK97]] and by Levy et al. [LDS90]. Because the former works with second derivatives, some smoothing was necessary. The latter produced acceptable results even without smoothing.

Regarding the redesigned draft tube (Figure 5.15), it is remarkable how the topology of the vortex cores depends on the scale. At the smallest scales, noise is dominating, which disappears at medium scales. At larger scales, a vortex core close to the machine axis appears, describing the global swirl in the draft tube. Tracking the vortices over different scales and displaying them by triangular feature meshes allows us to recognise the fragmented cores without sacrificing the positional accuracy (Figure 5.16 left).

The second dataset is an unsteady simulation (by Sulzer Pumpen) of a mixed-flow pump operated at 35% of its best efficiency point. The vortex extraction and tracking on a 5-processor SGI Power Challenge for 1800 time steps (1-degree increments) took 73.5 CPU-hours, thus about 2 1/2 CPU-minutes per time step. The temporal tracking of the vortex cores generates a description of the vortex motion. Figure 5.16 (right) reveals that most vortices are quite stable, whereas one vortex per diffusor channel is oscillating.

Figure 5.13 demonstrates the feature simplification aspect of the scale-space approach. Vortices have been extracted from all three datasets at fixed times but different scales. The number of features is plotted against the standard deviation of the Gaussian smoothing kernel. As is apparent, the number of features decreases significantly with increasing scale.

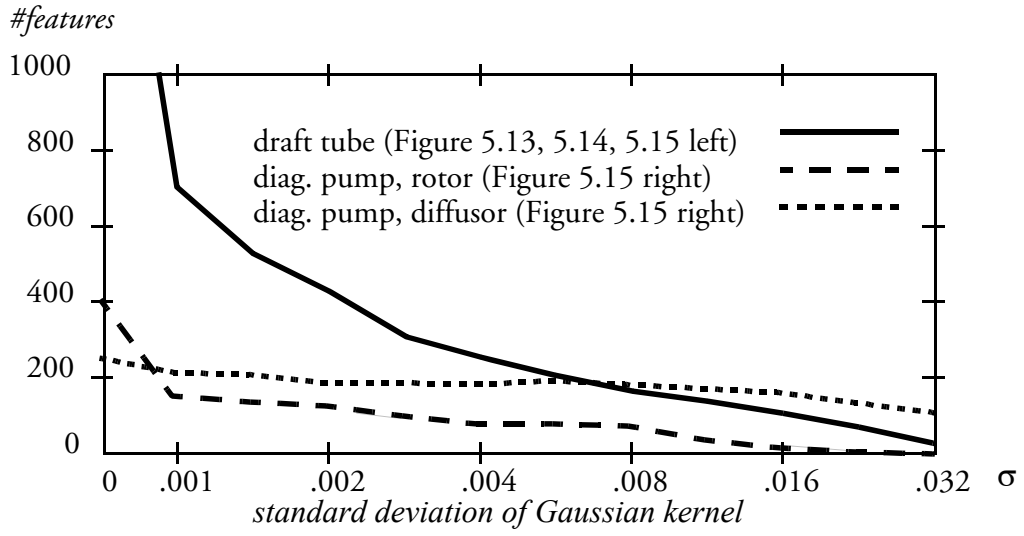


FIGURE 5.13 Number of features at different scales.

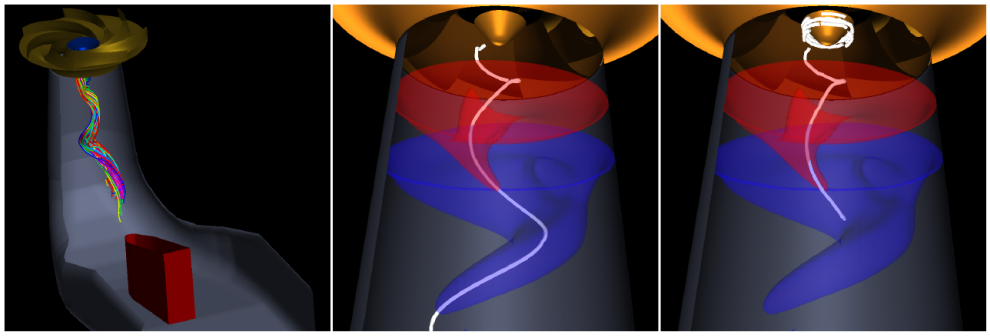


FIGURE 5.14 Original draft tube dataset: geometry and instantaneous streamlines (left). Pressure isosurfaces and pressure valley line indicate the so-called vortex rope. Pressure data for Miura/Kida method have been Gaussian-smoothed with $\sigma = 0.008$ (middle). Vortex core extraction based on normalised helicity (Levy method) can be successfully applied to unsmoothed data (right). See also Colour Figure A.1 on page 121.

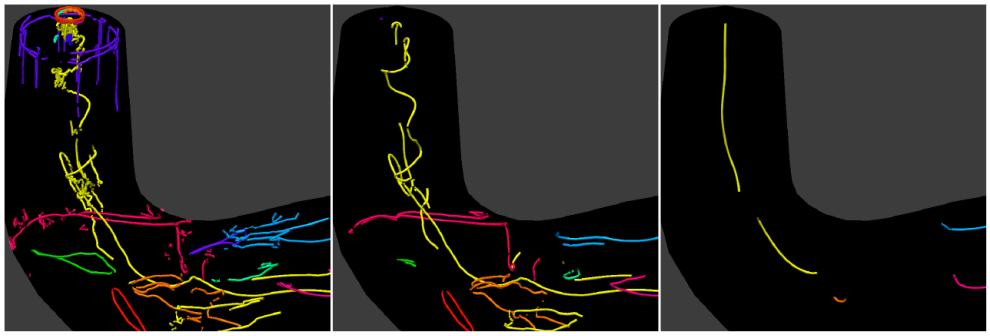


FIGURE 5.15 Modernised draft tube: vortices are less articulate and harder to extract. Vortex core extraction based on normalised helicity (Levy) has been applied to unsmoothed velocity field (left) and to Gaussian-smoothed data with $\sigma = 0.008$ (middle) and $\sigma = 0.032$ (right). Colors represent connected components of the surface swept by the core lines, see Figure 5.16. See also Colour Figure A.2 on page 122.

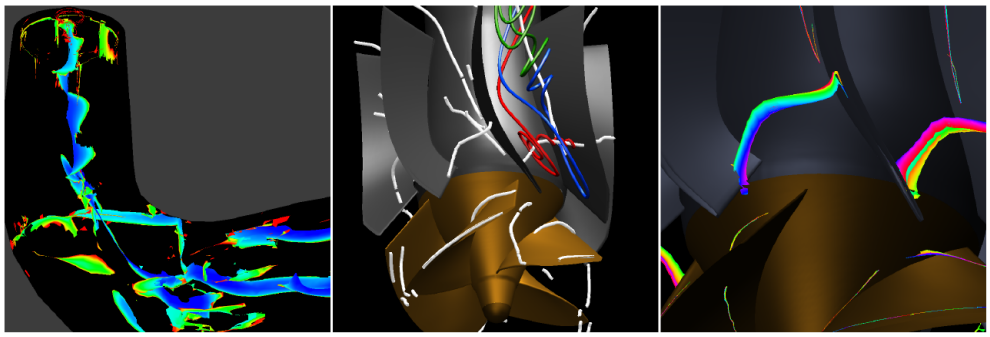


FIGURE 5.16 Vortex cores of modernised draft tube tracked through scales $\sigma \in [0, 0.032]$ (left). Diagonal pump dataset with vortices extracted in runner and diffusor along with manually seeded streamlines (middle). Vortex cores tracked temporally for a $90°$ rotation of the four-bladed runner (right). See also Colour Figure A.2 on page 122.

CHAPTER 6

VORTEX HULLS

The aim of the vortex hull construction is to make the vortex core better visible and also interpretable. Rather than showing only a one-dimensional (line-type) feature, we now want to get a notion of the "size" of the vortex in a local neighbourhood of the core line. The goal can be achieved by surrounding the core line using a tube structure which is a closed surface, and thus combining line-type with region-type feature extraction methods. This hybrid approach combines the advantages of line-type features with those of region-type features, namely clear separability and clear visibility of the vortices.

Most of the methods in literature regard the vortex hull as a *deformable model* which is topologically equal to a cylinder. In this case, the vortex hull is initialised with the vortex core line, corresponding to a cylinder of infinitesimal diameter. The vortex hull is then expanded until a certain termination condition is met. An example for this is Sadarjoen's method which uses deformable models to approximate surface-type features [Sad99]. A general introduction to deformable models was given by Terzopoulos and Fleischer [TF88].

Concerning the termination criterion of the expansion, Banks and Singer [BS94] postulate that the pressure must not exceed a predefined limit, and that the angle between vorticity at the core line and vorticity at the current point must be smaller than 90 degrees. However, their implementation is restricted to rectilinear grids. Roth [Rot00] describes an implementation which works fine for structured grids.

In contrast, we will in this chapter present a novel algorithm for constructing vortex hulls in *unstructured* grids. The implementation allows to select from different hull shapes and scalar fields, to define suitable thresholds, and to smooth the cross-sections and overall shape of the constructed tubes. At the end of this chapter, we will give some result images as well as performance analysis data for the algorithm. Further images of vortex hulls were created using a virtual reality system which will later be presented in Chapter 8.

6.1 BASIC VORTEX HULL ALGORITHM

In an earlier paper [BP02b] and in the previous chapter, we described how to extract a vortex core line from a 3-dimensional CFD dataset. Based on such a vortex core line, our vortex hull approach follows the deformable model paradigm by radially expanding the core line as long as a certain scalar field is above or below a given threshold. To achieve this goal, the vortex hull is built from tube sections, where one tube section is added for every vertex or segment of the core line.

For every vertex, a seed point is set on the vortex core line, a plane containing the seed point and being perpendicular to the core line is set up, and a fan of n rays originating at the seed point is defined in that plane (n being an arbitrary integer greater than two). As soon as the lengths of the rays are known, the endpoints of the rays can be connected to star-shaped polygons. The cross-sections of the vortex hull along the core line are therefore n-sided polygons (see Figure 6.1).

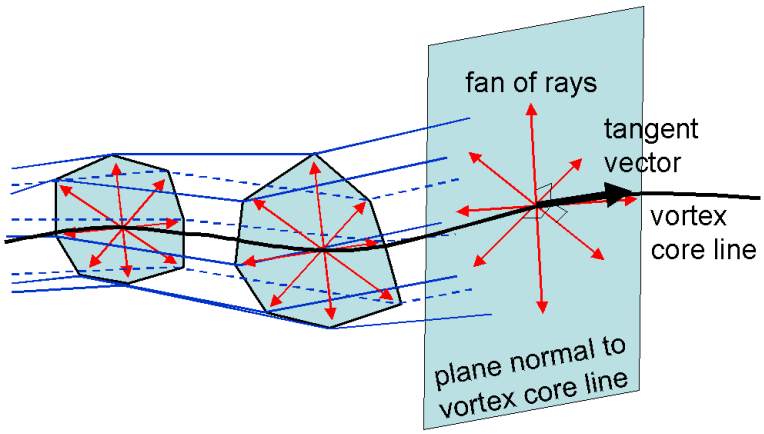


FIGURE 6.1 Principle of vortex hull construction.

The cross-section polygons can be of constant or variable shape and size, depending on what is preferred by the user. If the vortex only shall appear more plastic (e.g. for demonstration purposes), it may be sufficient to construct the tube sections as piecewise regular prisms or approximated cylinders of constant diameter, without a special physical meaning of the thickness. However, if the user is also interested in a measure for the "extent" of the vortex (e.g. for engineering and turbine design improvement), the information of a scalar field surrounding the core line will be needed for construction, and the resulting prismatic tube sections will be of varying thickness and shape, depending on the choice of scalar field and on the scalar threshold.

In a previous implementation (see [BP02a] and [SP03]), we used a fixed sampling rate for evaluating the scalar field on the rays, and vortex strength as the scalar field. Similar vortex hulls, but based on pressure values, have been computed by Banks and Singer [BS94]. In summary, our original algorithm consists of the following steps (Figure 6.2):

```
Define a seed point at every vertex on the vortex core line.
for each seed point on the vortex core line
    Estimate the core line tangent by adjacent core line segments.
    Set up a plane perpendicular to the vortex core line.
    Construct a fan of rays which spreads over the plane.
    for each ray of the fan
        Generate sample points on the ray at fixed small intervals.
        for each sample point on the ray
            Determine the cell containing the sample point.
            Evaluate the scalar field.
            if the scalar value has crossed the threshold then break
        next sample point
    Store the current sample point as endpoint of the ray.
    next ray
    Draw a star-shaped polygon connecting the ray endpoints.
next seed point
Connect the star-shaped polygons to prismatic tube sections
around the vortex core line.
```

FIGURE 6.2 The original vortex hull construction algorithm (fixed sampling intervals).

Figure 6.3 shows a vortex hull resulting from this algorithm, as well as the vortex core line and some streamlines indicating the vortical structure. With a slight modification, this algorithm produces a scalar field containing the vortex strength for nodes inside the vortex hull and zero values for nodes outside the vortex hull. In the remainder of this chapter, we will propose an improved version of this algorithm, which uses some more sophisticated techniques for the vortex hull computation and representation.

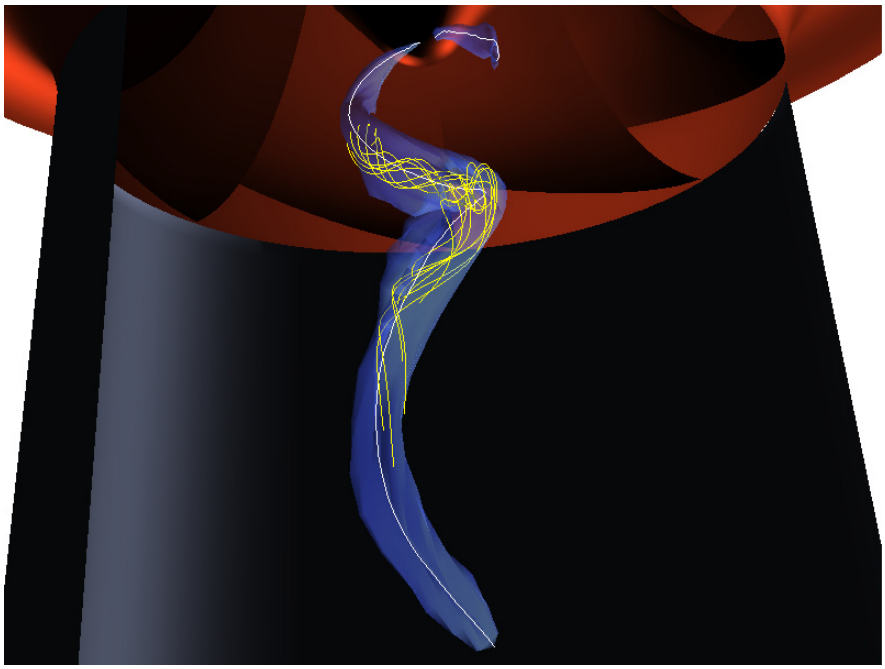


FIGURE 6.3 Vortex core line and computed vortex hull (streamlines added for better depicting the swirl in the draft tube).

6.2 ENHANCED VORTEX HULL ALGORITHM

Based on the original vortex hull algorithm, we made several modifications which help improve the quality and reduce the computational time of the vortex hulls. The first design decision (explained in more detail in Section 6.2.1) was to set up the cross-section planes (and thus the seed points of the rays) in the *middle* of the core line segments rather than at their endpoints (which are the core line vertices). This choice has two advantages:

1. The plane normal (which must be approximately tangential to the core line) needs not to be interpolated from the two adjacent core line segments of a vertex. Instead, it is directly given for each core line segment by the difference vector of its two vertices.
2. The curvature of the line connecting the seed points is reduced, which smoothes the resulting vortex hull along its central axis (see Figure 6.4). This is the same smoothing effect as occurs when applying one step of the de Casteljau algorithm to a control polygon in order to construct a Bézier curve ("corner cutting", [ESK96]).

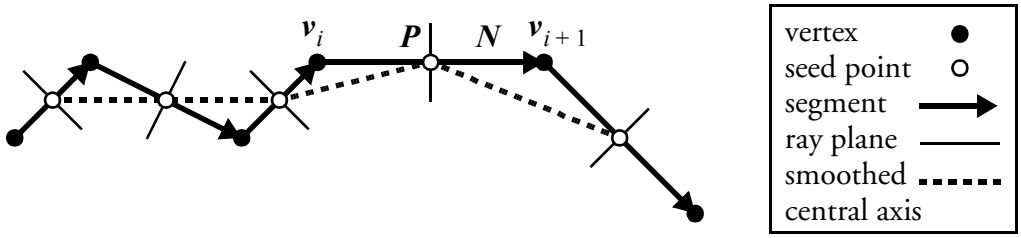


FIGURE 6.4 Location of the cross-section planes in the middle of the core line segments.

The second design decision was to use *adaptive* rather than fixed sampling intervals for traversing the rays and evaluating the scalar field. Rather than stepping forward in physical space and making point searches for the cell containing the current position, the ray traversal is done using an enhanced connectivity information of the (unstructured) grid cells. The modified procedure for the ray traversal will be discussed in Section 6.2.2.

The third design decision was to *filter* the lengths of the ray endpoints, and to *fair* the resulting mesh using a discrete approximation of the Laplacian at the mesh vertices. These postprocessing methods will later be described in Section 6.2.3 and Section 6.2.5. Our improved vortex hull algorithm basically consists of the following steps:

1. **Set up the cross-section planes** and rays:
   For every segment of the core line, choose the midpoint as a seed point,
   set up a normal plane, a local coordinate system and a fan of rays in this plane.
2. **Radially expand the vortex hull**:
   For every ray, traverse cell by cell, compute the intersection points
   with the cell faces, and check the scalar field at these points w.r.t. the threshold.
3. **Filter the resulting ray lengths** (optional):
   Apply a median filter in a k-neighbourhood of every ray to be filtered.
4. **Assembly the vortex hull** by connecting the ray endpoints using polygons.
5. **Fair the resulting mesh** (optional) by iterative Laplacian smoothing.

We will in the following treat some implementation aspects of the improved algorithm.

6.2.1 Setup of the cross-section planes

As we have seen in the previous section, a cross-section plane of the vortex hull is defined by a core line segment and its two endpoints. The plane origin $\boldsymbol{P}$ and the plane normal $\boldsymbol{N}$ can easily be computed as

$$\boldsymbol{P} = (\boldsymbol{v}_i + \boldsymbol{v}_{i+1})/2 \tag{6.1}$$

$$\boldsymbol{N} = \boldsymbol{v}_{i+1} - \boldsymbol{v}_i \tag{6.2}$$

where $\boldsymbol{v}_i$ and $\boldsymbol{v}_{i+1}$ are two successive vertices on the core line which form the endpoints of the core line segment (see Figure 6.4). Within such a plane, a two-dimensional coordinate system can be installed by setting up two basis vectors $\boldsymbol{L}$ and $\boldsymbol{M}$, which must be perpendicular to $\boldsymbol{N}$ and to each other. By taking suitable cross products and normalising $\boldsymbol{L}$, $\boldsymbol{M}$, $\boldsymbol{N}$, the three vectors build a right-handed coordinate system (see Figure 6.5, left).

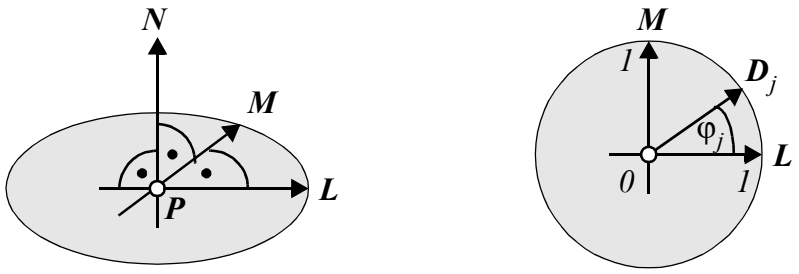


FIGURE 6.5 Plane coordinate system for the rays, and definition of its basis vectors (3D and 2D representation, respectively).

We now have a coordinate system in a plane perpendicular to the core line. To construct the n-sided polygon for the cross-section of the vortex hull, we build a fan of rays originating at the seed point $\boldsymbol{P}$ and pointing away from $\boldsymbol{P}$. The directions of the rays are equally distributed over the plane by dividing 360 degrees by the number n of desired rays (which is a user-defined constant for all vortex hulls). If the rays are numbered in ascending order from 1 to n, then the angle φ of the ray j w.r.t. the basis vector $\boldsymbol{L}$ is

$$\varphi_j = \frac{j}{n} \cdot 360° \qquad (j \in \mathbb{N},\ 1 \le j \le n) \tag{6.3}$$

(see Figure 6.5, right), and the direction $\boldsymbol{D}_j$ of the ray j in 3-space is

$$\boldsymbol{D}_j = \boldsymbol{L} \cdot \cos(\varphi_j) + \boldsymbol{M} \cdot \sin(\varphi_j) \tag{6.4}$$

Given the seed point $\boldsymbol{P}$ and the ray direction $\boldsymbol{D}_j$, we can write the equation of the ray j as

$$\boldsymbol{x}_j(\lambda) = \boldsymbol{P} + \lambda \cdot \boldsymbol{D}_j \qquad (\lambda \in \mathbb{R},\ \lambda \ge 0) \quad . \tag{6.5}$$

This equation serves as a mathematical basis for the ray traversal. The next section will treat the question how to find a suitable λ value (and thus the endpoint of a ray) so that a user-defined termination criterion for the vortex hull is met.

6.2.2 Radial vortex hull expansion

Once a certain cross-section-plane of the vortex hull has been established, the further proceeding of the algorithm depends on the users's choice. If only a better visualisation of the progression of the vortex core line is desired, it might be sufficient to give the vortex hull a constant thickness, i.e. the λ values of all ray endpoints are equal (and can be set to a user-defined value). This will lead to regular shaped prismatic tubes, and, in case of a high value for the number n of rays per fan, to a quasi-cylindric shape (see Figure 6.6, left).

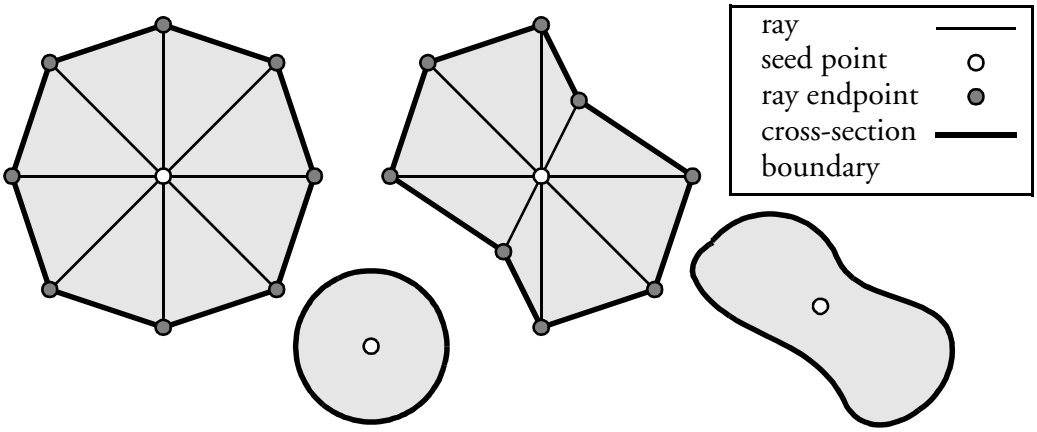


FIGURE 6.6 Vortex hull cross-sections based on different scalar fields.
Left: based on distance field of core line, right: based on independent scalar field (threshold searching). Shapes are drawn for n=8 and a large value of n, respectively.

