Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

DETLEV STALLING

Fast Texture-Based Algorithms for Vector Field Visualization

Dissertation am Fachbereich Mathematik und Informatik der Freien Universität Berlin

Preprint SC 98-40 (December 1998)

Acknowledgements

The work described in this thesis has been carried out from 1995 to 1998 at the department of Scientific Visualization at Konrad-Zuse-Zentrum Berlin (ZIB). I would like to thank my tutors Prof. Dr. Peter Deuflhard and Hans-Christian Hege who encouraged me to investigate an interesting new area between mathematics, computer science, and art, namely scientific visualization. I deeply appreciated the creative atmosphere at ZIB and enjoyed many fruitful and inspiring discussions. Malte Zöckler deserves special thanks for helping me to develop and implement many ideas presented in this work. I also would like to thank all other members and ex-members of the department of Scientific Visualization at ZIB for many suggestions and practical help offered every day.

Contents

Chapter 1

Introduction

1.1 Problem Formulation

Vector fields provide an important concept in science and engineering. They allow us to describe a wide variety of phenomena like fluid flow, electromagnetic fields, or, more generally, the dynamic evolution of arbitrary systems in phase space. Often vector fields exhibit complex structures of great diversity. Therefore, it is no surprise that visual methods have been applied to investigate these structures for a very long time. Today, when powerful graphics computers are available, visual methods play an even greater role. In the field of scientific visualization numerous methods have been developed to depict vector fields graphically. Among the questions one hopes to answer by means of visualization are the identification of features like sources or sinks, vortices, or periodic orbits. Without an intuitive image of a vector field the dynamical behaviour of the underlying system cannot be understood in full detail.

Despite the variety of application areas and types of vector fields, there are a number of general requirements which ideally have to be met by a visualization method. In particular we want to focus on the following issues:

- *Accuracy.* The structure of a vector field should be faithfully represented in an image. Otherwise, misleading conclusions can be drawn. Beside the visualization approach itself, on a computer the underlying numerical methods are of crucial importance. In many existing algorithms numerical quality still is sacrified. For example, when trajectories are to be computed and a poor integrator is used, severe errors can occur or singularities can be overlooked. Likewise, often vector fields are resampled on a regular grid before visualizing them, or improper interpolation methods are used for discrete data. This may lead to artifacts, e.g. periodic orbits may not appear closed or conservation laws may seem to be violated.

- *Performance.* The visualization method should be reasonably fast to allow interactive exploration of the data. On the one hand, this requires the use of suitable data-structures and efficient algorithms. If possible, modern 3D graphics hardware should be exploited. On the other hand, the dependence on user-defined parameters also influences the overall performance of a visualization method. For many existing techniques suitable parameters have to be found by trial and error. Thus, a considerable amount of time can be necessary until a meaningful image of a vector field is obtained.

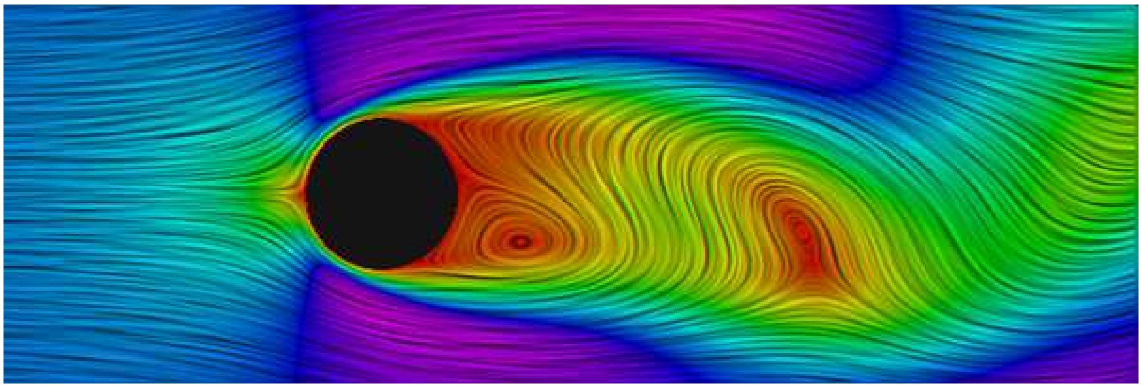
- *Cognition.* This point aims at the design of the visualization method itself. It is highly desirable to depict important information in an intuitive and easy-to-understand way. A severe drawback of many existing techniques is that they require a significant amount of interpretation or brain-work in order to understand the relevant structures of a vector field from its visual representation. For example, continuous objects like trajectories are often decomposed into discrete arrow-style symbols, which then have to be integrated mentally. Whenever possible, the display should be adapted to the human visual system. This includes proper use of color, texture, and shading.

In this thesis we develop new methods for visualizing vector fields which specifically address these three design goals: accuracy, performance, and cognition. Our methods will be general-purpose and can be applied to arbitrary vector fields in two- and three-dimensional space. The methodology behind our approach is *texture-based visualization*. Texture-based visualization methods imitate techniques known from experimental flow visualization, namely, the observation of randomly dispersed particles or dye injection patterns. Instead of depicting individual lines or symbols, a contiguous high-resolution image or texture is generated. This texture clearly reveals the directional structure of the field. In this way intuitive insight can be obtained and even small details of the field become visible.

The task is to elaborate the general idea of texture-based visualization into practical algorithms. What kind of textures are best suited for our purpose? How do we generate them? Can we apply these methods in three-dimensional space? In this work, we focus on a technique known as *line integral convolution* or LIC. This method turns out to be quite versatile and well-suited for visualizing many interesting vector fields. LIC images display the integral curves or field lines of a vector field at high spatial resolution. Although conceptually quite simple, line integral convolution implies a number of interesting mathematical and algorithmic questions. For these we shall find innovative and fruitful answers.

1.2 Vector Fields in Science and Engineering

At this point we want to present examples of vector fields from various disciplines. Our intention is to point out that vector field visualization plays an important role in many application areas. The figures in this section illustrate some of the visualization techniques developed later in this work.



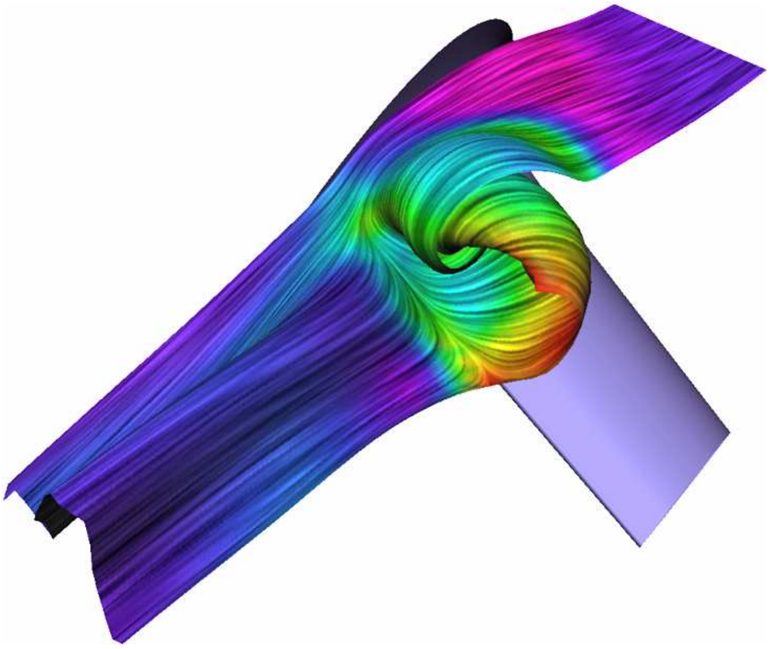
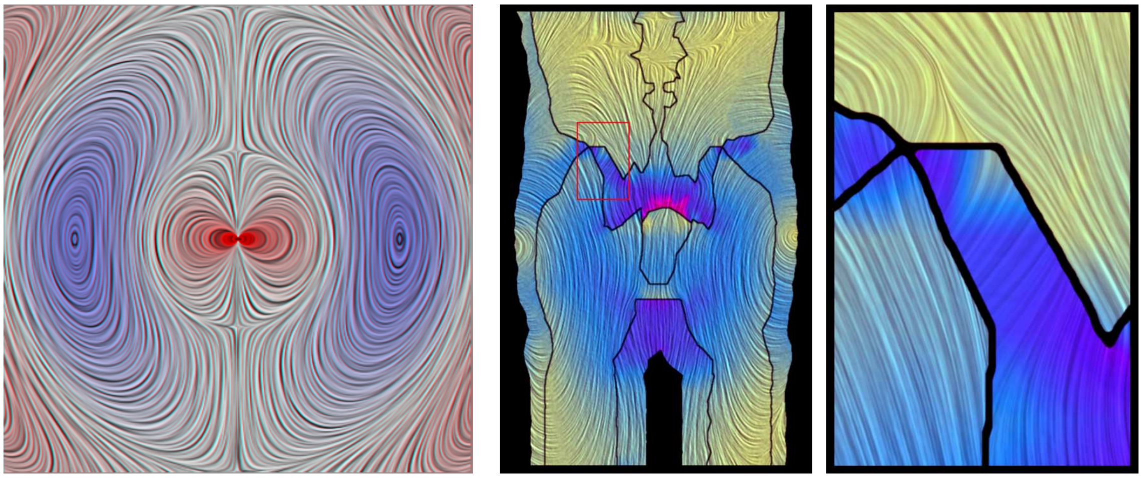


Figure 1.1: Vector fields in fluid mechanics. Above the instationary flow around a cylinder is shown. The LIC pattern depicts stream lines at a particular instance of time. On the left a surface LIC technique is applied to visualize vortex structures in a flow around an airfoil.