However, if the user is interested in seeing a measure for the "importance" of the vortex in the environment of the core line, a physically meaningful scalar field surrounding the core line is needed, as well as a user-defined scalar threshold. The endpoint of a certain ray is now computed as the first point on the ray where the scalar field crosses the scalar threshold. This will result in different ray lengths and non-regular shapes of the vortex hull cross-sections (see Figure 6.6, right). To achieve smooth vortex hulls, we could *subdivide* the resulting mesh of ray endpoints (e.g. by the *Catmull-Clark* operator [CC78]). However, since the core line vertices lie on neighbouring cell faces, the ray planes are relatively close together. And since we can also increase the number of rays, we decided to rather *fair* the resulting mesh using Laplacian smoothing without subdivision (see Section 6.2.5).

Possible choices for the scalar field are the pressure (which is often directly given on one of the data channels of industrial datasets), or the helicity (which can easily be computed from the velocity data of the flow). A good choice is also to use *vortex strength* as the scalar field, which is based on the velocity gradient tensor. Of course, the choice of constant thickness for the cross-sections can also be regarded as using a scalar field, namely the *distance* field of the core line. In this case, the scalar threshold is equal to the length of the rays (or, in the case of many rays, equal to the radius of the resulting nearby cylindric tube).

The actual algorithm to find the endpoint of a certain ray is based on a *region-growing* scheme. The idea is to traverse the ray away from the seed point, which has a parameter value of $\lambda = 0$ w.r.t. the ray equation (Equation 6.5). Our previously implemented approach stepped along the ray in fixed intervals in physical space, thus requiring a point search for every sample point to locate the cell containing it.

The disadvantage of this method is, of course, the trade-off between many small interval steps (consuming much computation time for the point searches) and few big interval steps (taking the risk of missing the first point where the scalar value crosses the threshold). Since the frequencies of the scalar field are not known in advance, it is not trivial to adequately choose the interval step size due to Shannon's *sampling theorem* [Sha49].

To circumvent this problem, we enhanced the connectivity structure of the unstructured grid. The new data structure stores for every face of the grid the two cells sharing the face, and for every cell of the grid its six side faces. This allows us to "switch" from one cell to a neighbouring cell without performing any point search. In combination with the ray direction, this means that it is sufficient to traverse cell by cell, test the cell faces for intersections with the ray, and to switch to the cell which follows after the "next" point on the ray (see Figure 6.7). In other words, it is in every step necessary to select the intersection point with the smallest λ value greater than the current λ value (see also [Gar90]).

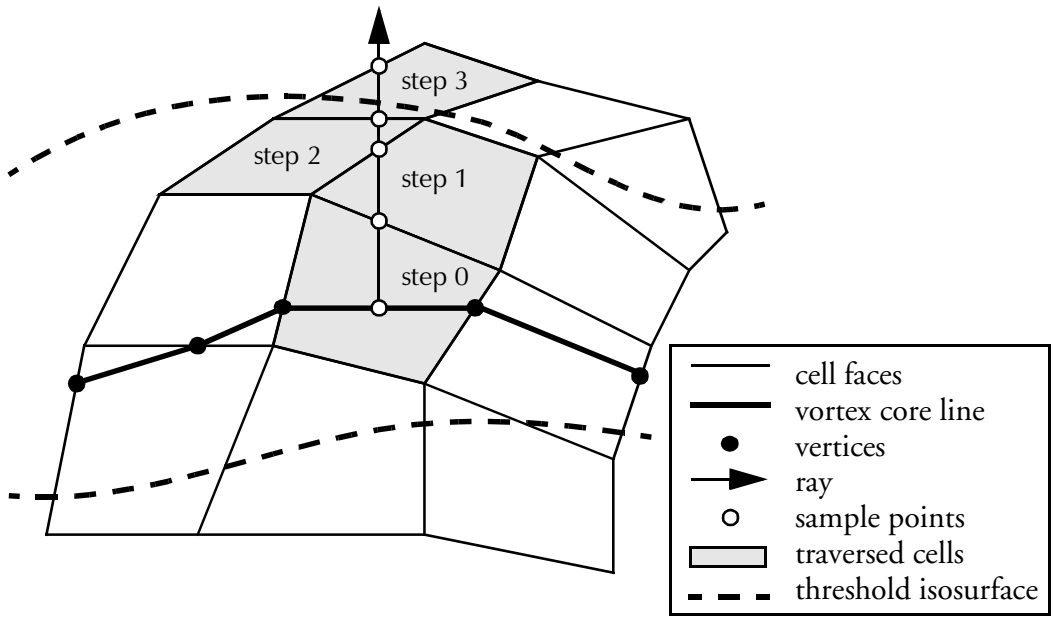


FIGURE 6.7 Ray traversal through the cells of the unstructured grid (projected view).

The drawback of our new propagation method is the necessity of raycasting the quadrangle cell faces. However, since we must traverse only few grid cells, this is more than compensated by the reduced number of sampling points. A more detailed description of the ray traversal and intersection computation is given in Appendix C. The algorithm stops when one out of several termination criteria has been fulfilled, namely if

- the scalar threshold has been crossed between two consecutive intersection points,
- the grid boundary has been reached,
- the maximum number of traversed cells (along the ray) has been reached,
- no new intersection point could be found on any face of the current cell.

In the latter three cases, the λ value of the last valid intersection point on the ray is returned. In the first case, the return value for λ is interpolated from the last two valid intersection points on the ray. A detailed pseudocode notation is given in Figure C.2.

6.2.3 Filtering of the ray lengths

The finding of the λ parameter values (and thus endpoints of the rays) strongly depends on the type of scalar field and on the scalar threshold. However, the raw data of this scalar field often originate from numerical CFD simulations and therefore contain inaccuracies. These inaccuracies can lead to artifacts such as high frequencies in the distribution of the ray lengths within a cross-section plane or self-intersections caused by rays of neighbouring cross-section planes (see Figure 6.8). Singularities show up in the resulting vortex hull, e.g. as holes or peaks in the triangular mesh representing its surface.

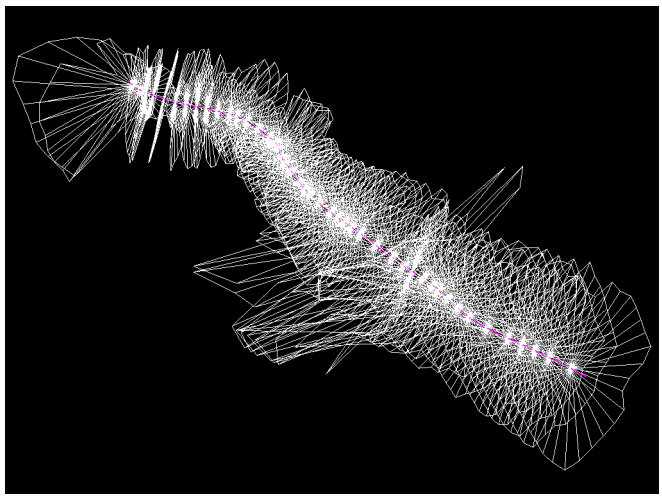


FIGURE 6.8 High frequencies of the ray lengths in the cross-sections of a vortex hull.

To cope with this undesirable effect, a user-defined filter mask can be activated for smoothing the λ values within the planes. A good choice is the use of a median filter because it eliminates numerical exceptions better than a simple average filter could do. The filter process loops over all rays of a certain plane and applies the filter mask to every ray, taking into account its k-neighbourhood (i.e. k neighbours to the left and k neighbours to the right, so the filter mask size is $(2k+1)$, see Figure 6.9). This filtering leads to significant noise removal (Figure 6.14 and 6.15), especially if the complete mesh is afterwards faired using a Laplacian operator (see Section 6.2.5, Figure 6.16 and 6.17).

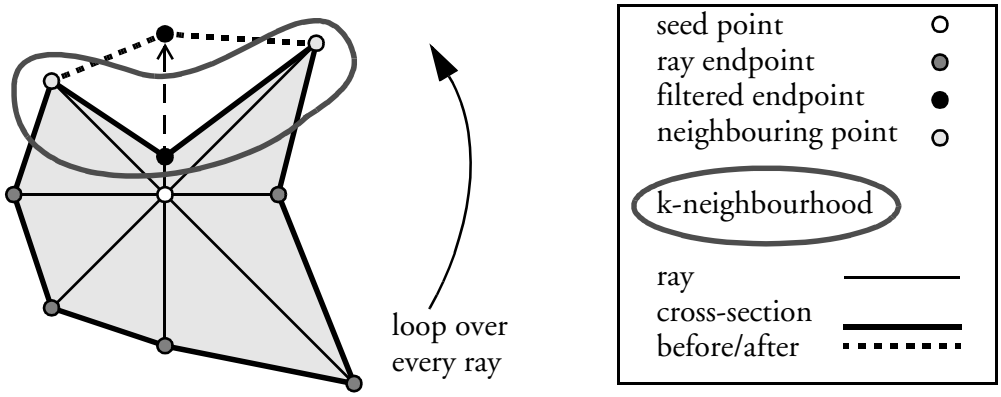


FIGURE 6.9 Smoothing the ray lengths by use of a filter mask (in this case k=1, filter size = 3).

6.2.4 Vortex hull assembly

Once the λ values of all rays in a certain plane have been found and eventually filtered, the endpoints of the rays are computed using the ray equation (Equation 6.5). Afterwards, the algorithm switches to the next core line segment and thus to the next cross-section plane. An *inner* tube section between two neighbouring planes is then built up using quadrangles. Each quadrangle connects two neighbouring ray endpoints of a plane and the corresponding two ray endpoints of the neighbouring plane. For instance, if the two neighbouring planes are successively indexed i and $i+1$, and the two neighbouring rays are indexed j and $j+1$ within any plane, then the four ray endpoints $Q_{i,j}$, $Q_{i,j+1}$, $Q_{i+1,j}$, $Q_{i+1,j+1}$ are connected to a quadrangle (see Figure 6.10). Before inserting it into a graphical object for the rendering pipeline, each quadrangle is subdivided into two triangles, which are easier to handle for normal calculation and lighting.

The beginning and ending of the core line have to be treated somewhat different from the inner segments. Here, we only have ray endpoints of *one* plane and an additional single vertex (the first or last vertex of the core line). In this case, we directly build triangles, every triangle connecting the vertex and two neighbouring endpoints Q_j, Q_{j+1} of the plane.

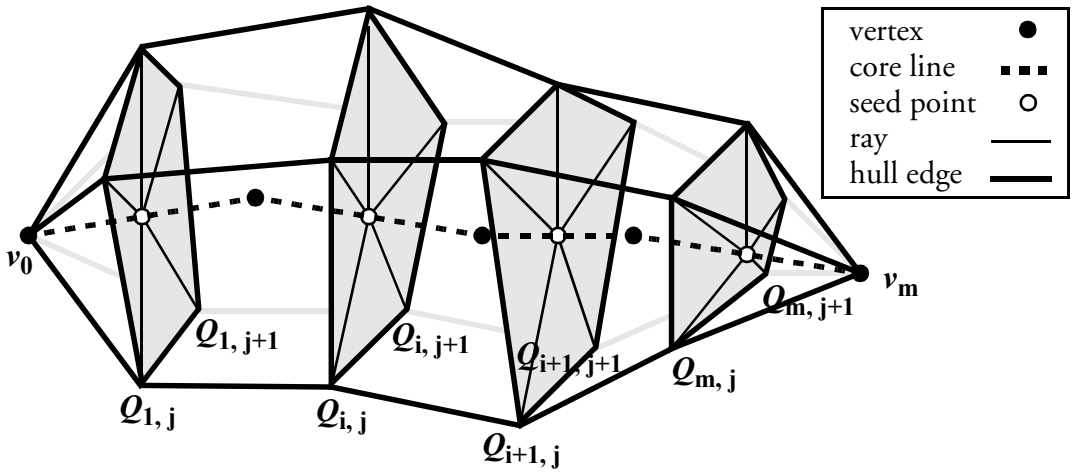


FIGURE 6.10 Construction of the final vortex hull from the ray endpoints (in this case, m = #segments = 4, n = #rays per fan = 5).

As a result, the final vortex hull is a closed triangulated surface completely surrounding the vortex core line. The geometrical complexity of a vortex hull can be computed as follows: Let m be the number of segments (and thus fans) of a core line, and n be the number of rays per fan. Then the number of inner tube sections is $m-1$, leading to $n(m-1)$ quadrangles and $2n(m-1)$ triangles. Additionally, each of the two pyramids built for the beginning and ending of the vortex hull consists of n triangles, leading to $2n$ triangles. The vortex hull has thus a total of $2n(m-1+1) = 2mn$ triangles, so its complexity is proportional to the number of core line segments and to the number of rays per fan.

This statement also holds for the complexity of the vortex hull construction algorithm (without filtering and assuming a constant average time for the ray extension, which can be justified by statistical reasons), as is evident from the pseudocode shown in Figure C.1. Some performance measurement results confirming this will be given in Section 6.3.

6.2.5 Mesh postprocessing

As is the case for CFD datasets, we can also reduce noise in the vortex hull, which is a geometric mesh. We only have to smooth *coordinates* rather than *data* at the mesh vertices. For this *fairing*, we once again use the diffusion equation (compare to Equation 4.9)

$$\dot{X}(t) = \frac{dX}{dt} = \lambda \cdot L(X) \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \qquad , \tag{6.6}$$

where t is the diffusion time, λ is a scaling factor, X denotes the mesh itself (a vector containing the coordinates of every mesh vertex), and $L(X)$ is the Laplacian of the mesh vertices (see the SIGGRAPH paper of Desbrun et al. [DMSB99]). For reasons of simplicity, we used in this case an *explicit* Euler scheme to integrate the diffusion equation over time. The aim is to recursively filter the mesh, i.e. to construct a series of meshes where each newly built mesh $X^{(k+1)}$ is computed from its predecessor mesh $X^{(k)}$ according to

$$X^{(k+1)} = (I + \lambda \cdot \Delta t \cdot L) X^{(k)} \tag{6.7}$$

where k is the number of the current iteration, Δt is the Euler time step and L is the matrix containing the Laplacian weights for all mesh vertices. In principle, our algorithm loops over each mesh vertex $\boldsymbol{x}_{i'}$, computes an approximated Laplacian $\boldsymbol{L}(\boldsymbol{x}_{i'})$ based upon its 1-neighbourhood (see below) and stores for every vertex $\boldsymbol{x}_{i'}$ its displacement

$$\Delta \boldsymbol{x}_{i'} = \lambda \cdot \Delta t \cdot \boldsymbol{L}(\boldsymbol{x}_{i'}) \, . \tag{6.8}$$

When all vertices have been processed, we add to every vertex its displacement:

$$\boldsymbol{x}_{i'} = \boldsymbol{x}_{i'} + \Delta \boldsymbol{x}_{i'} \, , \tag{6.9}$$

and perform the next Euler iteration. Since few iterations were sufficient in our case (see Section 6.3), we neglected the *mesh shrinking* effect. We approximated the Laplacian at a certain vertex using the *umbrella operator* proposed by Taubin [Tau95] (see Section 4.7.2), which is suitable even for unstructured meshes and requires a stability criterion of

$$\lambda \cdot \Delta t < 1 \, . \tag{6.10}$$

Let M be the *valence* (= number of direct neighbours) of vertex $\boldsymbol{x}_{i'}$ and $N_1(i')$ the set of direct neighbours connected to $\boldsymbol{x}_{i'}$ by an edge. Then the umbrella operator computes to

$$\boldsymbol{L}(\boldsymbol{x}_{i'}) = \frac{1}{M} \cdot \sum_{j' \in N_1(i')} (\boldsymbol{x}_{j'} - \boldsymbol{x}_{i'}) \tag{6.11}$$

which is the average vector of all edges pointing from $\boldsymbol{x}_{i'}$ to its direct neighbours. In our implementation, the computed vortex hull consists of quadrangles (see Figure 6.10) and is (apart from the two endcaps) a *cylindric structured* mesh. We stored this mesh as a two-dimensional array $Q(i = 0 \ldots m+1, \; j = 0 \ldots n+1)$ of vertices (see Figure 6.11), where m is the number of cross-section planes, n is the number of rays per fan and (i, j) denote the plane and ray where the mesh vertex $Q_{i,j}$ was found as a ray endpoint. These *inner vertices* are stored in the columns $(i = 1 \ldots m)$ and rows $(j = 1 \ldots n)$ of the vertex array. The first vertex $\boldsymbol{v}_0$ and last vertex $\boldsymbol{v}_m$ of the vortex core line, which are the tips of the vortex hull endcaps, have to be treated specially. For reasons of simplicity, they are multiplied $n+2$ times and copied to the left column $(i = 0)$ and right column $(i = m+1)$ of the vertex array. Futhermore, we made a *cyclic continuation* of the ray endpoints within

each plane. Since the vortex hull has a cylinder topology, we copied row $(j = 1)$ to row $(j = n + 1)$ and row $(j = n)$ to row $(j = 0)$. Concerning the boundary conditions of the Laplacian smoothing, we decided to fix the vertices $\boldsymbol{v}_0$ and $\boldsymbol{v}_m$, which helps minimise the shifting of the overall vortex hull due to the Laplacian smoothing. Therefore only the inner vertices have to be displaced during the fairing process.

We can now apply the umbrella operator to every inner node of the vertex array $(i = 1 \ldots m,\ \ j = 1 \ldots n)$ without testing for array boundaries, provided that we restrict the Laplace stencil to 4 or 8 neighbours per node. Of course the 1 st and n-th vertex of each inner column must again be circularly copied at the end of each Euler iteration step. The left and right column remain unchanged due to the fixed boundary condition.

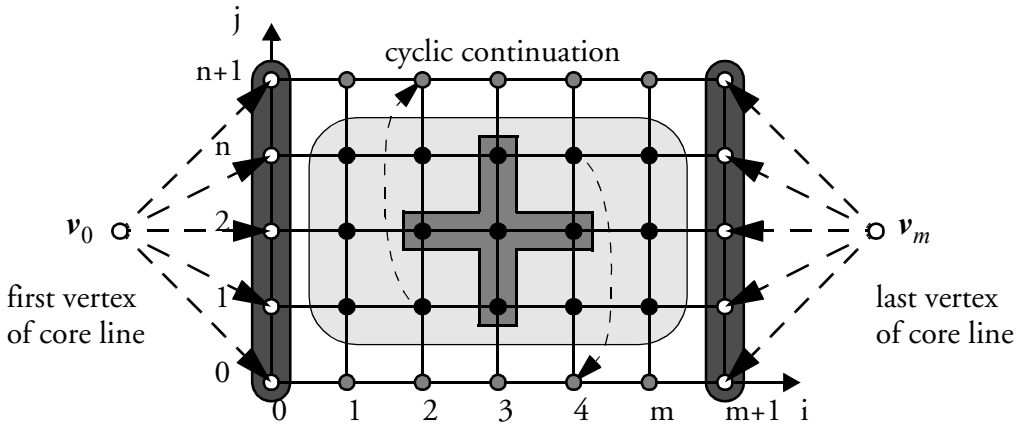


FIGURE 6.11 Storage of mesh vertices and application of umbrella operator (cross stencil). (In this case, m = #segments = 5, n = #rays per fan = 3, M = valence = 4).

6.3 RESULTS

The modified vortex hull algorithm was tested on an SGI Octane (640 MB main memory, MIPS R10000 CPU and MIPS R10010 FPU running at 250 MHz) for three different unstructured grid datasets, using the Levy vortex extraction method and vortex strength as the scalar field. Table 6.1 compares the computational times of the five phases

- setup of the two vector fields and the scalar field,
- extraction of the vortex core lines,
- construction of the appropriate vortex hulls,
- filtering of the rays by a median filter,
- fairing of the resulting mesh by Laplacian smoothing.

From the table, it is evident that the time consumed for vortex hull construction is indeed proportional to the number of core line segments and to the number of rays per fan, as was expected by the theoretical complexity analysis in Section 6.2.4. Furthermore, the vortex hull construction time is relatively small in comparison to the vector field setup and core line extraction time, provided that the number of rays per fan is moderate.

For large numbers of rays per fan, the hull construction time increasingly dominates the costs. However, this is in practice no serious limitation since for most applications, about 30 rays per fan are sufficient for good-quality results (especially if the hull mesh is afterwards faired using the umbrella operator). Since our implementation uses a simple

sorting method with quadratic order of complexity, sorting the lambda values of the unfiltered rays becomes a bottleneck for large filter sizes. Of course the sorting time could be reduced by a more sophisticated method like *Quicksort*. As was to expect, the time for Laplacian mesh fairing of the vortex hull is proportional to the number of Euler iterations.

Figures 6.12 to 6.17 show some examples of vortex hulls constructed by the algorithm. The underlying datasets are based upon the artificial *bent helix* and the Francis draft tube also treated in Chapter 5. Figure 6.12 shows a wireframe representation of the vortex hull around the bent helix, based upon the distance field of the vortex core line with 5 rays per fan and regular pentagons as cross-sections. For the Francis draft tube, the distance field option with 36 rays per fan yields a quasi-cylindric shape of the vortex hull (Figure 6.13).

The noise originating from the fragmented vortex core line at the Francis draft tube inlet clearly increases when using vortex strength (Figure 6.14), since this scalar field contains high frequencies. As postulated in Section 6.2.3, median filtering of the ray lengths removes a large amount of noise, since it eliminates numerical exceptions among the ray lengths (Figure 6.15). However, the resulting vortex hull looks somewhat "shrivelled", containing bumps and still self-intersections of neighbouring ray planes.

To cope with this effect, we faired the mesh with and without previous ray filtering. Pure Laplacian smoothing reduces the raw data noise but fails to remove the numerical exceptions at the region of maximum twist of the vortex (Figure 6.16). A combination of median ray filtering and subsequent Laplacian smoothing yields better results, leading to a smooth vortex hull while still preserving the characteristic shape and thickness of the vortex, despite of a slight shrinking (Figure 6.17). In practice, 4 neighbours per node, a step size of $\lambda \cdot \Delta t = 0.9$ and 10 explicit Euler iterations were sufficient for good visual results.

turbine design	core line segments *(m)*	rays per fan *(n)*	vortex hull triangles *(2mn)*	filter size/ Euler iterations *(k)*	vector field setup CPU[s]	core line extr. CPU[s]	vortex hull constr. CPU[s]	ray median filtering CPU[s]	Laplace mesh fairing CPU[s]
bent helix, 2000 nodes	19	10	380	0	0.08	0.04	0.07	0.00	0.00
		50	1900				0.34		
		100	3800				0.67		
Francis original, 654770 nodes	283	10	5660	0	30.39	16.00	1.31	0.00	0.00
		50	28300				6.38		
		100	56600				12.70		
Francis modified, 654770 nodes	418	100	83600	0	30.48	15.97	33.49	0.00	0.00
				1				0.07	0.15
				5				0.41	0.79
				10				1.12	1.59
				50				17.18	7.97

TABLE 6.1 Performance analysis of the vortex hull computation.

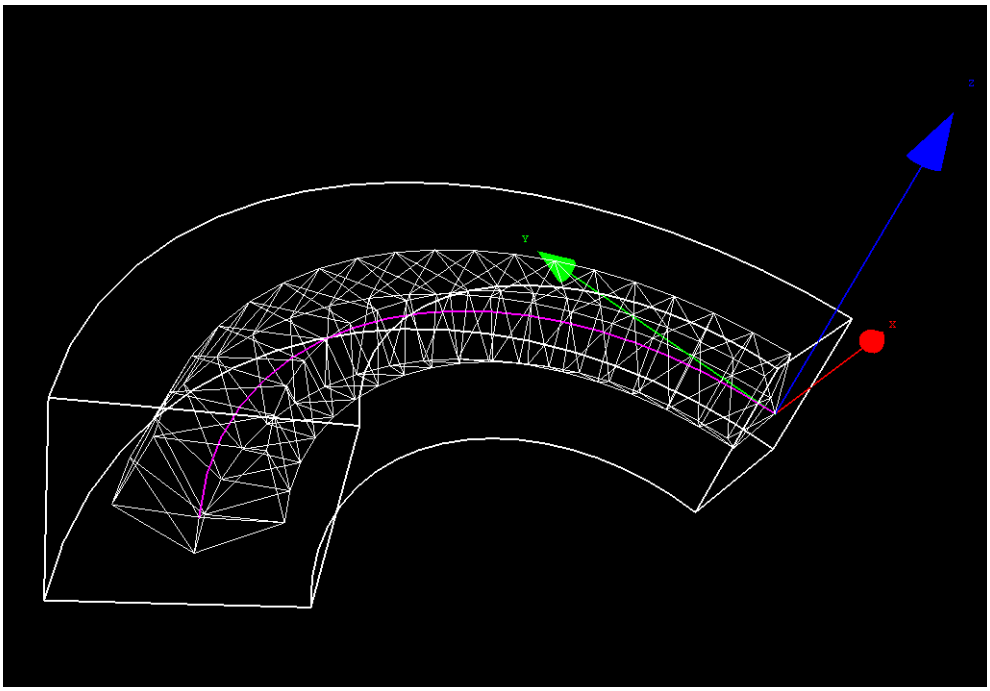


FIGURE 6.12 Vortex core line and hull construction for the bent helix dataset, based on Levy method and distance field (m = 19, n = 5, k = 0).

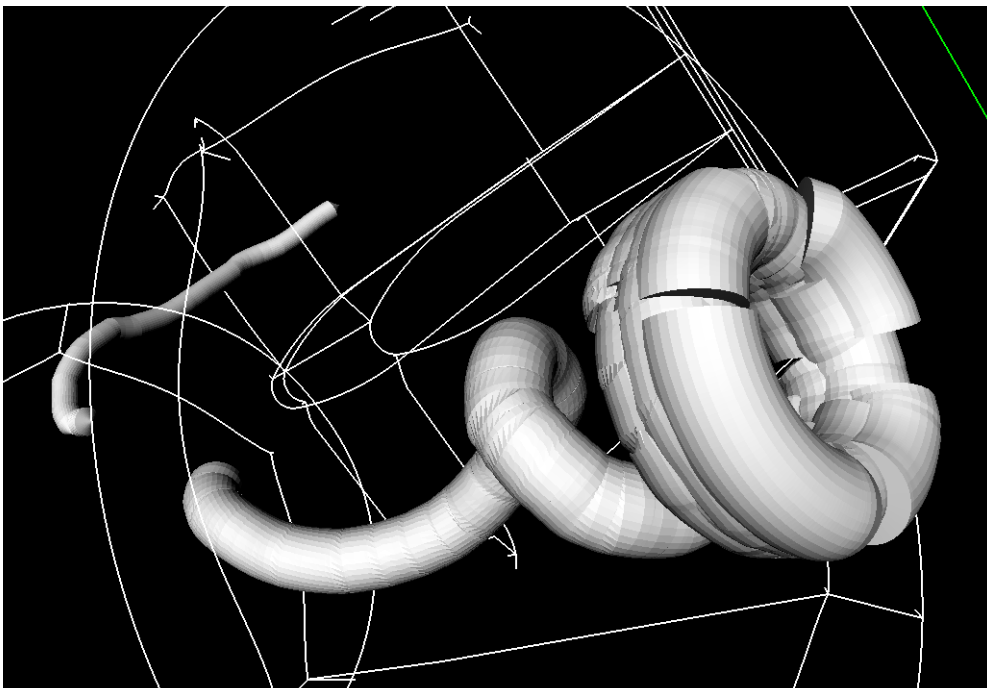


FIGURE 6.13 Vortex hulls based on Levy method and distance field (m = 283, n = 36, k = 0). Noise in the upper region (Francis draft tube inlet) due to unsmoothed CFD data.

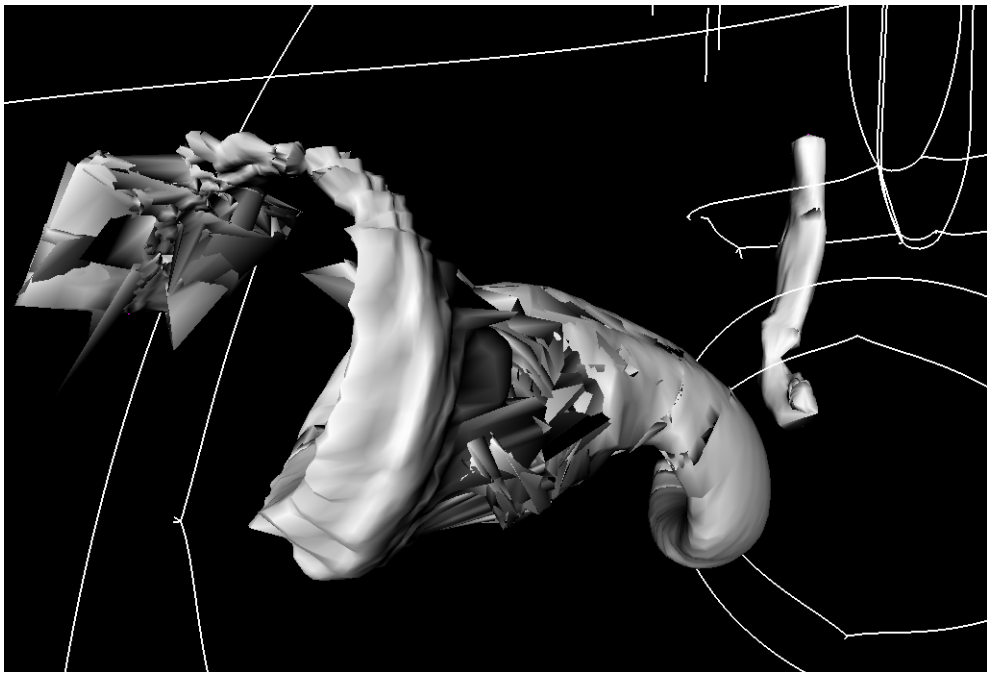


FIGURE 6.14 Vortex hulls for Levy method and max. vortex strength of 3.0 (m = 283, n = 36, k = 0). High-frequency noise at region of maximum twist of the vortex requires smoothing. See also Colour Figure A.3 on page 123.

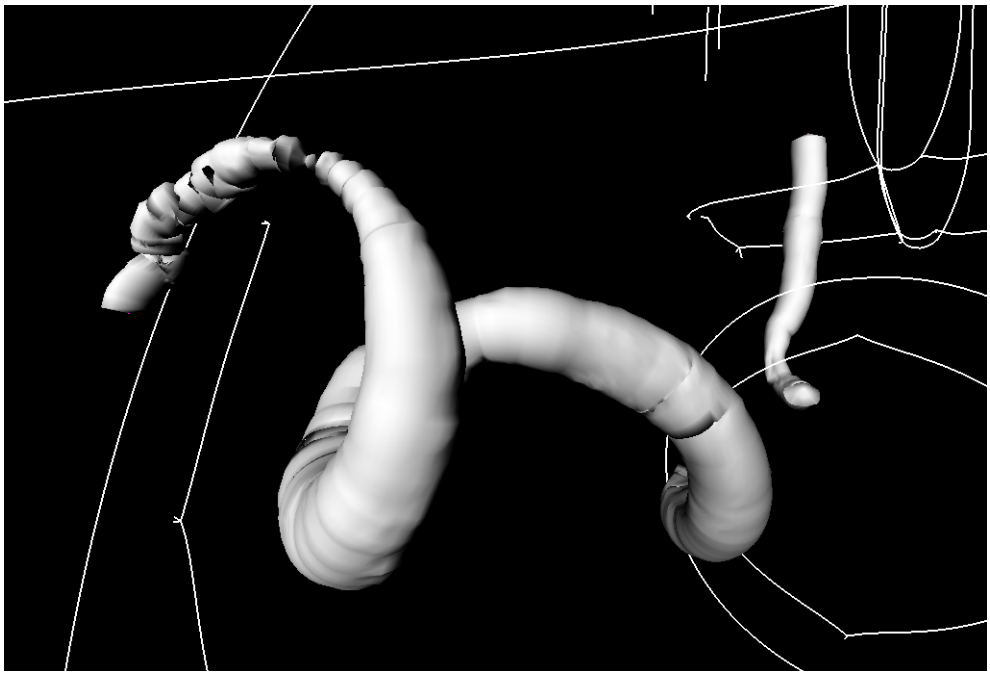


FIGURE 6.15 Vortex hulls for Levy method and max. vortex strength of 3.0 (m = 283, n = 36, k = 15). Median ray filtering reduced noise especially at region of maximum twist. See also Colour Figure A.3 on page 123.

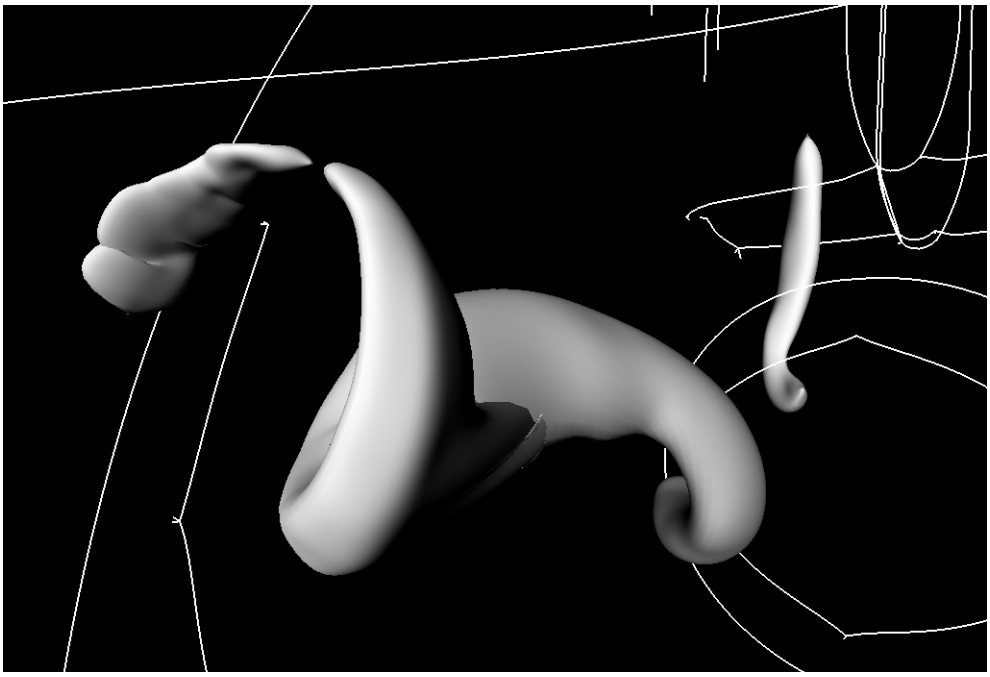


FIGURE 6.16 Vortex hulls for Levy method and max. vortex strength of 3.0 (m = 283, n = 36, k = 0). Laplacian mesh fairing (4 neighbours, $\lambda \cdot \Delta t = 0.9$, 10 iterations) without ray filtering. See also Colour Figure A.3 on page 123.

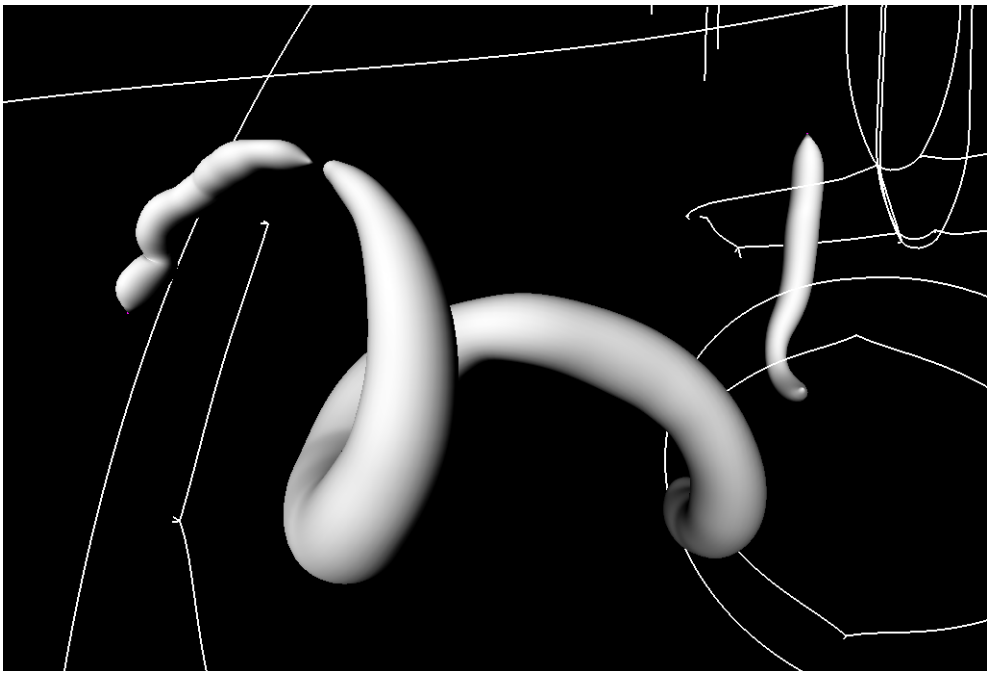


FIGURE 6.17 Vortex hulls for Levy method and max. vortex strength of 3.0 (m = 283, n = 36, k = 15). Combination of median ray filtering and subsequent Laplacian mesh fairing. See also Colour Figure A.3 on page 123.

CHAPTER 7

SELECTIVE PARTICLE TRACING

Tracing and rendering of moving particles is a standard technique for the visualisation of stationary or time-dependent 3D vector fields. Nevertheless, it is hard to find better techniques for this type of data. Moving streamlines, for instance, are not physically meaningful (see Chapter 3), while path lines should be reserved to static visualisations. Streaklines are an adequate technique, but by discretising them into a series of particles, some additional information can be conveyed.

Particles, in contrast, have the advantage of a small glyph size, which minimises occlusion in projected views. Furthermore, particle visualisations compare naturally to some types of flow experiments. The hydraulic CFD simulations which led to this work are often accompanied by flow experiments where the flow is measured and visually examined in scale models. In such experiments, cavitation bubbles can be observed (even in rotating machine parts if viewed under stroboscopic lighting, see Section 7.3).

In this chapter, we explore techniques for the purpose of visualising isolated moving particles in time-dependent flow data. Our primary industrial application is the visualisation of the *vortex rope*, a rotating helical structure which builds up in the draft tube of a Francis-type water turbine. The vortex rope can be characterised by high values of normalised helicity, which is a scalar field derived from the velocity data given in the underlying CFD datasets. In two related applications, the goal is to visualise the cavitation regions near the runner blades of a Kaplan turbine and a storage pump, respectively. Again, the flow structure of interest can be defined by a scalar field, namely by low pressure values.

In contrast to previous particle tracing approaches, we trace the particles *selectively*, which means that we focus on special *regions-of-interest (ROIs)* defined by two thresholds of a scalar field. We compute and visualise particles only within these regions, which significantly reduces the amount of data to be processed. This method not only gains storage efficiency but also reduces occlusion problems. Furthermore, we propose a particle seeding scheme based on *quasi-random numbers*, which minimises visual artifacts such as clusters or patterns.

7.1 PARTICLE TRACING VERSUS OTHER TECHNIQUES

Instead of tracing discrete particles, 3D textures could alternatively be used to visualise the flow field, e.g. by extending the *Lagrangian-Eulerian Advection (LEA)* algorithm proposed by Jobard et al. [JEH01] from two to three dimensions. However, we did not choose this approach because for time-dependent 3D textures, the texture loading and rendering time becomes significant. Also, hardware support for 3D texture is not generally available.

Concerning the seeding of the particles, the goal of evenly spaced glyphs has led to the *streamline placement* techniques [TB96], which, in principle, could be extended to four dimensions for time-dependent flow. Our approach is instead to exploit the *conservation of mass* property of physical flow fields. For incompressible flows, the conservation of mass means a divergence of zero, therefore the particle density remains constant if it was initially constant. For compressible flows, an initial particle distribution can be made to reflect the density of the medium.

A common problem of particle-based flow visualisation is the need for injecting new particles during the time evolution. The injection is necessary to maintain a uniform density of the particles over time. Texture-based methods accomplish this by continually adding noise. The noise is smeared enough to avoid visible artifacts by the large filter kernels typically used in *line integral convolution (LIC)*. But this solution is obviously not applicable to the discrete particles in our case. Instead, our strategy is to extend the region-of-interest by a few layers of "buffer" cells, where the particles are kept invisible. When particles enter or leave the ROI, their visibility is changed smoothly. In the buffer cells, the correct particle density can now be maintained by adding and deleting invisible particles. And because of the mass conservation, this carries over to the visible particles, too.

7.2 FLOW REGIONS OF INTEREST

While steady flows can be explored by scanning the data domain (e.g. extracting isosurfaces at different scalar levels or placing seed points for streamlines at different locations), this is not an option for time-dependent data. But then, visualisation has to be sparse, i.e. it can not simultaneously depict the flow everywhere in the computational domain. If a certain level of detail is expected, visualisation has to be constrained to a region, and as a consequence, the data must be explored repeatedly. In this chapter, we use the approach of defining flow regions in a data-guided way. Flow regions of our interest are vortices and cavitation regions, but other types of flow regions, such as recirculations, could be treated similarly.

We experienced that typical regions-of-interest consist of a few percents of all grid cells. Hence, it makes sense to generate and trace the particles only where needed. A common practice is to release particles from locations evenly spaced along lines or circles and at fixed time intervals. One can then observe how these geometrical patterns are deformed over time. A different approach, which we pursued instead, is to *randomly* spread particles in space, aiming at a uniform particle density while avoiding patterns and clustering.

We will generally assume that regions-of-interest (ROIs) can be specified by a scalar field. In many cases, such a scalar exists among the CFD data channels or can easily be derived from them. For example, cavitation regions are characterised by low pressure values (pressure values falling below the vapor pressure). And vortices can be characterised by high values of either helicity or normalised helicity.

There are miscellaneous possibilities to define a ROI by means of scalar thresholds:

1. In Chapter 3, we mentioned a purely region-based approach, namely to extract an isosurface for a certain scalar threshold. The result of such an isosurface extraction is often more than one connected component. It can then be necessary to perform a selection among these components, especially because the resulting isosurface often contains *false positives*, that means undesired additional solutions (e.g. the region at the grid boundary of the turbine case shown in Figure 7.1).
2. In the case of a vortex, there is an alternative way to define a ROI based upon an extracted vortex core. In Chapter 5 and Chapter 6, we described a mixture of region- and feature-based techniques, namely how to detect a vortex core line and to radially expand the core line as long as a certain scalar field (e.g. the pressure or the vortex strength) is above or below a given threshold. The resulting vortex hulls significantly reduce the false positives, separate different vortices from each other and help better perceive the shape of the vortex than a pure isosurface could do.
3. In this chapter, we will pursue a third possibility, which is a mixture of region- and integration-based methods. Similar to isosurface extraction, we will define the ROI only by scalar values, without precomputing flow features such as vortices. Instead, we will trace particles in regions limited by *two* scalar thresholds controlling their visibility. The flow features will implicitly be contained in the ROI and detected by observing the overall structure of the moving particle stream.

In the remainder of this chapter, we will propose the industrial context of our particle tracer and then describe its specific visualisation techniques in more detail.

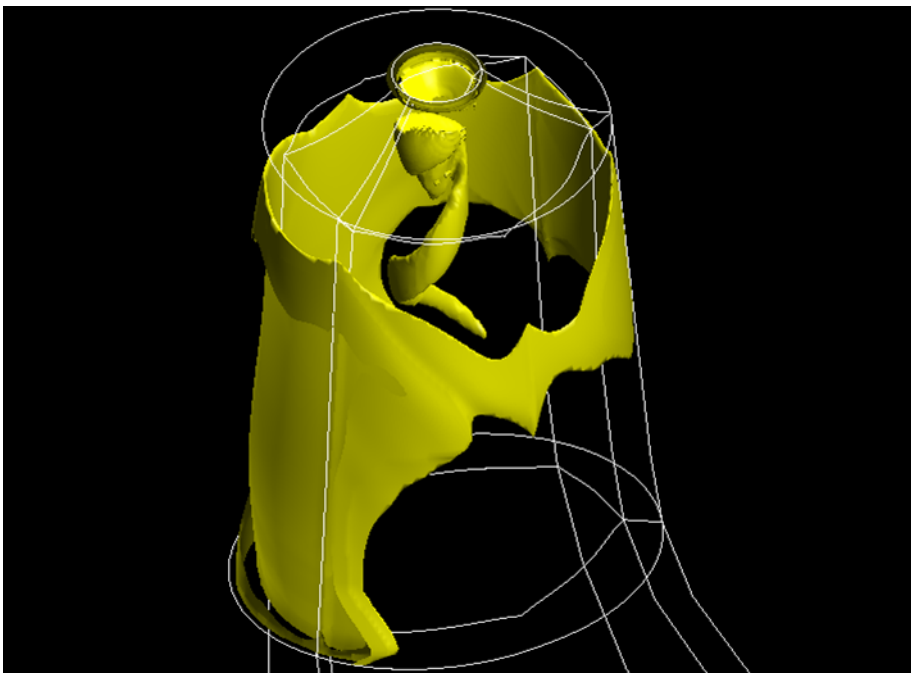


FIGURE 7.1 ROI definition by a scalar field and threshold. The isosurface consists of several connected components, including false positives at the boundary of the turbine case. See also Colour Figure A.1 on page 121.

7.3 INDUSTRIAL APPLICATION

The main industrial case under investigation was the flow field in a Francis water turbine. The flow in the turbine passes through the vaneless spiral casing and enters the stationary stay and guide vanes, which accelerate the flow. The flow afterwards decelerates through the Francis *runner* and enters the *draft tube*, which is the last component of the turbine. A draft tube generally is a diffuser with or without a bend, whose objective is to convert the available kinetic energy into a rise in pressure. Depending on the operating conditions, the swirl of the flow at the runner exit (and thereby at the draft tube inlet) is strong enough to cause a flow instability, which is called the draft tube vortex. The typical shape of a draft tube vortex is that of a rotating helix. Due to the low pressure in the center of the vortex, the flow is often cavitating for a wide range of operating conditions, so that on the test rig, the vortex rope can be naturally visualised by means of *cavitation bubbles*, see Figure 7.2.

FIGURE 7.2 Photograph of a cavitating vortex rope on the test rig (image courtesy of VA Tech Hydro, Zurich). See also Colour Figure A.4 on page 124.

The part-load vortex rope is of technical relevance for two reasons. Firstly, it causes serious variations in relative pressure within the runner and on its hub and shaft, as well as in the draft tube (see Figure 7.3 and Figure 7.4). Secondly, it leads to an unstable through-flow and unstable power output. The shaft vibrations caused by the pressure variations can cause severe damage of the machine. Therefore in some power plants the operating range of the machine is restricted in order to avoid a strong vortex and the resulting damage.

The design of the runner blades and the hydraulic contour of the draft tube have a strong influence on the onset and strength of the draft tube vortex. It is therefore important to understand the details of the vortex rope flow in order to define physically well founded design rules for a turbine design which is safe w.r.t. the draft tube vortex.

The visualisation methods demonstrated in this chapter are based on transient CFD simulations of the draft tube vortex in a pump turbine. The 3D Navier Stokes equations were solved using the commercial code *CFX-TASCflow* [Tec00] with a circumferentially averaged inlet velocity profile resulting from a CFD simulation of the runner flow.

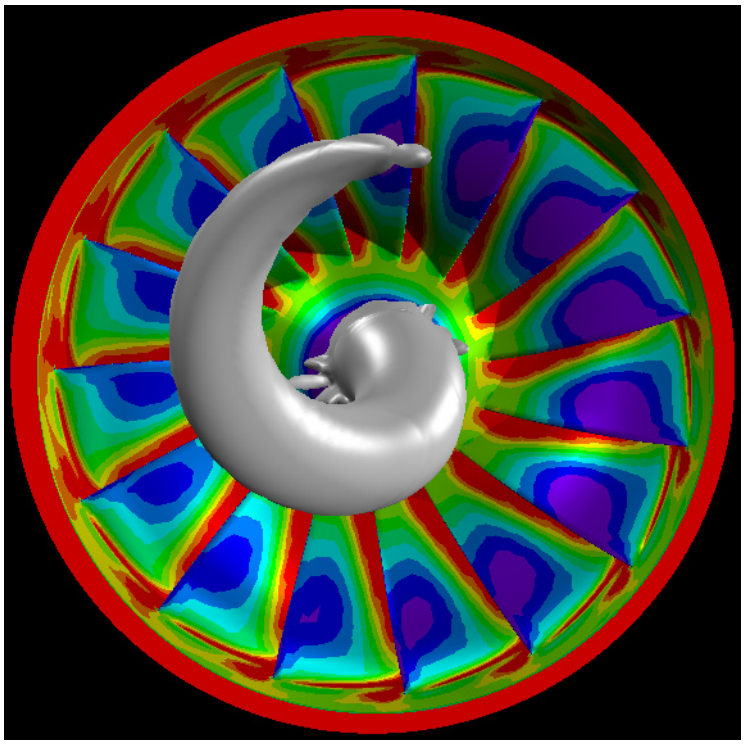


FIGURE 7.3 Variations of relative pressure at the runner blades of a Francis turbine (bottom view, vortex rope depicted as pressure isosurface). See also Colour Figure A.4 on page 124.