Fluid Mechanics. A vector field can be thought of as describing the path of a little particle in space and time. This intuitive interpretation is most evidently adequate in fluid mechanics, where the motion of some liquid or gaseous material is described by means of a velocity field $\boldsymbol{u}(\boldsymbol{x}, t)$. The equations governing fluid motion can be derived from some fundamental principles such as conservation of mass, momentum, and energy [13]. Under the most simple assumptions, the velocity field of a perfect non-viscuous fluid is determined by Euler's equations. Taking into account viscosity leads to the Navier-Stokes equations. In contrast to many other vector fields the flow of an incompressible fluid is characterized by vanishing divergence. Beside velocity itself in fluid mechanics there is another important vector field, namely vorticity or $\nabla \times \boldsymbol{u}$.

The numerical simulation of fluid flow has been an important area of research since the early days of computing. Likewise, the graphical display of fluid flows has influenced scientific visualization from its very beginnings. Today still the vast majority of vector field visualization methods are targeted to fluid mechanics.



a) dipole antenna

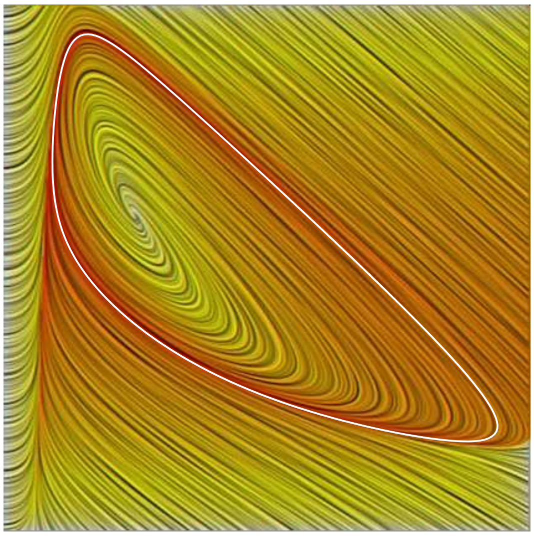
b) E-field from hyperthermia planning

Figure 1.2: Vector fields in electrical engineering. On the left the electric field of a radiating dipole antenna is shown. The images on the right come from a medical application called hyperthermia planning. The simulated electric field in a cross-section of a human body is visualized. Note, how vector field direction changes abruptly at tissue boundaries.

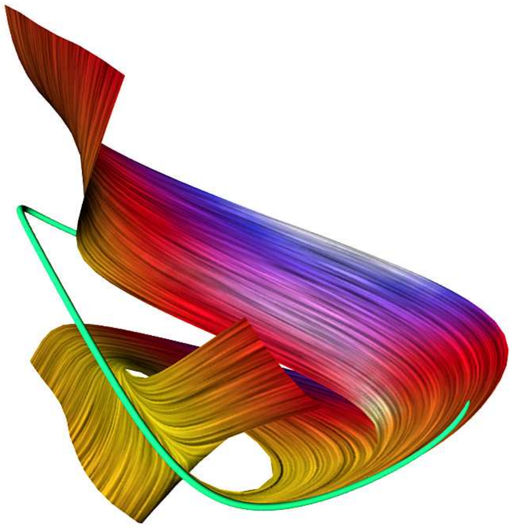
In Figure 1.1 the simulated flow around a cylinder is depicted by means of line integral convolution. The image shows instantaneous stream lines of the velocity field. From the physical point of view the formation of vortices and recirculation areas is interesting. These features can be clearly identified in the image. The same figure also contains a three-dimensional example, namely the flow around an airfoil. In this case a texture generated by line integral convolution has been mapped onto a so-called stream surface of the vector field. This surface together with the directional LIC texture clearly reveals the formation of vortices on top of the wing.