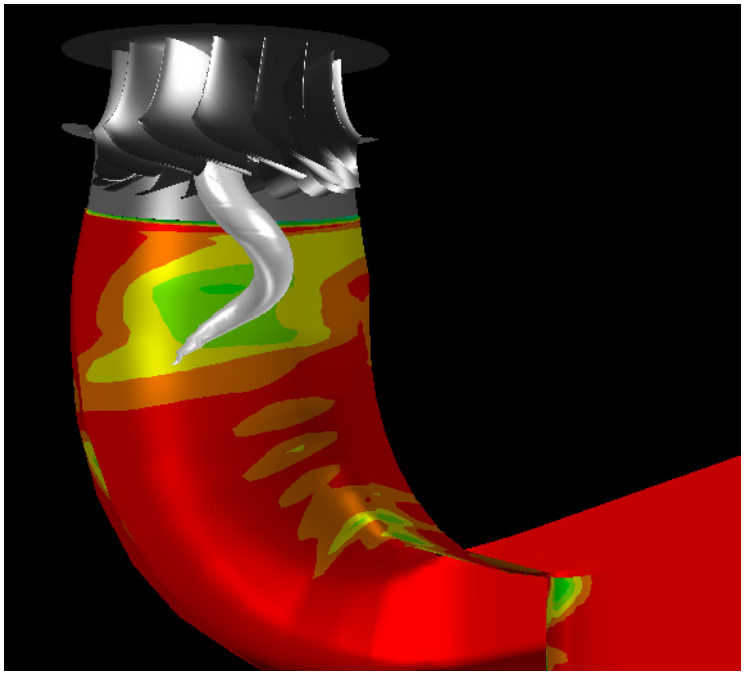


FIGURE 7.4 Variations of relative pressure in the draft tube of a Francis turbine (side view, vortex rope depicted as pressure isosurface). See also Colour Figure A.4 on page 124.

7.4 VISUALISATION TECHNIQUES

For the purpose of selectively visualising the flow in regions-of-interest, we modified the standard particle-based technique in the following ways:

- The particles are visible only in the ROI, with a smooth transition (fading) if they enter or leave the ROI. This method will reduce occlusion problems when rendering the resulting glyphs, and save memory as well as computation time.
- The particles are uniformly distributed in a way that no artifacts such as clusters or regular patterns occur. This way, the particle density roughly corresponds to the density of the medium during the complete tracing process. For incompressible flow, this means that the particle density is initially constant and remains constant over time.

We will in the following sections describe in more detail the specific visualisation techniques we used for our novel particle tracer.

7.4.1 Cell classification

Initially and after each time step of the particle tracing algorithm, the grid cells are classified as *inner*, *buffer* or *outer* cells. The cell classification is based on the scalar field $s(\boldsymbol{x})$ of the CFD dataset and two particle visibility thresholds s_0, s_1 (see also Section 7.4.3):

- The inner cells are those where at least one corner has a scalar value exceeding the lower visibility threshold, i.e. which satisfies $s(\boldsymbol{x}) > s_0$. The inner cells thus completely cover the region of partial and full visibility of the particles (see Figure 7.5).
- The buffer cells are those in a topological k-neighbourhood of the inner cells. Additionally, all inner cells within the k-neighbourhood of an inflow boundary are regarded as buffer cells.
- The remaining cells (often covering about 90 percent of the grid) are classified as outer cells.

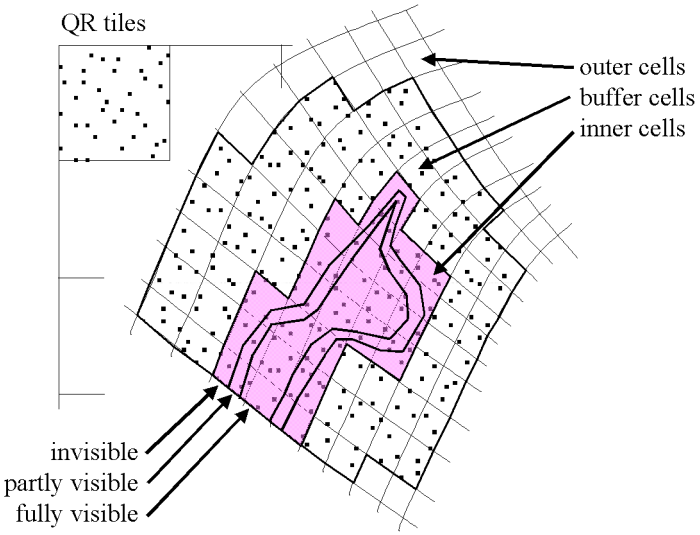


FIGURE 7.5 Classification of cells, particle visibility, and tiles of quasi-random points.

7.4.2 Particle seeding

The goal of the particle seeding is to generate a uniform distribution of the particles while avoiding clusters and regular patterns. In multi-dimensional spaces, *quasi-random* sequences give better results than *pseudo-random* sequences or *jittered regular* samples. The latter two methods lead to clustering (Figure 7.6 (a) and (b)) or regular patterns (Figure 7.6 (c)).

Sobol' quasi-random sequences were proposed [SH95] and have recently been used [SMA00] for LIC methods. An advantage of the Sobol' points is that they can be tiled with no visual seams, as is shown in Figure 7.6 (d) and (e). This allows us to tile the physical space and to create quasi-random points only for a single tile, which is repeatedly used wherever it covers the ROI (extended by a few layers of buffer grid cells, see Section 7.4.1).

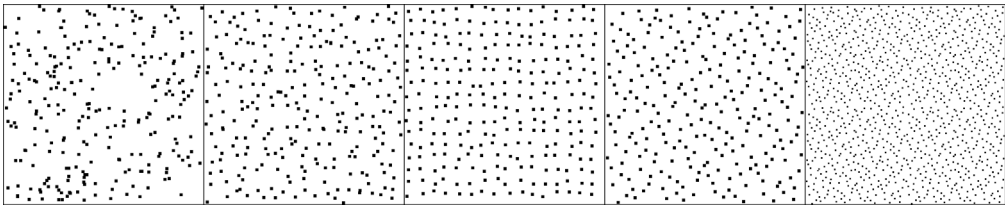


FIGURE 7.6 (a): Pseudo-random samples. (b), (c): jittered regular samples. (d): Sobol' points. (e): four tiles of Sobol' points.

7.4.3 Particle visibility

In our application, the ROI is defined by high values of a scalar field $s(\boldsymbol{x})$, which is either directly given in the datasets (e.g. pressure with inverted sign) or has previously been derived from them (e.g. helicity, computed from velocity). The particles are *invisible* as long as at their location, the scalar field $s(\boldsymbol{x})$ is below a predefined lower threshold s_0. Above an upper threshold s_1, they are *fully visible* (see Figure 7.7). Between the two thresholds, a smooth transition is made using either semi-transparency or reduced size of the rendered glyphs. Invisible particles can be generally treated as nonexistent by the implementation, which saves considerable memory and computation time. However, in the vicinity of the ROI it is good to trace invisible particles, too (see Section 7.4.1).

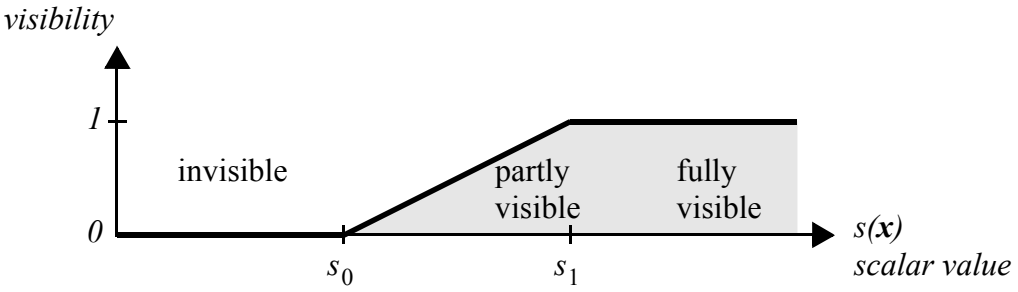


FIGURE 7.7 Smooth visibility transition by two scalar thresholds.

7.4.4 Particle tracing

The crucial point of our algorithm is that it traces particles only in the inner and buffer cells, which cover a minority of the grid. The particles are initially distributed over all inner and buffer cells and periodically advected using *Heun*'s second-order integration method, then covering a different area. As long as this area completely covers all inner cells, everything is fine. However, this condition can fail - not only because of the particle movement but also because the set of inner cells is dynamic due to the time-dependent scalar field.

After a certain *update interval*, we therefore clear all particles in the outer and buffer cells and generate new particles in the buffer cells. Since these particles are invisible by definition of the buffer cells, the replacement has no visual impact. Nevertheless, it is important that old and new particles join seamlessly at the border between buffer and inner cells, since the new particles may later become visible. By using Sobol' quasi-random points, we can meet this requirement sufficiently for the purpose of visualisation. And by making the ring of buffer cells wide enough, we avoid replacing the particles too often. Based on these considerations, we designed a selective particle tracer of the structure shown in Figure 7.8.

```
Create initial particles in INNER and BUFFER cells.
currentTime := startTime
while currentTime < endTime
  Get current velocity field and scalar field.
  if update interval has expired
    Update classification for each cell
    (according to the scalar values of its nodes).
    Delete all particles in OUTER and BUFFER cells.
    Generate new particles in BUFFER cells.
  end if
  currentTime := currentTime + timeStep
  for each particle in the particle list
    Get current velocity of the particle.
    Calculate new position of the particle (Heun integration, 2nd order).
    Find new cell and local coordinates of the particle.
    Classify the particle according to its cell classification.
  next particle
  if drawing interval has expired
    Draw the partly and fully visible INNER particles.
  end if
end while
```

FIGURE 7.8 Pseudo-code of the selective particle tracer.

The particles in the buffer cells can be replaced at fixed time intervals, although a safer method would be to release marked particles at the outer boundary of the buffer cells. A marked particle entering an inner cell would trigger the replacement. Instead of generating a new set of quasi-random points for each replacement operation, we experimented with a cheaper solution, namely to just randomly offset the old Sobol' tiles. This corresponds simply to a translation of all tiles, so there is no need to actually modify point coordinates.

A drawback of our algorithm is that the integration of particle paths in unstructured grids requires frequent incremental point searches [Bun89]. However, a *global* point search is only necessary for newly generated particles, mostly because there is no spatial coherence in quasi-random sequences. To optimise the global point search, we organise the particles in a regular grid, which is a refinement of the tiling grid.

7.4.5 Conservation of mass

When the velocity field (and the density in the case of a compressible flow) is interpolated from the node data, the resulting fields are expected to be mass-conservative. However, this is not the case if the standard techniques are used, namely trilinear interpolation in hexahedral grid cells and linear interpolation in tetrahedral grid cells. By comparing the influx and efflux of hexahedral grid cells, we observed that typically 1-5% of the mass is numerically "lost" due to the trilinear interpolation of the node data. This loss could in our case be reduced to about 0.5-2% by regarding the *dual grid* (dual cells are the control volumes of the *finite volume* method used by solvers such as CFX-TASCflow for the flow simulation).

In principle, a mass-conservative interpolation could alternatively be used [FWCS97]. However, this would require the computation of two *global stream functions* for every simulated or interpolated time step. We decided to use a special interpolation only in cells next to solid (no-slip) boundaries, where most of the numerical mass loss occurs. In these boundary cells, the main effect of trilinear interpolation is that particles tend to "stick" at the boundary, eventually compromising the postulated uniform distribution. To prevent this, we do not allow particles to get closer to a solid wall than a certain fraction of a cell. We achieve this by clamping the appropriate local coordinate after each integration step and neglecting the velocity component of the particle which directs towards the wall. This method is common practice [Bun89] and can be justified by the assumption of particles with a spatial extent (see Figure 7.9).

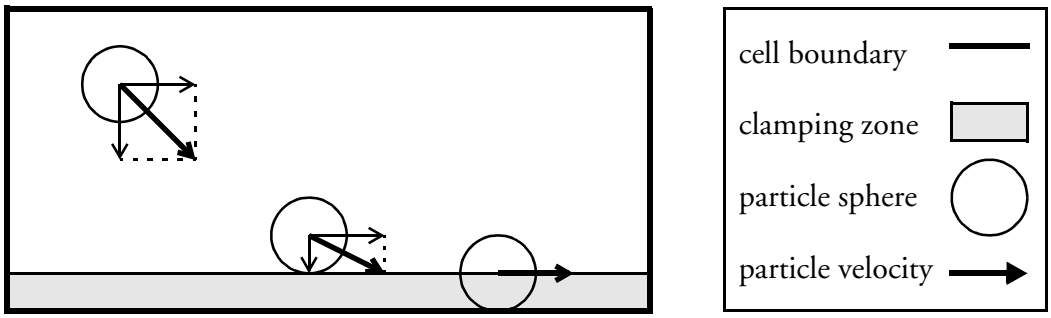


FIGURE 7.9 Clamping local coordinates at the grid boundary (2D representation).

7.4.6 Particle rendering

We used different techniques for the graphical output of the traced particles. An obvious technique is to render the particles as *spheres*, which minimises occlusion problems to the lowest possible degree and is easiest to implement. Furthermore, spheres are a natural equivalent to the cavitation bubbles observed in flow experiments (see Figure 7.2).

A good alternative technique is to render the particles as *streamlets*, which depict short pieces of their local path lines (see [WLG97]). The occlusion problem is in this case solved by alpha-texturing. For the shape of the streamlets, we used quadrilateral strips (ribbons) made up of *billboard polygons*. Billboard polygons are permanently directed towards the viewer and give the illusion of 3D rather than flat objects. Moreover, they can effectively be illuminated with diffuse and specular reflexion.

7.5 RESULTS

We applied our selective particle tracer to CFD data for different types of water turbines and using various rendering techniques.

Figure 7.10 shows a taking of a Kaplan turbine, which was investigated under different operating points and visualised when operating at its best efficiency point. The focus of the corresponding video animation was on the cavitation regions which developed on the suction side of the Kaplan runner blades and were identified by regions of low pressure. The cavitation bubbles were in this case visualised by rendering the particles as spheres.

A similar procedure was applied to the waterflow through a storage pump, which is depicted in Figure 7.11. The original machine is located at Ffestiniog (North Wales) between two water reservoirs at different heights above sea level. It acts as a turbine connected to an electric generator during peak hours (daytime) and to inversely pump water back to the higher reservoir during off-peak hours (nighttime). Four of these water turbines at the power station can generate 360 MW of electricity within 60 seconds of the need arising. As can be seen from the picture, rendering the particles as spheres was a good choice to keep the occlusion problems low, despite of the high particle density in the flow.

Finally, Figure 7.12 shows an example for rendering particles as streamlets. The flow was visualised in the draft tube after passing the runner of a Francis turbine similar to that investigated in the previous chapters. In this case, we used fewer particles, allowing for alpha-textured billboard polygons to give a better impression of the particle motion. Streamlets as rendering primitives turned out to be particularly suited for video animations - since even a freeze image gives a good impression of the flow behaviour and direction, the transition between two consecutive frames appears smoother than when using spheres.

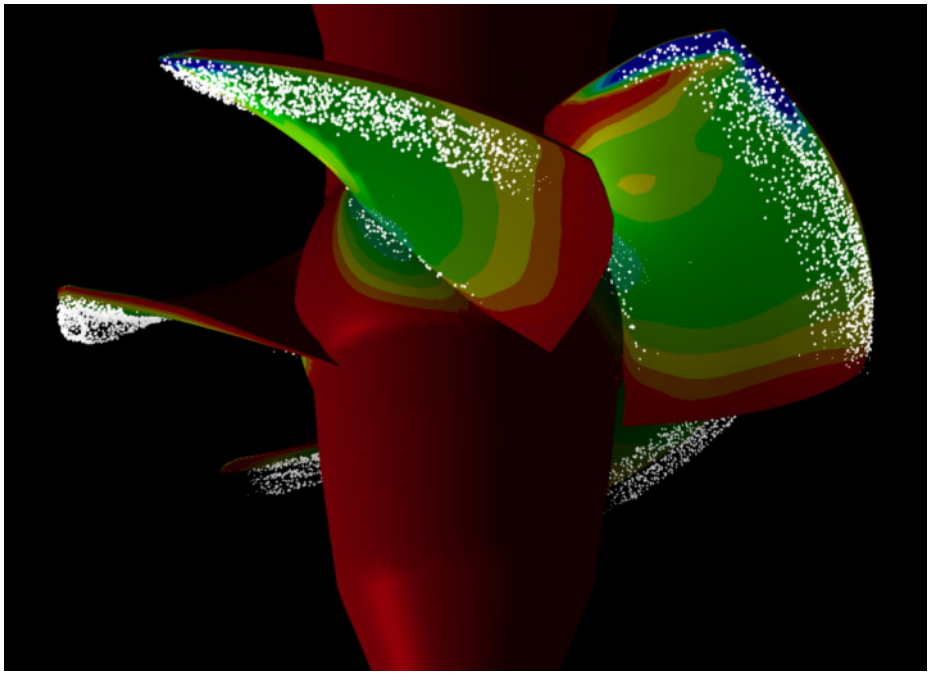


FIGURE 7.10 Cavitation bubbles near Kaplan runner blades, rendered as spheres (ROI was specified by low pressure). See also Colour Figure A.4 on page 124.

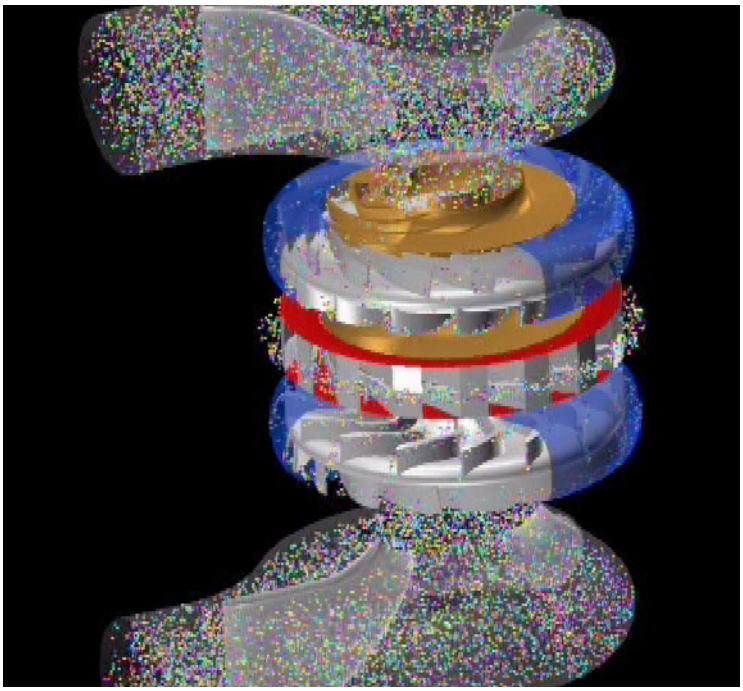


FIGURE 7.11 Particle stream through a storage pump, rendered as spheres (Runner case was opened in front for better insight). See also Colour Figure A.4 on page 124.

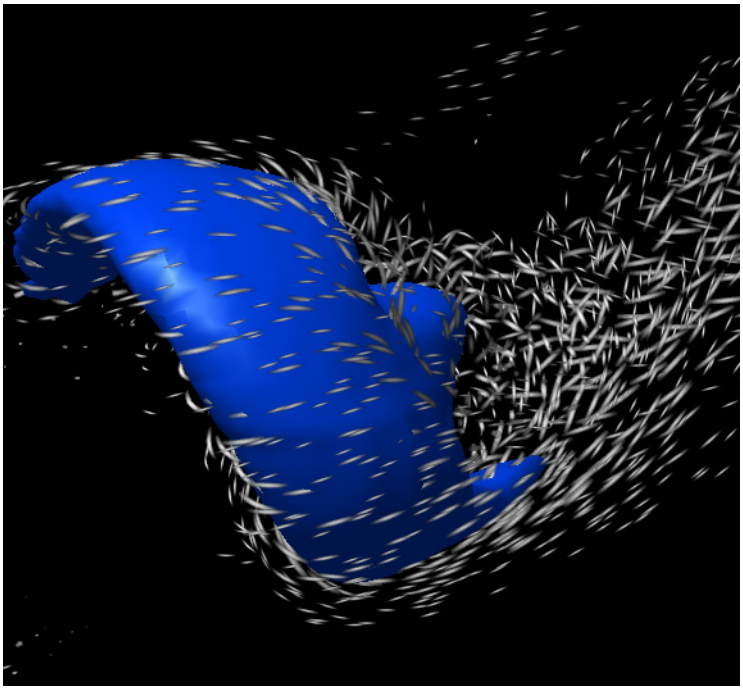


FIGURE 7.12 Vortex rope in Francis draft tube, rendered as streamlets (ROI is indicated by an isosurface of normalised helicity). See also Colour Figure A.4 on page 124.

CHAPTER 8

A VIRTUAL REALITY APPLICATION

Virtual environments have become a significant field of research and application during the past years. Their enhanced input and output devices and methods facilitate the exploration of complex data and geometries. Concurrently, they help reduce a variety of problems related to spatial 3D perception, such as cluttering and occlusion in complex geometrical scenes.

The vortex core line extraction and vortex hull construction described in Chapter 5 and Chapter 6 is a time-consuming process, taking up to one minute for a single timeframe. The user is thus forced to wait after every change of the input parameters until the new results are available and visible on the screen. To overcome this burden and facilitate an *interactive* use of the vortex application, a new system architecture is necessary.

In this chapter, we will present a framework for transferring results of the vortex core extraction and vortex hull construction to a virtual environment. We use a file system based approach to decouple the rendering phase from the computationally expensive data acquisition and feature extraction procedures. As a consequence of this, the VR application is able to efficiently load and handle the vortex data. The user can thus navigate through the preprocessed timeframes at comfortable frame rates and interactively transform, select, mark and remove vortex structures of the graphical scene.

The remainder of this chapter is organised as follows: In Section 8.1, we will give a brief introduction to the visualisation and programming environment that we used for our implementations. After a short comparison of the two visualisation platforms AVS and COVISE, we will motivate the choice of COVISE for the later stages of this thesis, and present its two basic VR concepts: a special renderer COVER and user-programmable plugins for extending its functionality. We will then in Section 8.2 demonstrate how our framework is built from three major components based upon these ingredients, and describe the specific features provided by the new VR application for vortices. At the end of this chapter, we will give some performance measurement results as well as result images (Section 8.3).

8.1 VISUALISATION SYSTEMS AND VIRTUAL ENVIRONMENTS

The algorithms described in this thesis were implemented for two visualisation platforms. The first one was AVS *(*Advanced Visualisation System*)*, which is available in two versions, AVS 5 and AVS Express. We preferred AVS 5 since there was already a vast amount of modules available which had been previously implemented for this system. The second platform was COVISE (Collaborative Visualisation and Simulation Environment), which was developed at the High Performance Computing Center (HLRS) of the University of Stuttgart [HLR04] and afterwards distributed by the spin-off company VisEnSo [VIS04]. There has been a cooperation between HLRS/VisEnSo and the ETH Zurich regarding the COVISE system for several years. Additionally, our industry partner VA Tech Hydro is also working with COVISE, and is using it in conjunction with VR devices like the mobile Cykloop (see Figure 8.3 right).