Electromagnetism. Like fluid mechanics eletromagnetism has a long tradition in science. Electromagnetic theory has already been developed in the 19th century, when the fundamental laws for the electric field $\boldsymbol{E}$ and the magnetic field $\boldsymbol{B}$ were found. Historically, the investigation of electric and magnetic phenomena inspired Faraday to introduce the concept of a field in science [64]. Electromagnetic fields are of abstract nature, since they do not describe the motion of real matter. Instead, an electric field $\boldsymbol{E}$ gives rise to a force $\boldsymbol{F} = q\,\boldsymbol{E}$ on a small test particle of charge q. Similarly, a magnetic field $\boldsymbol{B}$ gives rise to a torque or moment-of-force $\boldsymbol{N} = \boldsymbol{m} \times \boldsymbol{B}$ on a small magnetic dipole $\boldsymbol{m}$. This is the reason why compass needles align along the lines of a magnetic field.

The lines of an electric field can be thought of as originating from positive electric charges and terminating at negative ones. In particular, the divergence of an electric field is proportional to charge density. In contrast, there is no magnetic analogon to an electric charge. Consequently, the divergence of a magnetic field is always zero, i.e., magnetic field lines are always closed. A time-varying magnetic field gives rise to an electric field, while a time-varying electric field or electric current gives rise to a magnetic field. This mutual dependence allows the formation of electromagnetic waves. The full theory of



a) $\dot{x} = 1 + x^2y - 4x,\ \dot{y} = 3x - x^2y$



b) $\dot{x} = 1 + x^2y - (z{+}1)\,x,\ \dot{y} = xz - x^2y$
$\dot{z} = -xz + 1.45$

Figure 1.3: Solution of the 2D- and 3D-Brusselator model. These dynamical systems describe an oscillating chemical reaction. All solution curves approach the highlighted limit cycles. In the 2D case color illustrates sampling density in phase space. Red (dark) regions are covered by a large number of integral curves.

electromagnetism is summarized by Maxwell's equations [46].

Figure 1.2 a) illustrates an electromagetic wave phenomenon. It shows the electric field radiated by an oscillating dipole antenna at one instance of time. As time proceeds, the wave fronts move in outward direction. The solution of this problem has been found analytically by Hertz in 1897. In Figure 1.2 b) a numerically simulated field inside a human body is shown. This example comes from an application called hyperthermia planning [23]. The goal is to heat certain parts of the body by means of radiowaves. Note, that field direction abruptly changes at tissue boundaries. This is due to different electric properties of the tissue compartments.

Dynamical Systems. Free from any physical interpretation the right hand side of any system of first-order ordinary differential equations can be regarded as a vector field. Such a set of equations is commonly used to describe the evolution of state variables in time. Therefore the differential equations are also called a dynamical system. The state variables might be concentration rates in a chemical reaction, population rates in an ecological model, rates in stock market, or many more. The equations of motion of a mechanical system, e.g. a pendulum, can be interpreted as a dynamical system, too. In this case the state variables comprise of position and momentum of the different degrees of freedom. The state variables are said to form the phase space of the system. When analysing the evolution of a dynamical system one is interested in the existence of periodic solutions, in the asymptotic behaviour, or in the stability of the solutions.

In case of low-dimensional systems, i.e., two- or three-dimensional ones, methods for vector field visualization can be directly used to depict the flow of the system in phase space. For example, Figure 1.3 a) shows the solution of a 2D chemical reaction model, called the Brusselator [52]. The two axes denote concentration rates of two different substances. It can be observed that irrespectively of the initial situation the system approaches a limiting cycle. This means that the concentration rates change periodically. An extended three-dimensional version of the same Brusselator model is shown in Figure 1.3 b). In this case the structure of the system is visualized by means of texture-mapped stream surfaces. Again, it can be observed that the system runs into a limit cycle.

Differential Geometry. Vector fields can not only be defined in Euclidean space, but also on non-planar manifolds. Differential geometry provides a rich set of examples for this. In particular, various types of tangent curves defined on surfaces in $\mathbb{R}^3$ are amenable for straight-forward visualization. The most obvious example is direction of curvature. For each point on a surface there are two directions of principal curvature. In Figure 1.4 the directions of maximal curvature are shown for a triangulated torus. Note, how this direction changes abruptly at certain points of the surface. At these points there are two directions of equal curvature. Another example are so-called isophotes, lines which always make up equal angles to view direction and the surface's normal vector. These lines are commonly used in computer aided modelling as a tool for surface interrogation [68].

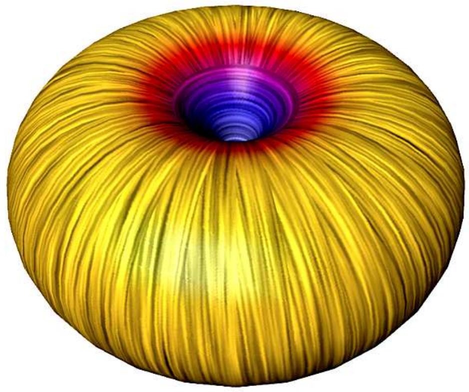


Figure 1.4: Principal lines of curvature of a triangulated torus. Color denotes magnitude of mean curvature.