Both visualisation platforms support the *graphical programming* paradigm, i.e. they offer a graphical user interface which facilitates the construction of major visualisation applications by composing them of minor units. Every application is stored as a *network* of individual programs, which are called *modules*. The modules possess input and output ports for a variety of data types like computational grids, vector/scalar fields and images. By connecting the module ports using pipes, the dataflow through the network is controlled and major applications can be built from minor ones, similar as if using a construction kit (see Figure 8.1).

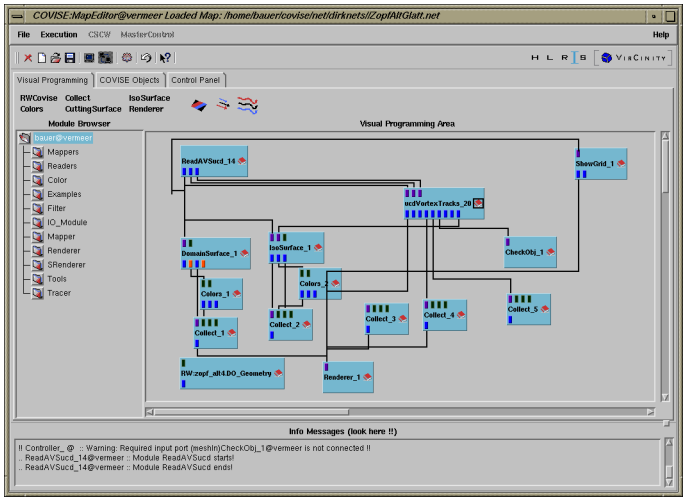


FIGURE 8.1 A typical COVISE network.

In both systems, a certain set of standard modules is available for general tasks, like readers for the input of computational grids and data fields, mappers and filters for intermediate processing of the data, and renderers for the output of the resulting geometry. In addition to the standard modules, it is also possible to enhance the systems with arbitrary user-programmed modules, which is in fact the actual strength of these visualisation systems. An essential difference between AVS and COVISE is, however, that COVISE offers *two* different modules for geometry output: a conventional Open Inventor based *Renderer* (see Figure 8.2) as well as a special Performer based renderer COVER for support of virtual reality applications (see Figure 8.4 left).

The conventional renderer is mainly used for single-workplace applications and *collaborative working* both using standard monitor output, whereas COVER also supports graphical presentation in *virtual environments* like caves (Figure 8.4 left), projection screens (Figure 8.3 left), and portable devices (such as the Cykloop developed by VisEnSo, Figure 8.3 right). To profit by the enhanced rendering and VR capabilities and apply them to the flow visualisation problems described in the previous chapters, we used COVISE as a second implementation platform during the later stages of this thesis.

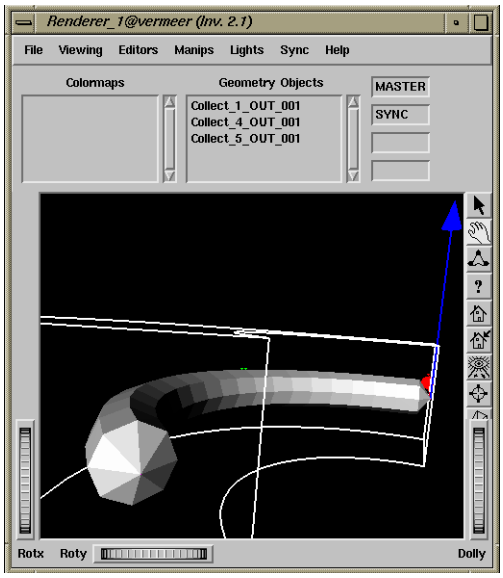


FIGURE 8.2 The Inventor-based COVISE standard renderer.

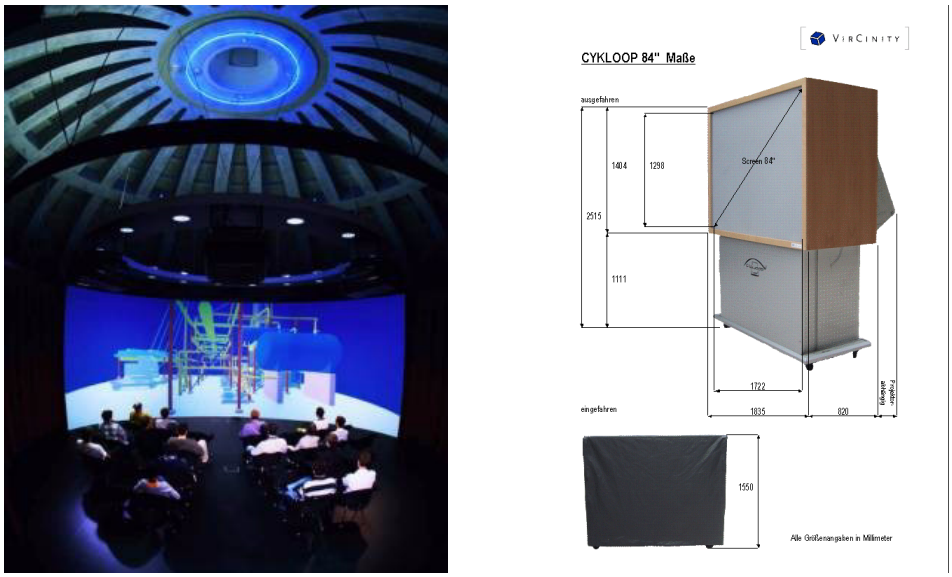


FIGURE 8.3 Typical VR environments. Left: large projection screen.
Right: the mobile Cykloop device (image courtesy of VisEnSo GmbH, Stuttgart).

We will now briefly present the basic functionality of the VR renderer COVER, which we used as a component for our interactive vortex visualisation application described in Section 8.2. COVER (**CO**vise **V**irtual **E**nvironment **R**enderer) is a special OpenGL Performer based renderer for virtual environments and their peripheral devices such as head trackers and 3D mouses. It also supports virtual devices, e.g. a lightsaber for selecting menu entries using the 3D mouse (see Figure 8.4, left). Its functionality can be extended by programming additional *plugins* [VIS03]. Besides projecting the graphical scene to a VR screen like a *cave* or *workbench*, a conventional monitor workplace is also supported, which is especially useful for testing purposes during the implementation phase. In this case, COVER renders the graphical output into a special window viewport of the desktop on a standard monitor and accepts conventional 2D mouse input as a substitute for 3D mouse input. Of course a 3D mouse position in virtual space has to be simulated from the 2D coordinates then. Although the user gets no real 3D perception of the scene in this case, it is still possible to grab and manipulate the scene objects using a simulated lightsaber as in the virtual environment case.

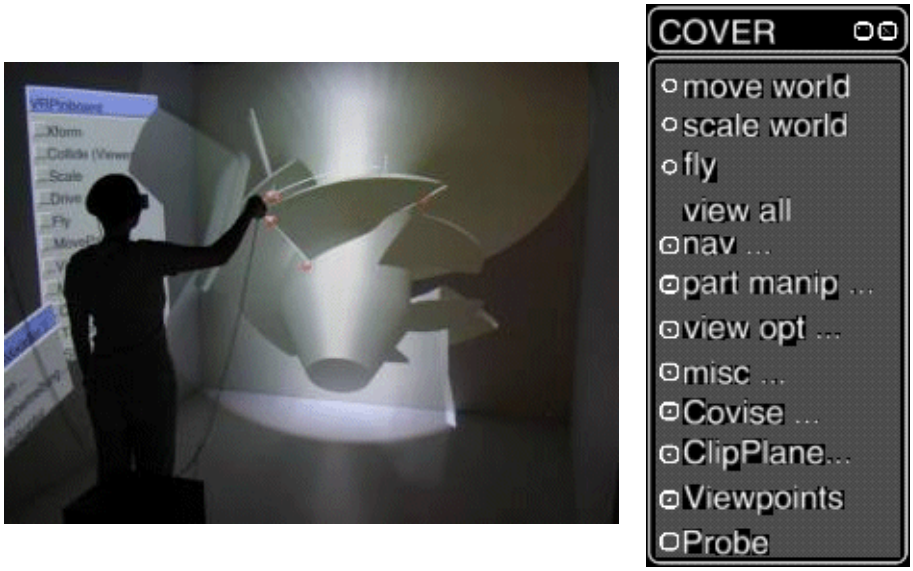


FIGURE 8.4 Left: A graphical scene projected from the COVER renderer into a cave (image courtesy of VisEnSo GmbH, Stuttgart. See also Colour Figure A.3 on page 123). Right: The standard main menu of the COVER renderer.

The basic COVER module, when integrated into a COVISE network, provides a main menu which offers fundamental graphical operations to the user. Figure 8.4 (right) shows the main menu of the COVER renderer as displayed on the screen. The *move world* option enables the user to pick up the whole graphical scene and to translate and rotate it on the screen. The *scale world* option allows for zooming into and out of the scene, and thus for inspecting its graphical objects in more detail. Graphical operations can be restricted to certain parts of the scene by choosing the *part manip* option. Furthermore, a *fly* option is available which starts a flight through the scene to a target position aimed to by the 3D input device. Selecting the *"Covise..."* entry opens a second window where the user can start a custom VR application and its appropriate COVER plugin. An additional menu entry showing the custom application name is then added to the COVER main menu.

8.2 THE VR APPLICATION FOR VORTEX VISUALISATION

Based on the concepts presented in the previous sections, we implemented a framework for the COVISE and COVER platform, which allows for *interactively* visualising basic results of the feature extraction methods described in Chapter 5 and Chapter 6, namely vortex core lines and vortex hulls.

8.2.1 System Overview

Figure 8.5 shows the structure of the overall system, which basically consists of three modules: a *feature extraction* module, a COVER plugin and a *feature processing* module. The feature extraction module is part of a COVISE network and contains the algorithms known from Chapter 5 and Chapter 6, namely for extracting vortex core lines and constructing vortex hulls. In a preprocessing step, this module extracts features from time-dependent CFD data by looping through a *timeframe directory* in a *batch mode*. For every timeframe in the directory, the vector and scalar fields are set up and the vortex core lines are extracted. The surrounding vortex hulls can also be computed and smoothed. Finally, the resulting geometry is exported to a result file on disk (see Figure 8.5, left). This procedure takes about 1 minute per frame and is repeated for every timeframe of the directory.

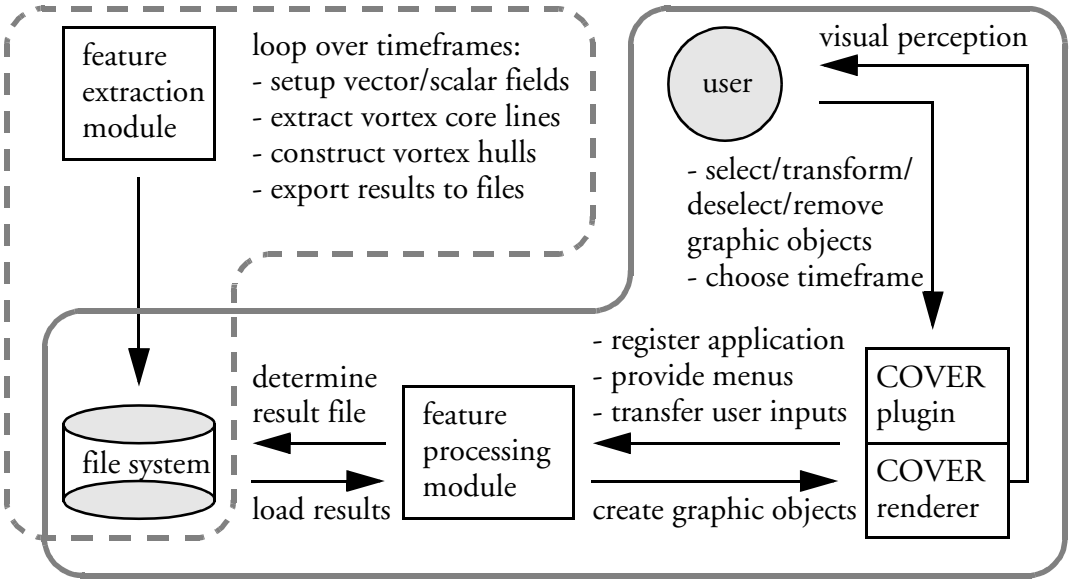


FIGURE 8.5 System architecture for the VR vortex visualisation framework. Dashed lines: batch mode network. Solid lines: interactive mode network.

Once the vortex results have been stored to disk, another COVISE network for *interactive* use in a VR environment can be started, which contains an instance of the COVER renderer and the feature processing module. The COVER renderer has been enhanced by a COVER plugin, which adds an additional entry to the COVER main menu and registers the feature processing module to COVER (see Figure 8.5, right). From now on, the plugin serves as an interface between the user, the COVER renderer and the feature processing module. It contains callback functions for polling the user input devices and triggers the renderer for graphical output of the scene. Also, it provides *picking* functionality, which we developed as a functional extension to COVER and which enables the user to select

the vortex which is nearest to the 3D mouse position in virtual space. We will in the following sections describe the three major components of the overall system, their functionality and their features in more detail.

8.2.2 The feature extraction module

The batch mode network (Figure 8.5 left) contains the feature extraction module *ucdVortexTracks* for vortex core line extraction, vortex tracking (not used here), and vortex hull construction, thus comprising the implementations of all visualisation methods presented in Chapter 5 and Chapter 6 of this thesis. The vortices are extracted in a batch mode, looping through all time steps of a certain flow simulation, vortex extraction method and scale step, and writing the results into a corresponding result directory. A single result file contains the number of vortex core lines found for this time step, and for every vortex core line:

- the number of vertices on the core line,
- the number of triangles building its vortex hull,
- for every vertex: the (x, y, z) coordinates of the vertex,
- for every triangle: the (x, y, z) coordinates of its three corners.

The result files are hierarchically organised in the file system, which means that the flow simulation, vortex method and scale step each define a level of the directory tree. Within a certain directory, the result files are indexed in ascending order due to their time step within the flow simulation. This hierarchical directory structure allows for easily building the correct result file name from the user inputs by converting the selected menu entries to strings and concatenating them to the required file name (see also Section 8.2.4).

8.2.3 The COVER plugin

The COVER plugin *ucdVortexTracksPlugin* communicates with the feature processing module via a *message passing* mechanism. It not only transfers the current settings and any update of the widgets, but also implements a newly developed *picking* mode, which takes the current 3D mouse position in virtual space and loops through every vortex core line segment. The core line segment which lies closest to the current mouse position defines the selected vortex core line and hull, which is then highlighted on the screen (see Figure 8.6). Several vortices can be selected and grouped together, deselected or removed from the screen to allow better insight into complex geometrical scenes.

8.2.4 The feature processing module

When the feature processing module *ucdVortexTracksCOVER* is started and registered to the COVER renderer, an additional menu entry appears in the COVER main menu, showing the name of the feature processing module (see lower right part of Figure 8.6). A click on this menu entry opens a pop-up window entitled with the same name, which is the main menu of the VR application (see upper left part of Figure 8.6).

This main menu of the VR application contains widgets like radio buttons (e.g. for switching the input device between the *Transform Object* and *Select Object* mode) or sliders (like the *Scale, Time* and *Opacity* slider). Furthermore, additional submenus can be opened from this menu (e.g. the *Turbine Design* and *Extraction Method* submenus shown in the upper right part of Figure 8.6).

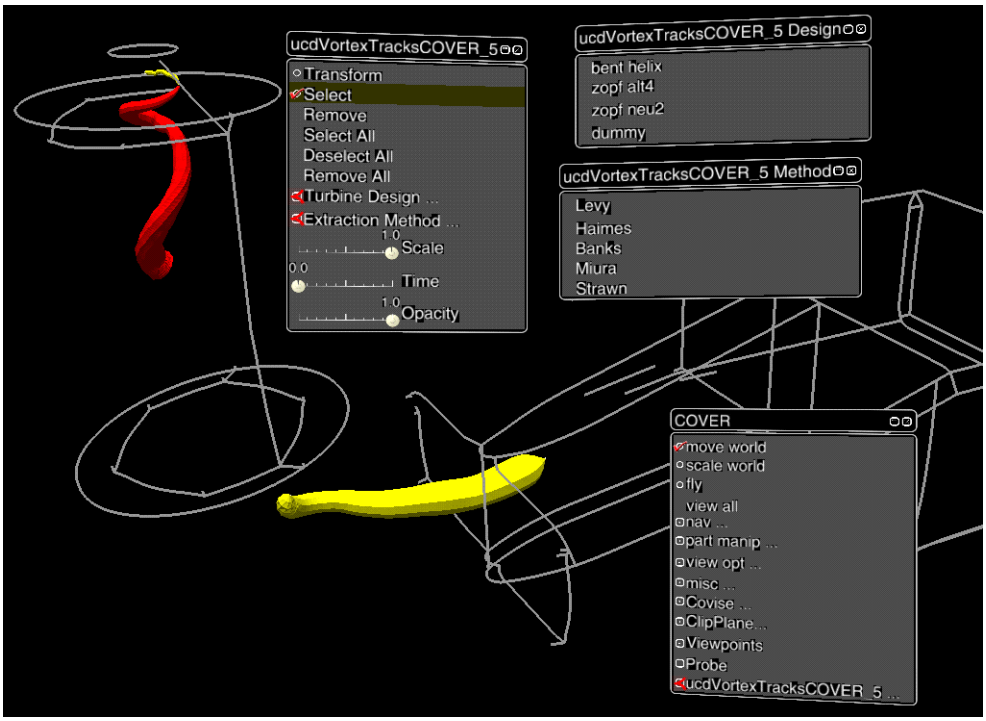


FIGURE 8.6 A typical scene showing COVER and VR application menus and vortex hulls. The twisted vortex in the upper part of the draft tube has been selected. See also Colour Figure A.3 on page 123.

After the VR application has been started, the user can choose one out of several flow simulations (or turbine designs, respectively) and vortex extraction methods from the multiple choice submenus mentioned above. Furthermore, he or she can set the scale and time sliders between a minimum and maximum value and this way choose an arbitrary degree of smoothing and time step for which vortex cores were extracted.

Based on these four user settings, the VR application determines the corresponding result file, loads the result data and transforms them into graphical objects for the COVER renderer. In practical experience, the sum of loading, rendering and user perception time is in the range of 1 second per timeframe (see performance results in Section 8.3). It is thus possible to navigate through the solution space of the vortex extraction and to interactively explore the behaviour of the detected flow features.

The *Transform* mode of the main menu allows for common geometrical transformations of the displayed vortices (like translation, rotation and scaling). Another option is the *Select* mode, which we implemented based upon the picking functionality described in Section 8.2.3. In this mode, the vortices can be selected and highlighted, deselected and removed individually or in groups (see Figure 8.6). This feature allows for better structuring complex geometrical scenes containing a larger number of vortices.

Besides the previous options, the user can continuously adjust the *opacity* of the vortex hulls using a slider widget similar to the scale and time slider. This feature allows for visualising a semi-transparent vortex hull together with the vortex core line at its center (see Figure 8.7 and Figure 8.8).

8.3 RESULTS

The VR application for vortex cores was tested on a 250 MHz SGI Octane in the desktop view mode (that means the user input was done by a conventional 2D mouse, the graphical output was rendered to a standard monitor screen). For three different turbine designs, we fixed a certain scale ($s = 10^{-6}$), vortex method (Levy), scalar field (vortex strength), and hull shape (10-sided polygons). The vortices of a whole unsteady CFD timeframe directory were then extracted and their hulls computed, the results for every time step written to disk.

In the VR application, the time slider was manually moved between its minimum and maximum value (= first and last frame of the unsteady CFD dataset) for measuring the performance. Concurrently, the number of thereby loaded and rendered timeframes was counted, as well as the number of triangles of the vortex hulls. To get a realistic measure for the interactive performance in practical use, the execution time was not measured as pure CPU time but for complete feedback cycles also including the perception and reaction time of the user. Table 8.1 shows

- the number of timeframes which were looped through,
- the number of vortex hull triangles summed over all frames,
- the amount of result file disk space summed over all frames,
- the time consumed for loading, rendering and perceiving the results of all frames,

and the corresponding numbers per second. From the table, it is evident that the application is interactive, but the frame rate is by one order of magnitude too low for a smooth animation of the complex geometry of the vortex hulls. Prefetching several timeframes into main memory could improve the performance of the system.

turbine design	number of frames	number of triangles	disk space [KB]	loading/ rendering time [s]	frame rate [frames/sec]	throughput [triangles/sec]	throughput [KB/sec]
bent helix	4	1520	56	1.0	4.0	1520	56
Francis original	26	130340	4756	20.0	1.3	6517	238
Francis modified	24	146460	5332	20.0	1.2	7323	267

TABLE 8.1 Performance analysis of the VR application for vortices.

Figure 8.7 and Figure 8.8 show some examples of vortex core lines and their surrounding vortex hulls in the COVER environment. The screenshots were taken for the original and modified draft tube design of the Francis turbine already treated in Chapter 5. Due to its improved design, the modified draft tube produces weaker vortices, leading to more fragmented and less articulate vortex structures (compare with result images of Section 5.5).

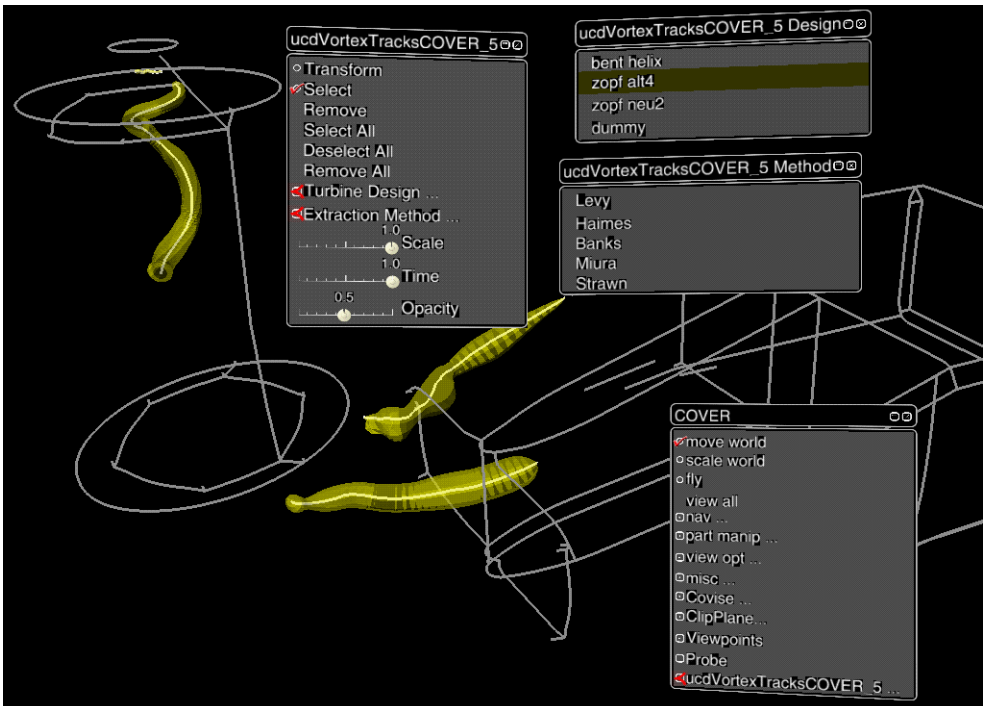


FIGURE 8.7 Vortex core lines and hulls of the original draft tube design. The hulls were computed for a vortex strength threshold of 1.0. See also Colour Figure A.3 on page 123.

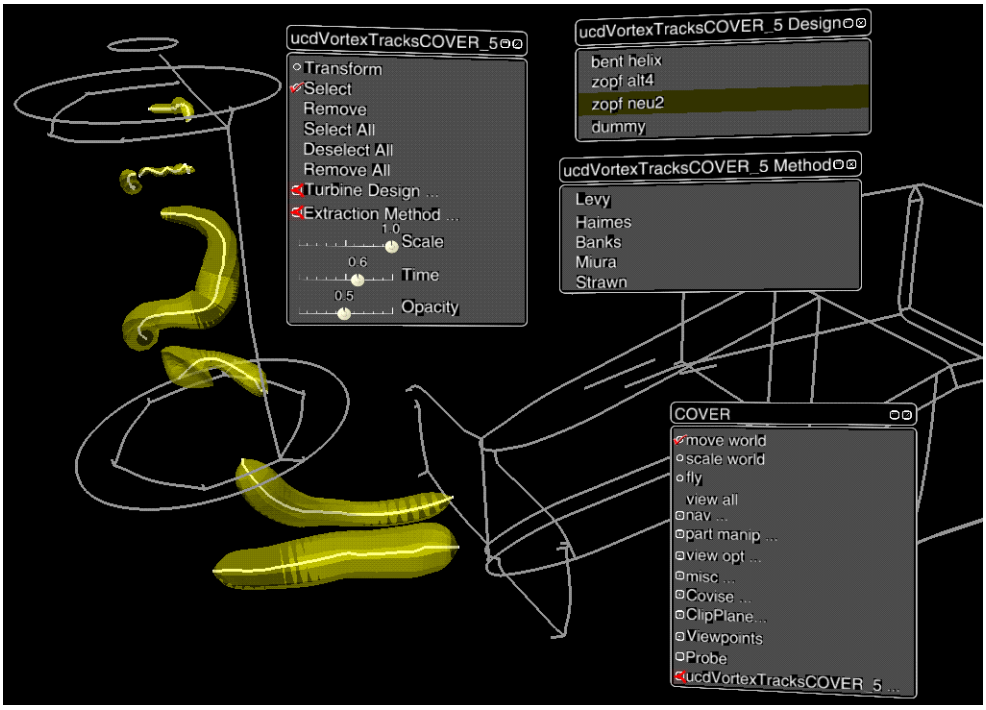


FIGURE 8.8 Vortex core lines and hulls of the modified draft tube design. The design optimisation led to fragmentation of the vortices. See also Colour Figure A.3 on page 123.

CHAPTER 9

CONCLUSIONS

This chapter concludes the dissertation with a summary of the main contributions, a discussion of advantages and drawbacks of the presented approaches, and some ideas for future research.

9.1 PRINCIPAL CONTRIBUTIONS

The main focus of this thesis is on the visualisation of steady and time-dependent flow structures by means of vortex core line extraction, vortex core line tracking, vortex hull construction and selective ROI-based visualisation of moving particles. In this context, the following contributions have been made to the field:

- **Scale-space techniques:**
  We have implemented a method for computing the linear scale-space of CFD data on unstructured grids. Rather than convolving the original data using Gaussian filters, we solved the diffusion equation to get the smoothed datasets. The diffusion equation was discretised using a finite element approach and solved using implicit Euler integration, where symmetric boundary conditions reduced the computational time.
- **Feature extraction and tracking:**
  We have implemented a modified version of the parallel vectors operator for feature extraction, namely the computation of vortex core lines in 3D space. By applying the precomputed scale-space and controlling the scale parameter, we are able to improve the extraction of features, in particular of features defined in terms of second spatial derivatives. We have also presented a novel 4D tracking method for line-type features, which allows for tracking vortex cores through different scales inherently carrying over correct topological information such as connectivity. The same algorithm can be used for temporal tracking, as is needed for selective visualisation of features in time-dependent data. Our implicit tracking method can reliably treat fast-moving features and is therefore a good alternative to proximity-based methods.

- **Vortex hulls:**
  We have refined and implemented an algorithm for constructing closed tubes surrounding the previously extracted vortex core lines in an unstructured grid. We use the deformable model paradigm to enhance the core lines by means of growing rays which lie in planes perpendicular to the core lines. By enhancing the connectivity data structure of the grid, we are able to migrate along each ray without global or local point searches, which significantly reduces the computational time. The shape of the resulting vortex hulls strongly depends on the choice of a scalar field and threshold. In case the scalar data are noisy, a median ray filter in combination with subsequent Laplacian mesh fairing can successfully smooth the vortex hulls to reduce visual artifacts.

- **Selective particle tracing:**
  We have developed a modified particle tracer for selectively visualising time-dependent 3D vector fields. Unlike previous representatives of its kind, our particle tracer is focused on special regions-of-interest in the flow, which can reduce the amount of data to be processed by one order of magnitude. By using quasi-random sequences when seeding new particles, a uniform particle density is maintained over time, and regular patterns and clusters are avoided. Due to the use of buffer cells and of two scalar thresholds for visibility, the particles fade in and out smoothly, which is favorable for animations. We applied our method to various datasets from our industry partners, visualising the vortex rope in the draft tube of a Francis turbine, as well as cavitation on the suction side of a Kaplan turbine, and the flow through a storage pump. The concept of selective visualisation of the relevant flow structures has proven to give additional insight into their complex dynamic behavior.

- **Virtual Reality Application for Flow Visualisation:**
  We have implemented a framework for transferring the results of the vortex extraction to a virtual environment. The VR environment allows for displaying and manipulating the graphical objects on a screen, in a cave, on a workbench or similar VR output devices, and to manipulate them using 3D input devices. By storing the vortex extraction results to the file system in a batch mode, the computationally expensive CFD timeframe retrieval and feature extraction is decoupled from the rendering process. The VR application thus allows the user to interactively navigate through the solution space of the vortex extraction, switching between the result timeframes for different turbine designs, vortex extraction methods, scale and time steps. In addition, distinct vortices and their hulls can be marked, selected, highlighted and removed using the 3D input device. The transparency of the rendered vortex hulls can be continuously adjusted, which permits insight to the vortex core lines at their center and improves their overall visual perception.

9.2 DISCUSSION AND FUTURE WORK

In this thesis, it has been demonstrated that feature-based, region-based and integration-based visualisation techniques can successfully be combined with scale-space and VR techniques to improve the extraction and visualisation of relevant flow features from time-dependent CFD datasets. We will list here some advantages and limitations of the presented approaches, as well as some possible directions for future research.

- The linear scale-space representation is a powerful instrument because it allows not only for smoothing the datasets but also to implicitly track features over different levels of scale or time. The vortex tracking algorithm inherently delivers the correct topology of line-type features, even when they are fast moving over cell boundaries between two successive time steps. This leads to more coherence of the features and to a reduction of heuristics and user inputs like thresholds. Since our tracking algorithm provides the basic information on bifurcations, future work could also solve the problem of *event detection*, which was not yet treated in this thesis.

- The drawback of the approach is its high computational effort. Smoothing large industrial CFD datasets requires discretising and solving the diffusion equation by numerical methods, which takes a lot of preprocessing time (see Section 5.5). Computing the scale-space for one timeframe is in the order of minutes, for an unsteady CFD dataset it is in the order of several hours. Once the scale-space representation is available, a feature extraction run for one timeframe still takes about one minute for a large grid with about 1 million nodes. Tracking over 100 time steps thus requires nearly two hours. Also, the amount of main memory needed for efficient working is an issue, since the industrial datasets are large and will rapidly increase also in the future. In summary, these techniques are well-suited for batch mode and hence for computing video animations. Interactive exploration of the solution space is still awkward. Overcoming this disadvantage was one motivation for developing the VR application of Chapter 8.

- The vortex hull concept combines the advantages of line-type features (clear separability) and surface-type features (better perception of vortex size and shape). However, the user needs a-priori knowledge for choosing a suitable scalar field and threshold, since the results of the vortex hull construction strongly depend on these. In the current implementation, a global scalar threshold is used for all vortex hulls. Since the scalar values on the core lines differ considerably, a scalar threshold per core line could improve the quality of the results. Relative rather than absolute threshold values are also thinkable, as suggested by Roth [Rot00].

- Often the scalar fields of the industrial datasets are noisy, so the computed vortex hulls need some postprocessing in terms of smoothing. A median filter was implemented to eliminate numerical exceptions affecting the cross-sections of the hulls. Furthermore, Laplacian fairing of the complete hull mesh helped improve the quality of the vortex hulls. However, alternative types of low-pass filters are also thinkable, like a binomial or discrete Gaussian filter for the rays. Future work could comprise a comparison of different such filters. Additionally, the scale-space representation of the underlying scalar fields could be exploited to reduce the noise in advance.

- Selective particle tracing is computationally more efficient than conventional versions since the region-of-interest reduces the amount of processed data to typically an order of magnitude. The particles are smoothly blended in and out due to the inner/outer/buffer cell classification, the Sobol' distribution and the two thresholds for visibility (see Chapter 7). These properties predestine the method for the production of video animations, especially when using streamlets as rendering primitives. However, the resulting images regarded as single stills are not as meaningful as those produced by some other techniques like streamlines, streaklines and path lines, which record the "history" of the particles.

- The VR application yields a significant efficiency and performance gain for exploring the results of the vortex extraction process. In contrast to the vortex extraction module, the response time to a user input (such as changing the time or scale of the underlying timeframe, or the extraction method) decreases from about 1 minute to about 1 second, which permits an interactive use of the tool. However, loading the results from the file system is still a bottleneck, which leads to popping effects when trying to animate the features by fast changing user inputs (e.g. slider movements). Future work should address this issue, for instance by a technique for prefetching several result timeframes into main memory.
- Finally, the current VR application only supports vortex core lines and vortex hulls. The framework, however, gives room to support more flow features, such as the feature meshes gained from vortex tracking (Chapter 5) and the positions from traced particles (Chapter 7). Also, an enhancement of the user interface by more interactive functionality could be investigated, such as the possibility to create streamlines at selected points near the vortex core lines.

APPENDIX A

COLOUR FIGURES

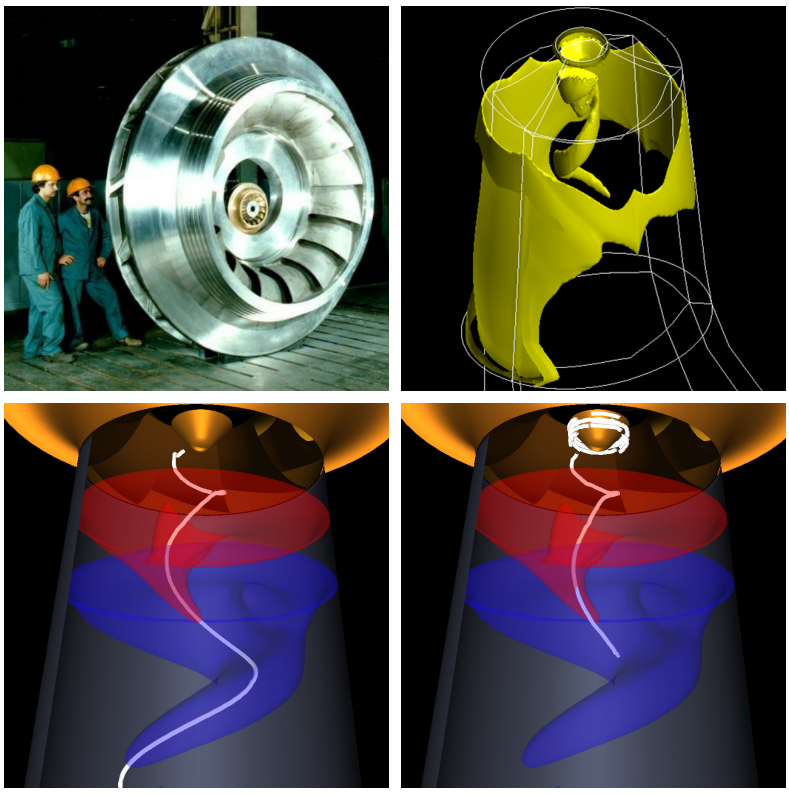


FIGURE A.1 **Vortex extraction** for a Francis turbine.

Top left: Francis runner and scaled model (Figure 2.1 on page 10).
Top right: vortex rope in draft tube visualised by isosurface (Figure 7.1 on page 97).
Bottom: vortex core lines due to Miura/Kida and Levy method (Figure 5.14 on page 77).

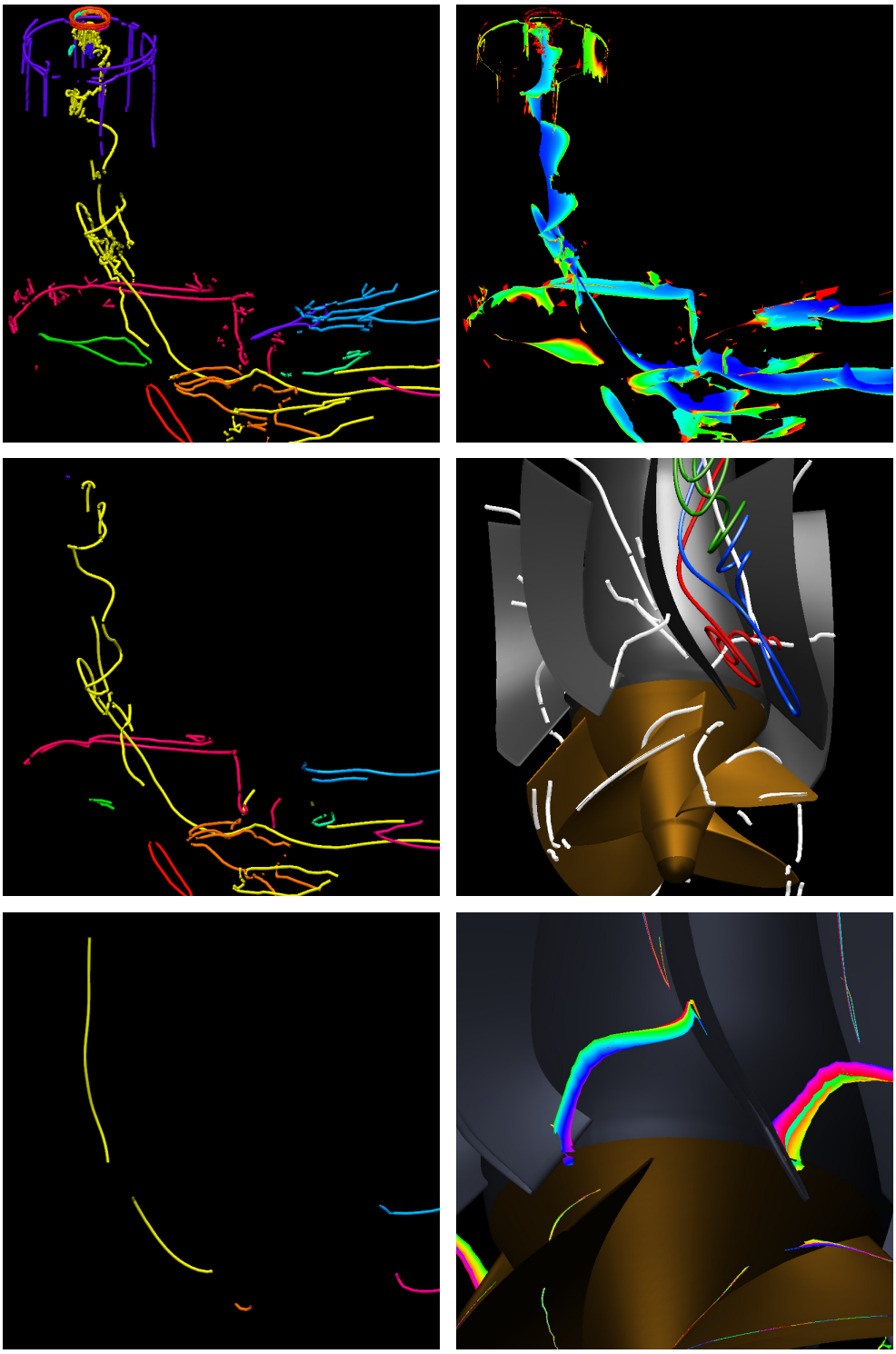


FIGURE A.2 **Vortex tracking** in scale and time.

Left column: vortex core lines for increasing scale level (Figure 5.15 on page 77).
Top right: surfaces swept by the vortex core lines shown in left column.
Center right: vortex core lines and streamlines in a mixed-flow pump.
Bottom right: vortices of mixed-flow pump tracked over time (Figure 5.16 on page 77).

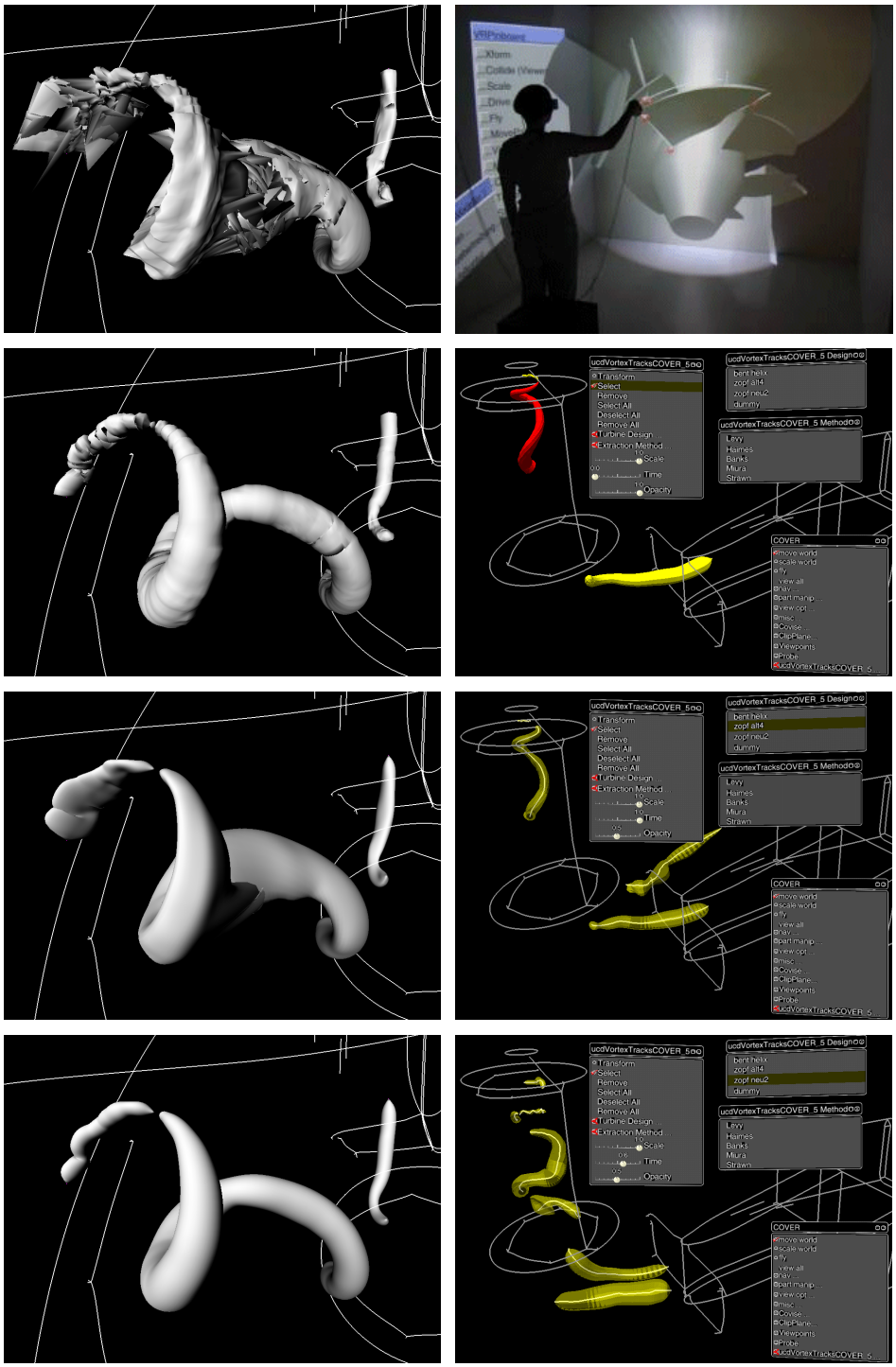


FIGURE A.3 **Vortex hulls** (left column) and **VR application** (right column).

Left: vortex hulls based on raw data (Figure 6.14 on page 92), after pure median filtering (Figure 6.15 on page 92), after pure Laplacian mesh fairing (Figure 6.16 on page 93), and after a combination of median and Laplacian smoothing (Figure 6.17 on page 93).

Right: COVER renderer (Figure 8.4 on page 110) and VR application showing vortex core lines and hulls (Figure 8.6 on page 113, Figure 8.7 and 8.8 on page 115).

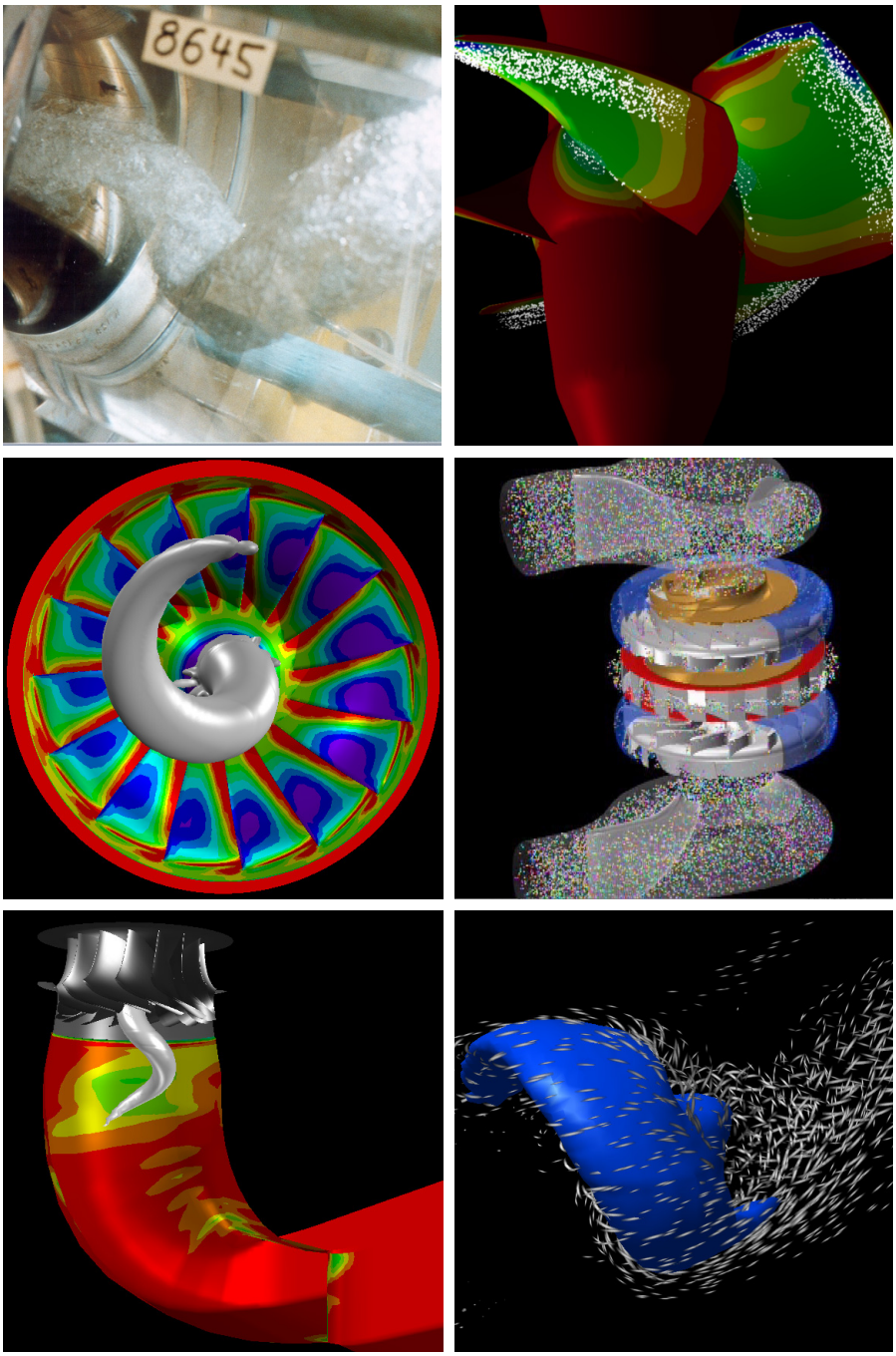


FIGURE A.4 **Industrial motivation** (left column) and **selective particle tracing** (right column).