Image Processing. Interesting vector fields also occur in image processing or computer vision. For example, given a motion series of arbitrary images, displacement vectors can be computed stating where a point in one image moves to in the next. This type of information allows one to track objects in a movie. Of course, displacement vectors cannot always be defined exactly. A feature visible in one image might not be present in the next one. Sudden changes in visibility lead to discontinuities. Methods to compute displacement vectors can be divided into differential approaches (optical flow) or correlation approaches [47].

Sometimes vector fields are also created synthetically in order to achieve digital art effects. For example, Figure 1.5 shows variations of a painting of Henri Matisse. The directional textures in this figure are aligned perpendicular, respectively parallel, to the contour lines of the distance map of the blue parts of the image.

Of course, the list of vector fields we mentioned certainly is not complete. There are many more interesting application areas. However, our intention was to illustrate the diversity of different types of vector fields. It is clear, that adequate methods for visualizing directional data are of general significance for many disciplines.

Figure 1.5: Variations of a painting of Henri Matisse. The original image is shown on the left. The directional texture in the middle image has been computed from the gradient of the distance map of the blue parts of the image. For the image on the right gradient vectors have been rotated by $90°$.

1.3 Outline of the Thesis

This work contains four major parts excluding the introductory chapter. The first two of them cover necessary background information, while the other two describe our vector field visualization algorithms in detail.

Chapter 2, *Concepts and Previous Work*, starts with a formal introduction of vector fields. We fix our notation and define mathematical objects like trajectories, stream lines, or streak lines. We then consider the special case of linear fields. Linear fields are of particular interest since they can be integrated analytically. Moveover, most topological features occuring in general vector fields can already be studied in the linear case. Thus, a short classification of topological features is presented to supplement our further work. After the necessary theoretical background has been discussed, we review previous approaches to vector field visualization in Section 2.3.

Chapter 3, *Underlying Numerical Methods*, provides the technical ground on which we can formulate our visualization algorithms. This chapter is divided into two parts. The first part describes methods for numerically computing integral curves in a vector field. We focus on a particular type of numerical integration methods, namely Runge-Kutta methods with error monitoring and adaptive step size control. This class of integrators can be easily equipped with efficient interpolation methods, allowing dense sampling of solution curves as required by our visualization algorithms. The second part of Chapter 3 is devoted to the representation and evaluation of vector fields on a computer. We introduce different grid types and interpolation methods. Fast evaluation of fields at arbitrary locations in a non-uniform grid is non-trivial. It requires the efficient use of spatial data structures and efficient root finding techniques.

In Chapter 4, *Line Integral Convolution in* $\mathbb{R}^2$, visualization of planar vector fields is considered. We motivate texture-based visualization methods by analyzing particle streak experiments in fluid mechanics. We outline the principles of line integral convolution (LIC), our favoured method for vector field visualization in 2D. A fast algorithm for computing LIC images will be developed. A sound analysis of the involved statistical processes turns out to be the key for computing high-quality images of vector fields. These images can also be animated in order to emphasize sign or magnitude of the flow. Techniques for computing such animations conclude this chapter.

Finally, in Chapter 5, *Application to 3D Vector Fields*, we discuss how to generalize LIC for visualizing three-dimensional fields. Actually, we don't cover direct volume rendering but concentrate on the development of surface LIC algorithms. It turns out, that stream surfaces provide an excellent tool for extracting relevant structures from a 3D vector field. Therefore in Section 5.1 we first discuss how to construct such surfaces. We review existing algorithms and propose improvements which allow us to process not only flow fields but also general vector fields containing sinks and sources. In order to visualize the flow structure on a stream surface, in Section 5.2 we first investigate the generation of surface LIC textures in parameter space. A technique for compensating distortions caused by non-isometric texture mapping is presented. Alternatively, surface LIC textures can also be computed in physical space. Algorithmic details of such an approach are treated in Section 5.3. Among other things, in this section we discuss integration of field lines on triangular domains, compare different ways of defining local triangle textures, and finally present strategies to optimze the use of texture memory.

Key ideas presented in this work have been published and presented at international conferences during the last three years. In particular, the "fast" algorithm for computing LIC images has been described in a joint paper with Hans-Christian Hege presented at *SIGGRAPH'95* [81]. A first implementation of a surface LIC technique was developed together with Henrik Battke and Hans-Christian Hege in 1996 [3]. In the following, these algorithms were applied to particular types of vector fields and further algorithmic improvements were achieved [82, 96]. All these techniques were presented in a tutorial course at *SIGGRAPH'97* [40, 80]. Finally, an extension of the fast LIC algorithm to higher-order filter kernels was first described in [41]. New and unpublished results presented hereafter include the comparison of LIC and particle streak experiments in fluid mechanics, the formulation of a numerical stable evaluation scheme for fast LIC with higher-order filter kernels, statistical analysis of LIC images, design of anti-aliasing strategies, development of an improved stream surface algorithm, noise synthesis for surface LIC in parameter space, as well as the design of a texture-blending technique to be used for surface LIC in physical space.

Chapter 2

Concepts and Previous Work

2.1 Basic Notation

In this section we want to introduce vector fields formally. We recall the important concept of integral curves and state, under which conditions these curves exist. For a more detailed discussion and for proofs we refer to standard textbooks such as [66].

2.1.1 Vector Fields

A vector field $\boldsymbol{f}$ defined on an open set E of Euclidean space $\mathbb{R}^n$ can be regarded as a mapping $\boldsymbol{f}(\boldsymbol{x}) : E \mapsto \mathbb{R}^n$. This mapping assigns each point of E a directional quantity such as a force or velocity vector. In this paper we restrict ourselves to the cases $n = 2$ and $n = 3$, since these are amenable to visualization in an intuitive way. In addition to stationary fields $\boldsymbol{f}(\boldsymbol{x})$, we will also consider time-dependent fields $\boldsymbol{f}(\boldsymbol{x}, t) : E \times J \mapsto \mathbb{R}^n$, where J is some time interval $a \leq t \leq b$. Continuity as well as the derivative of a vector field are defined in the usual way as follows.

Definition 1. Suppose U_1 and U_2 are normed linear spaces with norms $|| \cdot ||_1$ and $|| \cdot ||_2$. Then a mapping $\boldsymbol{f} : U_1 \mapsto U_2$ is *continuous* at $\boldsymbol{x}_0 \in U_1$ if for all $\epsilon > 0$ there exists a $\delta > 0$ such that $\boldsymbol{x} \in U_1$ and $||\boldsymbol{x} - \boldsymbol{x}_0||_1 < \delta$ implies that

$$|| \, \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x}_0) \, ||_2 < \epsilon. \tag{2.1}$$

The field $\boldsymbol{f}$ is said to be continuous on the set $E \in U_1$ if it is continuous at each point $\boldsymbol{x} \in E$. In this case we write $\boldsymbol{f} \in C^0(E)$.

For vector fields defined in $\mathbb{R}^n$ we usually apply the Euclidean norm of $\boldsymbol{x} \in \mathbb{R}^n$,

$$|\boldsymbol{x}| = \sqrt{x_1^2 + ... + x_n^2}. \tag{2.2}$$

Definition 2. Suppose U_1 and U_2 are normed linear spaces with norms $|| \cdot ||_1$ and $|| \cdot ||_2$. Then a mapping $\boldsymbol{f} : U_1 \mapsto U_2$ is *differentiable* at $\boldsymbol{x}_0 \in U_1$ if there is a linear transformation $\boldsymbol{f}'(\boldsymbol{x}_0) \in L(U_1, U_2)$ that satisfies

$$\lim_{||\boldsymbol{h}||_1 \to 0} \frac{||\, \boldsymbol{f}(\boldsymbol{x}_0 - \boldsymbol{h}) - \boldsymbol{f}(\boldsymbol{x}_0) - \boldsymbol{f}'(\boldsymbol{x}_0)\, \boldsymbol{h}\, ||_2}{||\, \boldsymbol{h}\, ||_1} = 0. \tag{2.3}$$

The linear transformation $\boldsymbol{f}'(\boldsymbol{x}_0)$ is called the derivative of $\boldsymbol{f}$ at $\boldsymbol{x}_0$.

The derivative of a mapping $\boldsymbol{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ can be expressed in a particular coordinate system by the $n \times m$ Jacobian matrix

$$\boldsymbol{f}' = \left[\frac{\partial f_i}{\partial x_j} \right]. \tag{2.4}$$

Higher-order derivatives are defined in a similar way. For example, the second-order derivative $\boldsymbol{f}''$ can expressed by the tensor

$$\boldsymbol{f}'' = \left[\frac{\partial^2 f_i}{\partial x_j\, \partial x_k} \right] \tag{2.5}$$

The concept of continuity may be applied to the derivatives of $\boldsymbol{f}$ by making use of the operator norm

$$||\, T\, || = \max_{|\boldsymbol{x}| \leq 1} |\, T(\boldsymbol{x})\, |. \tag{2.6}$$

If the n-th order derivative of $\boldsymbol{f}$ exists and is continuous in an open set E of U_1 we write $\boldsymbol{f} \in C^n(E)$. A mapping $\boldsymbol{f} : E \mapsto \mathbb{R}^n$ belongs to $C^n(E)$ if and only if the n-th order partial derivatives $\partial^n f_i / \partial x_{j_1} ... \partial x_{j_n}$ exist and are continuous on E.