Top left: cavitation bubbles on an industrial test rig (Figure 7.2 on page 98).
Center left: relative pressure variations in a Francis runner (Figure 7.3 on page 99).
Bottom left: relative pressure variations in a Francis draft tube (Figure 7.4 on page 99).

Top right: Cavitation bubbles at Kaplan runner blades (Figure 7.10 on page 104).
Center right: Particle stream through a storage pump (Figure 7.11 on page 105).
Bottom right: Streamlets and isosurface indicating the vortex rope
in a Francis draft tube (Figure 7.12 on page 105).

APPENDIX B

REFERENCES

References

[BCE92] K.W. Brodlie, L.A. Carpenter, and R.A. Earnshaw. *Scientific Visualization - Techniques and Applications*. Springer-Verlag, 1992.

[BP02a] D. Bauer and R. Peikert. A case study in selective visualization of unsteady 3d flow. In *Proceedings of IEEE Visualization 02*, pages 525–528, Oct 2002.

[BP02b] D. Bauer and R. Peikert. Vortex tracking in scale-space. In I. Navazo (Editors) D. Ebert, P. Brunet, editor, *Data Visualization 2002. Proceedings of VisSym 02*, pages 233–240, May 2002.

[BS94] D. Banks and B. Singer. Vortex tubes in turbulent flows: Identification, representation, reconstruction. In *Proceedings of IEEE Visualization 94*, pages 132–139, Oct 1994.

[BSH97] H. Battke, D. Stalling, and H.C. Hege. Fast line integral convolution for arbitrary surfaces in 3d. *Visualization and Mathematics*, pages 181–195, 1997.

[Bun89] P. Buning. Numerical algorithms in cfd post-processing. In von Karman Institute for Fluid Dynamics Lecture Series 1989-07, editor, *Computer Graphics and Flow Visualization in Computational Fluid Dynamics*, Sep 1989.

[Bur81] P.J. Burt. Fast filter transforms for image processing. In *Computer Vision, Graphics, and Image Processing*, volume 16, pages 20–51, 1981.

[CC78] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[CK00] G.-H. Cottet and P.D. Koumoutsakos. *Vortex Methods: Theory and Practice*. Cambridge University Press, 2000.

[CL93] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 1993, Computer Graphics Proceedings, Annual Conference Series*, pages 263–270, Aug 1993.

[CPC90] M.S. Chong, A.E. Perry, and B.J. Cantwell. A general classification of three-dimensional flow fields. *Phys. Fluids A*, 2(5):765–777, May 1990.

[Cro81] J.L. Crowley. *A Representation for Visual Information*. PhD thesis, Carnegie-Mellon University, Robotics Institute, Pittsburg, Pennsylvania, 1981.

[Dau88] I. Daubechies. Orthonormal bases of compactly supported wavelets. In *Comm. on Pure and Applied Mathematics*, volume XLI, pages 909–996, 1988.

[dLvL99] W.C. de Leeuw and R. van Liere. Visualization of global flow structures using multiple levels of topology. In *Data Visualization 1999. Proceedings of VisSym 99*, pages 45–52, May 1999.

[DMSB99] M. Desbrun, M. Meyer, P. Schroeder, and A. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 1999, Computer Graphics Proceedings, Annual Conference Series*, pages 317–324, Aug 1999.

[DPR00] U. Diewald, T. Preusser, and M. Rumpf. Anisotropic diffusion in vector field visualization on euclidean domains and surfaces. *IEEE Transactions on Visualization and Computer Graphics*, pages 139–149, Apr-Jun 2000.

[ESK96] J. Encarnacao, W. Strasser, and R. Klein. *Graphische Datenverarbeitung 1 - Ger tetechnik, Programmierung und Anwendung graphischer Systeme*. R. Oldenbourg Verlag Muenchen Wien, 1996.

[EW92] R.A. Earnshaw and N. Wiseman. *An Introductory Guide to Scientific Visualization*. Springer-Verlag, 1992.

[FtHRKV92] L.M.J. Florack, B.M. ter Haar Romeny, J.J. Koenderink, and M.A. Viergever. Scale and the differential structure of images. *Image and Vision Computing*, 10:376–388, 1992.

[FWCS97] D. Feng, X. Wang, W. Cai, and J. Shi. A mass conservative flow field visualization method. *Computers & Graphics*, 21(6):749–756, Nov 1997.

[Gar90] M.P. Garrity. Raytracing irregular volume data. In *Proceedings of the 1990 San Diego Workshop on Volume Visualization*, pages 35–40, Nov 1990.

[GM77] R.A. Gingold and J.J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, Nov 1977.

[GSS99] I. Guskov, W. Sweldens, and P. Schroeder. Multiresolution signal processing for meshes. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 1999, Computer Graphics Proceedings, Annual Conference Series*, pages 325–334, Aug 1999.

[GTS+04] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface techniques for vortex visualization. In *Data Visualization 2004. Proceedings of VisSym 04*, pages 155–164, May 2004.

[Hac85] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer Verlag (New York), 1985.

[HH89] J.L. Helman and L. Hesselink. Representation and display of vector field topology in fluid flow datasets. *IEEE Computer*, 22(8):27–36, Aug 1989.

[HH91] J.L. Helman and L. Hesselink. Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11(3):36–46, May 1991.

[HLR04] HLRS. *Covise website of High Performance Computing Center Stuttgart (HLRS)*. http://www.hlrs.de/organization/vis/covise, 2004.

[HM90] R.B. Haber and D.A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in Scientific Computing 1990*, pages 74–93, 1990.

[HPvW94] L. Hesselink, F.H. Post, and J. van Wijk. Research issues in vector and tensor field visualization. *IEEE CG&A*, 14(2):76–79, Mar 1994.

[Hul90] J.P.M. Hultquist. Interactive numerical flow visualization using stream surfaces. *Computing Systems in Engineering*, 1(2-4):349–353, 1990.

[Hul92] J.P.M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of IEEE Visualization 92*, pages 171–178, Oct 1992.

[IG97] V. Interrante and C. Grosch. Strategies for effectively visualizing a 3d flow using volume line integral convolution. *Proceedings of IEEE Visualization 97*, pages 421–424, Oct 1997.

[IG98] V. Interrante and C. Grosch. Visualizing 3d flow. *IEEE Computer Graphics and Applications*, 18(4):49–53, 1998.

[JEH01] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-eulerian advection for unsteady flow visualization. In *Proceedings of IEEE Visualization 01*, pages 53–60, Oct 2001.

[JH95] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[JL97] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings of the Eurographics Workshop on Visualization in Scientific Computing*, pages 45–55, 1997.

[Ken98] D. Kenwright. Automatic detection of open and closed separation and attachment lines. *Proceedings of IEEE Visualization 98*, pages 151–158, Oct 1998.

[KH91] R.V. Klassen and S.J. Harrington. Shadowed hedgehogs: A technique for visualizing 2d slices of 3d vector fields. In *Proceedings of IEEE Visualization 91*, pages 148–153, 1991.

[Kli71] A. Klinger. Pattern and search statistics. In *Optimizing Methods in Statistics*. J.S. Rustagi (ed.), Academic Press (New York), 1971.

[Koe84] J. J. Koenderink. The structure of images. *Biological Cybernetics*, 50:363–370, 1984.

[LC87] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):163–169, Jul 1987.

[LDS90] Y. Levy, D. Degani, and A. Seginer. Graphical visualization of vortical flows by means of helicity. *AIAA*, 28(8):1347–1352, Aug 1990.

[LGE97] C. Luerig, R. Grosso, and T. Ertl. Combining wavelet transform and graph theory. *Visualization in Scientific Computing 1997*, pages 137–144, 1997.

[LHD+04] R.S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F.H. Post, and D. Weiskopf. The state of the art in flow visualisation: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2):203–221, May 2004.

[Lin94] T. Lindeberg. *Scale-Space Theory in Computer Vision*. Kluwer Academic Publishers (Boston/London/Dordrecht), 1994.

[Lip98] L. Lippert. *Wavelet-Based Volume Rendering*, volume Selected Readings in Vision and Graphics (Volume 9), Diss ETH No. 12612. PhD thesis, ETH Zurich, Department of Computer Science, Computer Graphics Laboratory, Zurich, Switzerland, 1998.

[MCS02] T. Maekelae, P. Clarysse, and O. Sipilae. A review of cardiac image registration methods. *IEEE Transactions on Medical Imaging*, 21(9):1011–1021, September 2002.

[MDB87] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6):1–14, Nov 1987.

[Mic94] C.A. Michelli. *Mathematical Aspects of Geometric Modeling*. SIAM/CBMS-NSF (Regional Conference Series in Applied Mathematics 65), 1994.

[MK96] H. Miura and S. Kida. Identification of central lines of swirling motion in turbulence. In *Proceedings of International Conference on Plasma Physics, Nagoya, Japan*, pages 866–869, 1996.

[MK97a] H. Miura and S. Kida. Identification and analysis of vortical structures. *NIFS-520 Research Report*, Nov 1997.

[MK97b] H. Miura and S. Kida. Identification of tubular vortices in turbulence. *Journal of the Physical Society of Japan*, 66(5):1331–1334, May 1997.

[MK98a] H. Miura and S. Kida. Dynamics of low-pressure swirling vortices in turbulence. *Advances in Turbulence VII*, pages 347–348, 1998.

[MK98b] H. Miura and S. Kida. Identification and analysis of vortical structures. *European Journal of Mechanics B/Fluids*, 17(4):471–488, 1998.

[MK98c] H. Miura and S. Kida. Swirl condition in low-pressure vortices. *Journal of the Physical Society of Japan*, 67(7):2166–2169, July 1998.

[NAG01] NAG. *NAG Fortran Library. Chapter F11: Sparse Linear Algebra*. The Numerical Algorithms Group Ltd (http://www.nag.com/numeric/fl/manual20/html/toc/f11.html), 2001.

[NHM97] G.M. Nielson, H. Hagen, and H. Mueller. *Scientific Visualization - Overviews, Methodologies, Techniques*. IEEE Computer Society Press, 1997.

[NSR90] G.M. Nielson, B. Shriver, and L.J. Rosenblum. *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990.

[Pau03] M. Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*, volume Selected Readings in Vision and Graphics (Volume 23), Diss ETH No. 15134. PhD thesis, ETH Zurich, Department of Computer Science, Computer Graphics Laboratory, Zurich, Switzerland, 2003.

[PC87] A.E. Perry and M.S. Chong. A description of eddying motions and flow patterns using critical-point concepts. *Ann. Rev. Fluid Mech.*, 19:125–155, 1987.

[PC92] A.E. Perry and M.S. Chong. Topology of flow patterns in vortex motions and turbulence. In J.P. Bonnet and Eds. Kluwer Academic Publishers M.N. Glauser, editors, *Eddy Structure Identification in Free Turbulent Shear Flows (IUTAM Symposium, Poitiers, France)*, Oct 1992.

[Pei03] R. Peikert. *Purely Linear 2D Vector Fields*. http://graphics.ethz.ch/ peikert/personal/Linear2D, 2003.

[PG01] M. Pauly and M. Gross. Spectral processing of point-sampled geometry. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series*, pages 379–386, Aug 2001.

[PGK02] M. Pauly, M. Gross, and L. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of Visualization 2002*, pages 163–170, Oct 2002.

[PKG03] M. Pauly, R. Keiser, and M. Gross. Multi-scale feature extraction on point-sampled models. In Computer Graphics Forum, editor, *Proceedings of the 24th Annual Conference of the European Association of Computer Graphics (Eurographics)*, 2003.

[PKKG03] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 2003, Computer Graphics Proceedings, Annual Conference Series*, Aug 2003.

[PM87] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Computer Society Workshop on Computer Vision (Miami, FL)*, pages 16–22, 1987.

[PVH+03] F.H. Post, B. Vrolijk, H. Hauser, R.S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):773–790, Dec 2003.

[PvW94] F.H. Post and J.J. van Wijk. Visual representation of vector fields: Recent developments and research directions. *Chapter 23 in: Scientific Visualization - Advances and Challenges*, pages 367–390, 1994.

[PZK+03] M. Pauly, M. Zwicker, O. Knoll, T. Weyrich, R. Keiser, and M. Gross. *PointShop3D website*. http://graphics.ethz.ch/pointshop3d, 2003.

[Rei01] F. Reinders. *Feature-Based Visualization of Time-Dependent Data*. PhD thesis, Technical University of Delft, Netherlands, Mar 2001.

[Rot00] M. Roth. *Automatic Extraction of Vortex Core Lines and Other Line-Type Features for Scientic Visualization*, volume Selected Readings in Vision and Graphics (Volume 9), Diss ETH No. 13673. PhD thesis, ETH Zurich, Department of Computer Science, Computer Graphics Laboratory, Zurich, Switzerland, 2000.

[RP96] M. Roth and R. Peikert. Flow visualization for turbomachinery design. In *Proceedings of IEEE Visualization 96*, pages 381–384, Oct 1996.

[RP98] M. Roth and R. Peikert. A higher-order method for finding vortex core lines. In *Proceedings of IEEE Visualization 98*, pages 143–150, Oct 1998.

[RP99] M. Roth and R. Peikert. The parallel vectors operator - a vector field visualization primitive. In *Proceedings of IEEE Visualization 99*, pages 261–268, Oct 1999.

[RPS99] F. Reinders, F.H. Post, and H.J.W. Spoelder. Attribute-based feature tracking. In Springer Verlag, editor, *Data Visualization 99 (VisSym 99 Proceedings)*, pages 63–72, 1999.

[RPS01] F. Reinders, F.H. Post, and H.J.W. Spoelder. Visualization of time-dependent data using feature tracking and event detection. *The Visual Computer*, 17(1):55–71, Feb 2001.

[Sad99] A. Sadarjoen. *Extraction and Visualization of Geometries in Fluid Flow Fields*. PhD thesis, TU Delft, the Netherlands, 1999.

[SB94] B. Singer and D. Banks. A predictor-corrector scheme for vortex identification. *NASA Contractor Report 194882, ICASE Report No. 94-11*, Mar 1994.

[SC92] J. Soria and B.J. Cantwell. Identification and classification of topological structures in free shear flows. In J.P. Bonnet and Eds. Kluwer Academic Publishers M.N. Glauser, editors, *Eddy Structure Identification in Free Turbulent Shear Flows (IUTAM Symposium, Poitiers, France)*, Oct 1992.

[Sch97] H.R. Schwarz. *Numerische Mathematik*. B.G. Teubner Stuttgart, 1997.

[SH95a] D. Stalling and H.C. Hege. Fast and resolution independent line integral convolution. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 1995, Computer Graphics Proceedings, Annual Conference Series*, pages 249–256, Aug 1995.

[SH95b] D. Sujudi and R. Haimes. Identification of swirling flow in 3d vector fields. *Tech. Report, Dept. of Aeronautics and Astronautics, MIT, Cambridge, MA*, 1995.

[SH95c] D. Sujudi and R. Haimes. Identification of swirling flow in 3d vector fields. *AIAA Paper 95-1715, 12th AIAA CFD Conference, San Diego, CA*, Jun 1995.

[Sha49] C.E. Shannon. Communication in the presence of noise. *Proc. Institute of Radio Engineers (IRE)*, 37:10–21, 1949.

[SKA98] R.C. Strawn, D.N. Kenwright, and J. Ahmad. Computer visualization of vortex wake systems. In *American Helicopter Society 54th Annual Forum*, May 1998.

[SKA99] R.C. Strawn, D.N. Kenwright, and J. Ahmad. Computer visualization of vortex wake systems. *AIAA Journal*, 37(4):511–512, April 1999.

[SMA00] A. Sanna, B. Montrucchio, and R. Arina. Visualizing unsteady flows by adaptive streaklines. In *Proc. WSCG 2000, The 8-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media2000*, pages I84–I91, Feb 2000.

[SML03] W.J. Schroeder, K.M. Martin, and W.E. Lorensen. *The Visualization Toolkit (VTK) (3rd edition)*. Kitware, Inc., 2003.

[SP03] M. Sato and R. Peikert. Core-line-based vortex hulls in turbomachinery flows. *Journal of the Visualization Society of Japan (Utsunomiya Visualization Symposium, 2003)*, 23(2):151–154, 2003.

[SSC94] R. Samtaney, D. Silver, and J. Cao. Visualizing features and tracking their evolution. *IEEE Computer*, 27(7):20–27, July 1994.

[Sta01] O. Staadt. *Multiresolution Representation and Compression of Surfaces and Volumes*, volume Selected Readings in Vision and Graphics (Volume 12), Diss ETH No. 14013. PhD thesis, ETH Zurich, Department of Computer Science, Computer Graphics Laboratory, Zurich, Switzerland, 2001.

[Ste93] R. Steinbrecher. *Bildverarbeitung in der Praxis*. R. Oldenbourg Verlag GmbH, Muenchen, 1993.

[SvWHP97] A. Sadarjoen, T. van Walsum, A.J.S. Hin, and F.H. Post. Particle tracing algorithms for 3d curvilinear grids. In G.M. Nielson, H. Mueller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques, Chapter 14, IEEE Computer Science Press*, pages 311–335, 1997.

[SW96] D. Silver and X. Wang. Volume tracking. In *Proceedings of IEEE Visualization 96*, pages 157–164, Oct 1996.

[SW97] D. Silver and X. Wang. Tracking and visualizing turbulent 3d features. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), Jun 1997.

[SW99] D. Silver and X. Wang. Visualizing evolving scalar phenomena. *Future Generation Computer Systems*, 15(1):99–108, Feb 1999.

[SZF+91] D. Silver, N. Zabusky, V. Fernandez, M. Gao, and R. Samtaney. Ellipsoidal quantification of evolving phenomena. *Scientific Visualization of Physical Phenomena (MIT, Cambridge), N. Patrikalakis (ed.)*, pages 573–588, 1991.

[Tau95] G. Taubin. A signal processing approach to fair surface design. In ACM Press / ACM SIGGRAPH, editor, *SIGGRAPH 1995, Computer Graphics Proceedings, Annual Conference Series*, pages 351–358, Aug 1995.

[TB96] G. Turk and D.C. Banks. Image-guided streamline placement. In ACM SIGGRAPH, editor, *Computer Graphics Proceedings '96*, pages 453–460, 1996.

[Tec00] AEA Technology. *CFX-TASCflow Theory documentation, Chapter 4: Turbulence Closure Models*. Harwell, U.K., http://www.software.aeat.com/cfx/, 2000.

[TF88] D. Terzopoulos and K. Fleischer. Deformable models. *The Visual Computer*, 4:306–331, 1988.

[TP82] M. Tobak and D.J. Peake. Topology of 3d separated flow. *Ann. Rev. Fluid Mech.*, 14:61–85, 1982.

[TSH01] X. Tricoche, G. Scheuermann, and H. Hagen. Topology-based visualization of time-dependent 2d vector fields. In Springer Verlag, editor, *Data Visualization 2001 (VisSym 01 Proceedings)*, pages 117–126, 2001.

[TvW99] A. Telea and J. van Wijk. Simplified representation of vector fields. In *Proceedings of IEEE Visualization 99*, pages 35–42, Oct 1999.

[USM97] S. Ueng, C. Sikorski, and K. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE TVCG*, 3(4):370 ff., Oct 1997.

[VIS03] VISENSO. *Covise Tutorial (Users Guide and Programming Guide)*. Covise Version 5.3 CD-ROM, 2003.

[VIS04] VISENSO. *COVISE product website of VISENSO GmbH, Stuttgart*. http://www.covise.de/d+e/data0801/deutsch/products/d_fs_prod.html, 2004.

[WE97] R. Westermann and T. Ertl. A multiscale approach to integrated volume segmentation and rendering. *Computer Graphics Forum*, 16(3):117–127, Sep 1997.

[Wes01] R. Westermann. The rendering of unstructured grids revisited. In *Data Visualization 2001. Proceedings of VisSym 01*, pages 65–74, May 2001.

[Wit83] A.P. Witkin. Scale-space filtering. In *Proc. of 8th International Joint Conference on Artificial Intelligence, (Karlsruhe, Germany)*, pages 1019–1023, Aug 1983.

[WLG97] R. Wegenkittel, H. Loeffelmann, and E. Groeller. Visualizing the behavior of higher dimensional dynamical systems. In R. Yagel and H. Hagen, editors, *Proceedings of IEEE Visualization 97*, pages 119–125, 1997.

[WT83] A.P. Witkin and J.M. Tenenbaum. *On the Role of Structure in Vision*. Human and Machine Vision (J. Beck, B. Hope, A. Rosenfeld, eds.), New York, Academic Press, 1983.

[YMC00] C. Yang, T. Mitra, and T. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Proceedings of IEEE Visualization 00*, pages 101–108, Oct 2000.

[ZBP+91] N. Zabusky, O. Boratav, R. Pelz, M. Gao, D. Silver, and S. Cooper. Emergence of coherent patterns of vortex stretching during reconnection: A scattering paradigm. *Physical Review Letters*, 67(18):2469–2472, 1991.

[ZT00] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method - Volume 1 (The Basis)*. Butterworth-Heinemann, 5th edition, 2000.

APPENDIX C

IMPLEMENTATIONAL ASPECTS

This appendix contains some more detailed explanations on the implementation of the vortex hulls described in Chapter 6. The focus is on the problem of expanding a vortex hull, beginning at a given vortex core line and stepping outwards along rays in predefined planes. Our method of traversing the hexahedral grid cells along the rays requires intersecting the rays with cell faces. Due to the discrete nature of the underlying unstructured grids and due to the noise contained in some of the raw CFD datasets we used, a number of numerical problems had to be solved in addition to the theoretical procedure.

In Chapter 6, we explained how to construct a vortex hull from fans of rays originating at the central vortex core line. The pseudocode of this procedure *BuildVortexHulls()* is shown in Figure C.1. Furthermore, we proposed a scheme to step along a ray and traverse the cells of an unstructured grid until a given scalar threshold has been crossed. The pseudocode of this function *FindEndpointOfRay()* is shown in Figure C.2.

What remains to describe is the method for computing the intersection points of a ray with a cell face, which is in general a non-planar quadrangle in 3D-space. Of course this computation could be done in *physical* space, directly yielding the global (x, y, z) coordinates of the intersection points. However, we also need the local (r, s, t) coordinates of the intersection point w.r.t. the four corners of the quadrangle face, for two purposes:

1. Testing if a computed intersection point of the ray with the unbounded surface (containing the cell face) lies *inside* the quadrangle spanned by the face corners (or outside, which would make the intersection point irrelevant),
2. Evaluation of the *scalar field* at a computed intersection point from the values at the face corners (for we can determine whether the scalar threshold has been crossed).