2.1.2 Integral Curves

The interest of vector fields mainly originates from the fact that they can be used to describe the dynamical evolution of a system in phase space, i.e., they give rise to a set of *integral curves* $\boldsymbol{x}(t)$. These curves are defined by an ordinary differential equation supplied with some proper initial value condition,

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \quad \text{with} \quad (\boldsymbol{x}_0, t_0) \in E \times J \tag{2.7}$$

Since at any point the curve $\boldsymbol{x}(t)$ is tangent to the vector field $\boldsymbol{f}$ integral curves are also called *tangent curves* of $\boldsymbol{f}$. Yet two other names are *trajectories* or *path lines*. These two names are motivated by the interpretation of a vector field as a flow field. In this case $\boldsymbol{x}(t)$ describes the path of a particle in the flow. Sometimes we may want to consider a vector field at a particular instance of time. The curves obtained by integrating Eq. (2.7) with fixed right hand side $\boldsymbol{f}(\boldsymbol{x}, t_0)$ are commonly called *field lines* or *stream lines*. For stationary vector fields stream lines and path lines coincide.

A point $\boldsymbol{x}_0$ with $\boldsymbol{f}(\boldsymbol{x}_0) = 0$ is called a *critical point* of a vector field. At such a point any particle will be at rest and Eq. (2.7) has the trivial solution $\boldsymbol{x}(t) = \boldsymbol{x}_0$.

In general, if the vector field $\boldsymbol{f}$ is continuous then there will be at least one solution of the initial value problem (2.7). This is stated by Peano's theorem. A unique solution exists, if $\boldsymbol{f}$ satisfies a so-called Lipschitz condition.

Definition 3. The mapping $\boldsymbol{f}(\boldsymbol{x}, t) : E \times J \mapsto \mathbb{R}^n$ is said to satisfy a Lipschitz condition on $E \times J$ with respect to $\boldsymbol{x}$ if there is a positive constant K such that for all $\boldsymbol{x}, \boldsymbol{y} \in E$ and $t \in J$

$$| \, \boldsymbol{f}(\boldsymbol{x}, t) - \boldsymbol{f}(\boldsymbol{y}, t) \, | \leq K \, | \, \boldsymbol{x} - \boldsymbol{y} \, |. \tag{2.8}$$

The mapping $\boldsymbol{f}$ is said to be *locally Lipschitz* with respect to $\boldsymbol{x}$ if for each point $(\boldsymbol{x}_0, t) \in E \times J$ there is an ε-neighborhood $N_\varepsilon(\boldsymbol{x}_0, t) \in E \times J$ so that a Lipschitz condition is satisfied in $N_\varepsilon(\boldsymbol{x}_0, t)$.

In particular, it can be shown that any continuously differentiable mapping, i.e., $\boldsymbol{f} \in C^1(E \times J)$, satisfies a Lipschitz condition. The fundamental existence-uniqueness theorem states:

Theorem 1. Suppose the mapping $\boldsymbol{f}(\boldsymbol{x}, t) : E \times J \mapsto \mathbb{R}^n$ is continuous and locally Lipschitz with respect to $\boldsymbol{x}$ for all $(\boldsymbol{x}, t) \in E \times J$. Then the initial value problem (2.7) has a unique solution $\boldsymbol{x}(t)$ which can be continued up to the boundary of $E \times J$.

Finally, we want to introduce a useful quantity to denote the solution of a given initial value problem formally, namely the *propagator* or *evolution* or *flow* of a vector field.

Definition 4. Suppose $\boldsymbol{f}(\boldsymbol{x}, t)$ to be locally Lipschitz on $E \times J$ and let $\boldsymbol{x}(\,\cdot\,; \boldsymbol{x}_0, t_0) \in C^1(J_{\text{max}}(\boldsymbol{x}_0, t_0), \mathbb{R}^n)$ be the solution of the initial value problem (2.7), where $J_{\text{max}}(\boldsymbol{x}_0, t_0)$ denotes the maximal time interval for which this solution is defined. Then the two-parametric family of mappings

$$\phi^{t,t_0} \, \boldsymbol{x}_0 = \boldsymbol{x}(t; \boldsymbol{x}_0, t_0) \tag{2.9}$$

is called the propagator of a vector field.