To avoid a point search on the quadrangle face (requiring the solution of a nonlinear equation system, e.g. by Newton iterations), it is better to compute the intersection points in *computational* space, where all coordinates are local to the face corners. In Section C.1, we will describe the mathematical background of the ray/face intersection in more detail. In Section C.2, we will finally treat some numerical issues of the intersection computation.

```
procedure BuildVortexHulls()

for each vortex core line
   m := number of segments of this vortex core line

   // Construct the vortex hull:

   for segment i = 1 to m
      Determine the start cell (= cell containing the segment).
      Set up a plane perpendicular to the core line:

      Compute plane origin P (= midpoint of segment).
      Compute plane normal N (= difference of segment vertices).

      Set up a 2D plane coordinate system by seed point P.
      and two orthonormal basis vectors L and M.

      // Compute all ray endpoints and their lambda values:
      for ray j = 1 to n
         Compute ray angle phi[j] and ray direction D[j].
         lambda[j] := FindEndpointOfRay(P, D[j])
      next ray

      // Filter each ray (median filter, k-neighbourhood):
      for ray j = 1 to n
         Collect old lambda values in k-neighbourhood of ray j.
         Sort the (2k+1) lambda values in ascending order.
         Set new lambda[j] to median of the (2k+1) lambda values.
      next ray

      // Compute the ray endpoints using the ray equations:
      for ray j = 1 to n
         Q[i,j] := P + lambda[j] * D[j]
      next ray
   next segment

   // Assembly the vortex hull:

   for segment i = 1 to m-1
      // Build inner tube section between two neighbouring planes:
      for ray j = 1 to n
         Quadrangle = (Q[i,j], Q[i,j+1], Q[i+1,j], Q[i+1,j+1])
         StoreTriangle(Q[i,j], Q[i+1,j], Q[i+1,j+1])
         StoreTriangle(Q[i,j], Q[i+1,j+1], Q[i,j+1])
      next ray
   next segment

   // Build pyramids for first and last segment of the core line:
   for ray j = 1 to n
      StoreTriangle(v[0], Q[1,j], Q[1,j+1])
      StoreTriangle(v[m], Q[m,j], Q[m,j+1])
   next j
next vortex core line
```

FIGURE C.1 The enhanced vortex hull construction algorithm.

```
function FindEndpointOfRay(seedPoint, rayDirection)

// Initialisation: start at the seed point P
current[CellNr, Lambda, Scalar] := (seedCellNr, 0.0, seedScalar)

// Traverse the ray, away from the seed point

for step = 0 to MAXSTEP
   Collect coords and scalars of all 8 nodes of current cell

   // For each of the 6 side faces of the current cell,
   // compute all intersection points with the ray:
   nextFaceNr := VOID
   nextLambda := INFINITE

   for face i = 0 to 5
      Collect coords and scalars of all 4 nodes of face i

      // Find all (0,1 or 2) intersection points of ray and face i
      // (compute lambda and scalar values and inside face flag):
      RayCastQuadrangleFace(seedPoint, rayDirection,
                            &lambda[2], &scalar[2], &inside[2])

      // Select nearest intersection point and next cell:
      for solution k = 0 to 1
         if lambda[k] > currentLambda and lambda[k] < nextLambda
         if inside[k] = TRUE then
            nextFaceNr := GlobalFaceNumber[currentCellNr, i]
            nextCellNr := NeighbouringCell[currentCellNr, nextFaceNr]
            nextLambda := lambda[k]
            nextScalar := scalar[k]
         end if
      next solution k
   next face

   // If no intersection has been found on any face,
   // stop and return the old ray point:
   if (nextFaceNr = VOID) then return currentLambda

   // If grid boundary or maximum number of steps has been reached,
   // stop and return the new ray point:
   if (nextCellNr = VOID) or (step = MAXSTEP) then return nextLambda

   // If the scalar threshold has been crossed,
   // stop and return the interpolated crossing point:
   if thresholdCrossed(currentScalar, scalarThresh, nextScalar) then
      return Interpolation(scalarThresh,
              currentLambda, nextLambda, currentScalar, nextScalar)

   // Else switch to next cell and continue with next step
   current[CellNr, Lambda, Scalar] := next[CellNr, Lambda, Scalar]
next step
```

FIGURE C.2 The algorithm for finding the endpoint of a ray.

C.1 INTERSECTIONS OF RAYS WITH CELL FACES

In the following, the assumption is made that the quadrangle $(P_0P_1P_2P_3)$ formed by a cell face is non-degenerate, i.e. its four corners P_0, P_1, P_2, P_3 are all different and the edges are not collinear. The case of a *planar* quadrangle can easily be treated by subdividing the quadrangle into two triangles and intersecting the ray with the plane containing these triangles. We will, however, in the following assume that the quadrangle is *non-planar*, which is the more general and more complex case.

At first, we must establish the transformation between physical and computational space. Since the quadrangle $(P_0P_1P_2P_3)$ is non-planar, it can be regarded as part of a *bilinear surface*, and mapped from physical space to computational space as follows (see Figure C.3):

$P_0(x_0, y_0, z_0)$ is mapped to $Q_0(0, 0, 0)$

$P_1(x_1, y_1, z_1)$ is mapped to $Q_1(0, 1, 0)$

$P_2(x_2, y_2, z_2)$ is mapped to $Q_2(1, 0, 0)$

$P_3(x_3, y_3, z_3)$ is mapped to $Q_3(1, 1, 1)$

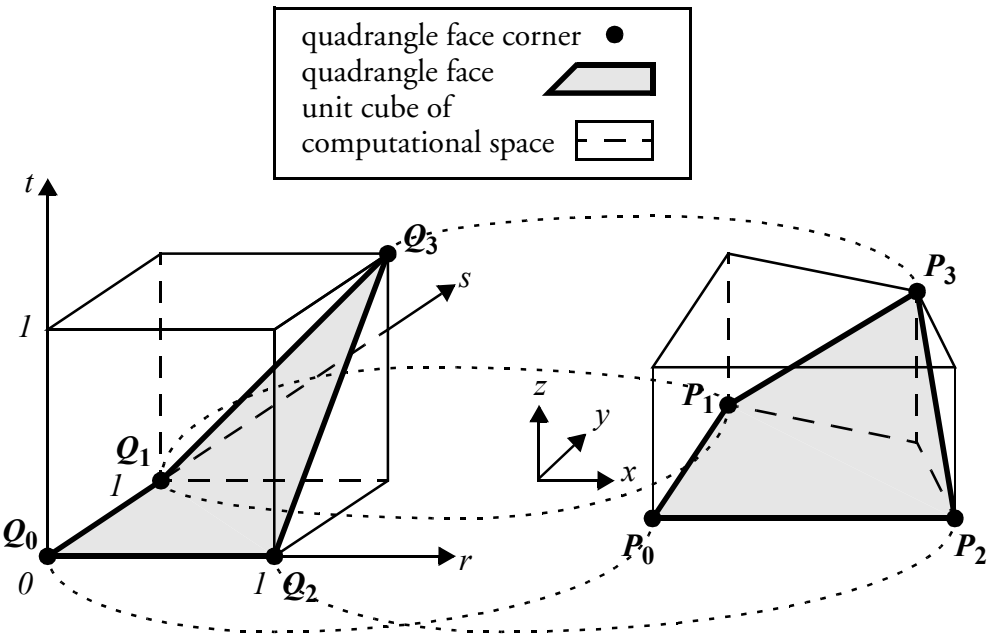


FIGURE C.3 Mapping of a non-planar quadrangle between computational and physical space.

Q_0, Q_1, Q_2, Q_3 are corners of the unit cube and span a bilinear face, since for all these corners, the three coordinates (r, s, t) in computational space fulfil the bilinear surface equation

$$r \cdot s = t \tag{C.1}$$

The mapping from computational to physical space can easily be done using an *affine transformation*. Since the quadrangle $(P_0P_1P_2P_3)$ is non-degenerate and non-planar, the three span vectors

$$\boldsymbol{p_{01}} = \boldsymbol{p_1} - \boldsymbol{p_0} \tag{C.2}$$

$$\boldsymbol{p_{02}} = \boldsymbol{p_2} - \boldsymbol{p_0} \tag{C.3}$$

$$\boldsymbol{p_{03}} = \boldsymbol{p_3} - \boldsymbol{p_0} \tag{C.4}$$

are linearly independent, thus the matrix

$$A = \left[\boldsymbol{p_{02}} \quad \boldsymbol{p_{01}} \quad (\boldsymbol{p_{03}} - \boldsymbol{p_{02}} - \boldsymbol{p_{01}})\right] \tag{C.5}$$

is regular. As can easily be shown, the affine *point* transformation

$$\mathbb{R}^3 \to \mathbb{R}^3: \qquad \left[x\ y\ z\right] = A\left[r\ s\ t\right] + \boldsymbol{p_0} \tag{C.6}$$

is then bijective and maps Q_0, Q_1, Q_2, Q_3 onto P_0, P_1, P_2, P_3, and also every point from computational space to physical space unambiguously. The corresponding affine *vector* transformation is the same, apart from the missing translation vector $\boldsymbol{p_0}$:

$$\mathbb{R}^3 \to \mathbb{R}^3: \qquad \left[x\ y\ z\right] = A\left[r\ s\ t\right] \tag{C.7}$$

However, since we also need the *inverse* mapping from physical to computational space, we must invert the matrix A and use the *inverse point* transformation

$$\mathbb{R}^3 \to \mathbb{R}^3: \qquad \left[r\ s\ t\right] = A^{-1}(\left[x\ y\ z\right] - \boldsymbol{p_0}) \tag{C.8}$$

to map P_0, P_1, P_2, P_3 onto Q_0, Q_1, Q_2, Q_3, and also every point from physical space to computational space in a unique manner. Its respective *inverse vector* transformation is

$$\mathbb{R}^3 \to \mathbb{R}^3: \qquad \left[r\ s\ t\right] = A^{-1}\left[x\ y\ z\right] \tag{C.9}$$

To compute the intersections of the (physical) ray

$$\overrightarrow{\boldsymbol{xyz}}[\lambda] = \boldsymbol{p} + \lambda \cdot \boldsymbol{d} \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \tag{C.10}$$

with the bilinear surface, we now transform the ray by inserting the start point $\boldsymbol{p}$ of the ray into the inverse point transformation:

$$\boldsymbol{p'} = A^{-1}(\boldsymbol{p} - \boldsymbol{p_0}) \tag{C.11}$$

and by inserting the ray direction $\boldsymbol{d}$ into the inverse vector transformation:

$$\boldsymbol{d'} = A^{-1}\boldsymbol{d} \tag{C.12}$$

It is easy to prove that the λ parameter needs not to be transformed, since this real number has the same value in physical and computational space (due to the *aspect ratio conservation* property of affine transformations). The transformed ray thus can be written as

$$\overrightarrow{\boldsymbol{rst}}(\lambda) = \boldsymbol{p}' + \lambda \cdot \boldsymbol{d}' \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \tag{C.13}$$

or, component-wise, as a set of three scalar functions:

$$r(\lambda) = p'_r + \lambda \cdot d'_r \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \tag{C.14}$$

$$s(\lambda) = p'_s + \lambda \cdot d'_s \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \tag{C.15}$$

$$t(\lambda) = p'_t + \lambda \cdot d'_t \qquad (\lambda \in \mathbb{R}, \lambda \geq 0) \quad . \tag{C.16}$$

Substituting r, s, t in the bilinear surface equation $r \cdot s = t$ (Equation C.1) by these scalar functions yields the intersection equation

$$r(\lambda) \cdot s(\lambda) = t(\lambda) \tag{C.17}$$

$$(p'_r + \lambda \cdot d'_r) \cdot (p'_s + \lambda \cdot d'_s) = (p'_t + \lambda \cdot d'_t) \tag{C.18}$$

$$(d'_r d'_s) \cdot \lambda^2 + (p'_r d'_s + p'_s d'_r - d'_t) \cdot \lambda + (p'_r p'_s - p'_t) = 0 \tag{C.19}$$

Solving this quadratic equation for λ, and inserting the resulting two values λ_1, λ_2 (provided that they are real) into the transformed ray equation, we get the local coordinates of the intersection points I_1, I_2:

$$\boldsymbol{i}_k' = \overrightarrow{\boldsymbol{rst}}(\lambda_k) = \boldsymbol{p}' + \lambda_k \cdot \boldsymbol{d}' \qquad (k \in \{1, 2\}, \lambda_k \in \mathbb{R}) \tag{C.20}$$

If the local coordinates of an intersection point I are written as

$$\boldsymbol{i}' = \begin{bmatrix} r & s & t \end{bmatrix}, \tag{C.21}$$

then the test whether the intersection point lies inside the quadrangle now reduces to a test whether its local coordinates lie between 0 and 1:

$$I \in (P_0 P_1 P_2 P_3) \Leftrightarrow (r, s) \in \begin{bmatrix} 0, 1 \end{bmatrix} \times \begin{bmatrix} 0, 1 \end{bmatrix} \tag{C.22}$$

(since $r \cdot s = t$ holds for all points of the bilinear surface, the third coordinate t then automatically lies between 0 and 1, too).

The local coordinates of an intersection point can also be used for bilinear interpolation of its scalar value from the scalar values at the four quadrangle corners. Let $f_{00}, f_{01}, f_{10}, f_{11}$ be the scalar values at the four face corners P_0, P_1, P_2, P_3. Then the scalar value at the intersection point with local coordinates (r, s, t) computes to

$$f(r, s, t) = (1 - r)(1 - s) \cdot f_{00} + (1 - r)s \cdot f_{01} + r(1 - s) \cdot f_{10} + rs \cdot f_{11} \tag{C.23}$$

The procedure *RayCastQuadrangleFace()* shown in Figure C.4 is called by the function *FindEndpointOfRay()* of Figure C.2. It returns the λ values, local coordinates, inside quadrangle test result flags and scalar values of all real intersection points of the current ray with the current face. If the solutions of the intersection equation are complex, they are ignored by setting the inside quadrangle test result flags to FALSE.

```
procedure RayCastQuadrangleFace(startpoint p, raydirection d)
                                coords0, coords1, coords2, coords3,
                                scalar0, scalar1, scalar2, scalar3,
                                &lambda[2], &scalar[2], &inside[2])

// Initialisations
for k = 0 to 1
   lambda[k] := 0.0
   scalar[k] := 0.0
   inside[k] := FALSE
next k

// Compute span vectors p01, p02, p03 and transformation matrix A.
// Check if matrix A is regular or singular.
p01 := p1 - p0
p02 := p2 - p0
p03 := p3 - p0
A := [p02, p01, (p03 - p02 - p01)]

// General case: non-planar quadrangle
if det(A) <> 0.0 then

   // Invert matrix A and transform the ray
   // from physical to computational space

   A_inv := invertMatrix(A)

   p' := (pr,ps,pt) := A_inv * (p - p0)
   d' := (dr,ds,dt) := A_inv * d

   NumReal := SolveDegree2(dr*ds, pr*ds + ps*dr - dt, pr*ps - pt,
                           &lambda[0], &lambda[1])

   for solution k = 0 to 1
      rst[k] := (r,s,t) := p' + lambda[k] * d'

      // Check if intersection point lies inside QUADRANGLE
      inside[k] := (r >= 0.0) and (r <= 1.0) and (s >= 0.0) and (s <= 1.0)

      // Evaluate scalar field at intersection point
      scalar[k] := ScalarBilinearInterpolation
                   (scalar0,scalar2,scalar1,scalar3, r,s)
   next solution k

// (Special case: planar quadrangle)
else
...// Can be treated by subdividing the planar quadrangle
...// into two triangles and linear interpolation of the scalar values
end if
```

FIGURE C.4 The algorithm for intersecting a ray with a quadrangle cell face.

C.2 CELL TRAVERSAL IN HEXAHEDRAL GRIDS

To minimise numerical inaccuracies, the algorithms described in this chapter were implemented using floating point variables of double precision. Nevertheless, there were some numerical problems to deal with that were not trivial to solve.

One of the most difficult problems was the decision whether a newly found intersection point of a ray with a cell face will extend the ray (see also Section 6.2.2). In theory, one computes the intersection points with *all* faces of the cell currently being traversed. The point with the *smallest* λ value *greater* than the old λ value *currentLambda* (from the previous cell) is then chosen to extend the ray (see Figure 6.7). But this decision is somewhat critical because it depends on numerical comparisons of real-number λ values. Let us have a look at an excerpt of the procedure *FindEndpointOfRay()* from Figure C.2:

```
// Select the nearest intersection point:
for face i = 0 to 5
   if inside[k] = TRUE and
      lambda[k] > currentLambda and
      lambda[k] < nextLambda then
         nextLambda := lambda[k]
         nextScalar := scalar[k]
   end if
next face i
```

The second comparison

```
if lambda[k] > currentLambda ...
```

shall ensure that any chosen intersection point must indeed extend the ray. The comparison works well when the two λ values belong to points on *different* faces of the *same* cell (see Figure C.5, left).

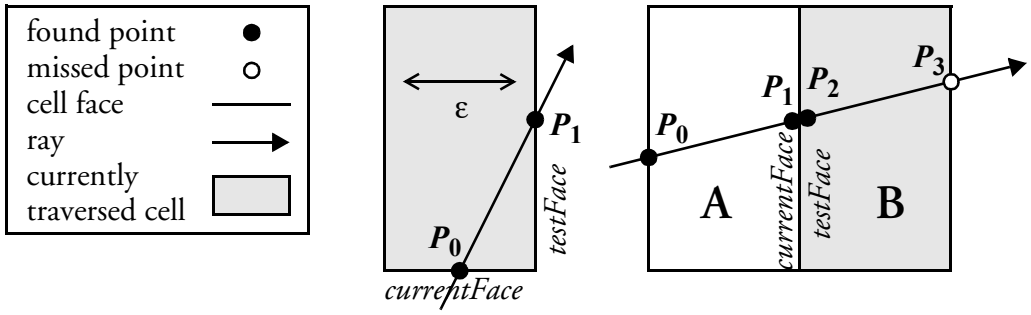


FIGURE C.5 Different intersection situations (2D representation).
Left: points of same cell on different faces. Right: points of different cells on same face.

However, regard the case of two points lying on the *same* face of two *different* neighbouring cells (see Figure C.5, right). Assume that the ray has been extended up to point P_1 (*currentLambda*) when cell A was traversed. When the algorithm switches to cell B, the newly found point P_2 (*lambda[k]*) might have a λ value slightly greater than P_1 (*currentLambda*), although the two λ values in theory must be equal. P_2 is therefore wrongly selected to extend the ray, since its λ value is the smallest of all intersection points (P_2, P_3) of cell B with the ray. But this forces the algorithm to stop ahead of time, since the point P_3 found in the same cell B has a greater λ value than P_2 and thus will be ignored.

We therefore had to modify the second comparison by introducing a small ε constant and testing if the new λ value is "really" greater than the current one:

```
if lambda[k] > currentLambda + EPSILON ...
```

However, this was still not sufficient because the case of two points lying on different faces on the same cell (which had worked well without the ε) did now not always work properly anymore. The new problem was caused by rays intersecting a cell near an edge of the grid, where two faces meet (Figure C.6, left). In such a case, the two λ values are close together but not equal, so the ε might prevent the higher λ value of P_1 from being accepted (because it is not enough greater than that of P_0). The algorithm will then also stop too early, yielding a ray endpoint too close to the vortex core line, and thus a well visible "chuck hole" in the vortex hull surface.

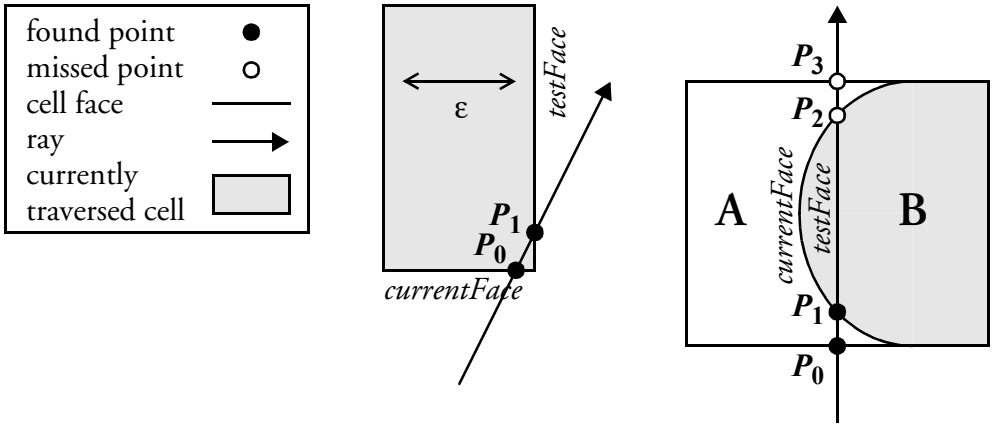


FIGURE C.6 Further intersection situations (2D representation). Left: two intersections near an edge. Right: two intersections on the same face.

It is therefore necessary to distinguish between two cases, and to store for each intersection point extending the ray, on which grid face (global face number) it was found. For every newly treated face, its global face number is determined (*testFace*) and compared to the global face number of the current ray endpoint (*currentFace*). Depending on the result, the ε value is used for the comparison or not.

One could argue that it is not necessary to treat a face which has already been treated in the preceding cell. However, this is not always true because a cell face can be non-planar and thus have several (if bilinearly interpolated, two) intersection points with the ray (see Figure C.6, right). The algorithm in this case finds the point P_1 on the non-planar face inside cell A, stores P_1 as the current ray endpoint, memorises the global face number of the non-planar face and then switches to cell B. Ignoring the non-planar face inside cell B would miss the point P_2 and stop the algorithm again ahead of time (because no intersection point can be found on any other face of cell B).

Storing the *greater* rather than the smaller λ value resulting from the quadratic intersection equation within cell A, though, would extend the ray immediately to point P_2 rather than to point P_1. But if we omit point P_1, we could miss a threshold crossing of the scalar field between P_0 and P_1, or between P_1 and P_2. So this would neither be a correct way to handle the matter.

It is therefore necessary to compute and check *every* intersection point, so we must in each step choose the *nearest* intersection point of a cell as the next ray endpoint (as long as the threshold of the scalar field has not yet been crossed). Taking all these considerations into account, the modified *FindEndpointOfRay()* function looks as shown in Figure C.7:

```
function FindEndpointOfRay(seedPoint, rayDirection)

// Initialisations
EPSILON := 0.0001
current[Cell, Face, Lambda, Scalar] := (seedCell, VOID, 0.0, seedScalar)

// main loop for traversing the ray
for step = 0 to MAXSTEP
   ...
   nextFace := VOID
   nextLambda := INFINITE

   for face i = 0 to 5
      // Get global face number of current test face
      testFace := GlobalFaceNumber[currentCell, i]

      // Find intersection points of ray with test face
      RayCastQuadrangleFace(seedPoint, rayDirection,
                            &lambda[2], &scalar[2], &inside[2])

      // Select the nearest intersection point of the ray
      // with the cell currently being traversed
      for solution k = 0 to 1
         if testFace = currentFace then epsilon := EPSILON
         if testFace <> currentFace then epsilon := 0

         lambdaBigger := (lambda[k] > currentLambda + epsilon)
         lambdaSmaller := (lambda[k] < nextLambda)

         if inside[k] and lambdaBigger and lambdaSmaller then
            next[Face, Lambda, Scalar] := (testFace, lambda[k], scalar[k])
         end if
      next solution k
   next face i

   // If grid boundary has been reached or
   // if scalar threshold has been crossed or
   // if no intersection point could be found with any cell face,
   // terminate and return the current ray endpoint
   if ... then return currentLambda

   // Otherwise, switch to next cell
   nextCell := NeighbouringCell[currentCell, nextFace]
   current[Cell, Face, Lambda, Scalar] := next[Cell, Face, Lambda, Scalar]
next step
```

FIGURE C.7 The modified function *FindEndpointOfRay()*.

APPENDIX D

CURRICULUM VITAE

Personal Data

Dirk Bauer
Swiss Federal Institute of Technology (ETH)
Computer Graphics Laboratory (CGL)
ETH Zentrum, IFW C 27.1
CH-8092 Zürich, Switzerland
Phone: 0041-1-632 0783
Email: bauer@inf.ethz.ch
Web: http://graphics.ethz.ch/~bauer/

01 Jan 1971	Born in Tübingen, Germany

Education and Career

1977 - 1981	Primary school in Tübingen, Germany
1981 - 1990	Secondary school in Tübingen, Germany
1990	University-entrance diploma in Tübingen, Germany
1991 - 1997	Study of Computer Science at University of Tübingen, Germany
1995 - 1996	Student research assistant at University of Tübingen, Germany
1997	Diploma thesis at debis Systemhaus, Stuttgart, Germany
1998	Computer Science diploma at University of Tübingen, Germany
1997 - 2000	Software engineer at Hewlett Packard, Böblingen, Germany
2000 - 2006	Scientific assistant and Ph.D. study at ETH Zurich
since 2006	Scientific assistant at University of Zurich

Publications

University of Tübingen diploma thesis 1997	Dirk Bauer: *OCR-Tool-Integration in ein Dokumentenverwaltungssystem* *(OCR tool integration into a document management system)*
Contribution to *Taschenbuch mathematischer Formeln* Fachbuchverlag Leipzig 19th edition, 2001	Hans-Jochen Bartsch (editor), Dirk Bauer (contributor): *Allgemeine Gleichung 4. Grades* (improved solution formula for general quartic equation)
IEEE TCVG Visualization Symposium 2002 Barcelona, Spain	Dirk Bauer, Ronald Peikert: *Vortex Tracking in Scale-Space*
IEEE Visualization 2002 Boston, USA	Dirk Bauer, Ronald Peikert, Mie Sato, Mirjam Sick: *A Case Study in Selective Visualization of Unsteady 3D Flow*