If the vector field $\boldsymbol{f}$ does not explicitely depend on time, the solution curves will only depend on the difference $t - t_0$, i.e., it doesn't matter when a curve has started, but only how long it has evolved. Thus, in this case we may omit the second parameter of the propagator and write ϕ^{t-t_0} instead of ϕ^{t,t_0}.

2.1.3 Other Curves and Surfaces

In the previous section we introduced the notion of stream lines and path lines. Beside these lines there are also other mathematical objects which can be used to analyse vector fields.

First of all, *streak lines* play an important role in fluid mechanics. These lines can be imagined as the trace of dye particles continuously being released into a fluid. Thus, streak lines can be directly observed in an experimental set-up. Mathematically a streak line $\boldsymbol{s}(t, \tau)$ with fixed t and $0 \leq \tau \leq t$ is defined as the locus of particles released from a fixed point x_0 during the time interval t, compare Figure 2.1 a). It is obtained by integrating

$$\frac{d}{dt}\boldsymbol{s}(t,\tau) = \boldsymbol{f}(\boldsymbol{s},t), \quad \boldsymbol{s}(\tau,\tau) = \boldsymbol{x}_0. \tag{2.10}$$

In a steady flow streak lines coincide with stream lines and path lines.

Another type of curve amenable to experimental measurements in fluid mechanics are *material lines* or *time lines*. A time line $\boldsymbol{m}(t,\sigma)$ with fixed t is the image of an initial curve $\boldsymbol{m}_0(\sigma)$ after propagation by the flow for a time t, compare Figure 2.1 b). Each point on a time line is transported in the usual way, i.e.,

$$\frac{d}{dt}\boldsymbol{m}(t,\sigma) = \boldsymbol{f}(\boldsymbol{m},t), \quad \boldsymbol{m}(0,\sigma) = \boldsymbol{m}_0(\sigma). \tag{2.11}$$

The visualization of a single curve does only reveal a limited amount of information about the vector field. Therefore in $\mathbb{R}^3$ often surfaces or volumes are investigated instead of lines. For example, *stream surfaces* are obtained by integrating a continuous set of stream lines at once. Similarly, streak lines could be released from a line or area source instead of just a single point. While these techniques are commonly used for visualization, the analysis of path lines or even path surfaces does not reveal much insight, since these objects readily become enourmously complex. However, a concept similar to time lines, namely the tracing of individual particles or whole volumes of particles, can be quite useful for analysing instationary data.

All curves and surfaces mentioned so far were related to the path of individual particles in the flow, i.e., they could be constructed by computing integral curves in a vector field. Beside these objects there are other mathematical entities which are of relevance in certain applications. These include vorticity lines, vortex sheets, critical points, separatrices, or periodic orbits. While the first two are commonly studied in fluid dynamics, the latter are related to vector field topology. Although we cannot discuss all details of vector field topology in this paper, in the following section we want to illustrate some general ideas by considering the special case of linear vector fields in $\mathbb{R}^2$ and $\mathbb{R}^3$.

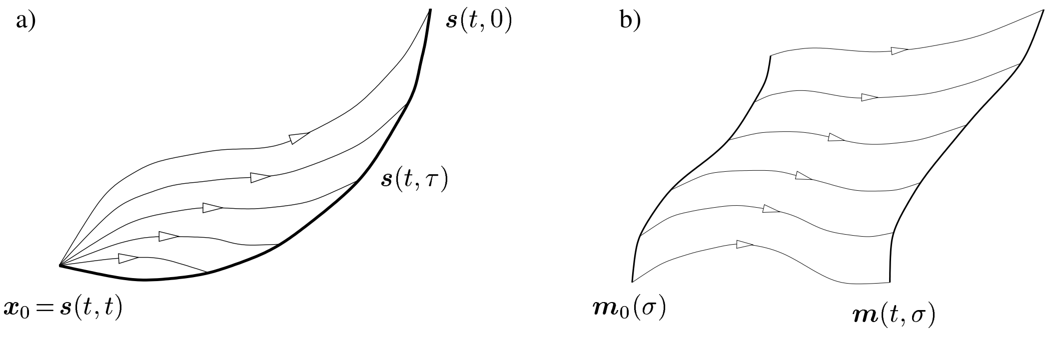


Figure 2.1: A streak line is defined as the locus of particles continously being released from a fixed point $\boldsymbol{x}_0$ (a). A time line is the image of an initial curve $\boldsymbol{m}_0(\sigma)$ after propagated by the flow.

2.2 Linear Systems

The integral curves of a linear vector field can be found analytically. Analysing the shape of these curves gives us an excellent summary about the possible topological configurations in general vector fields. In fact, in many cases the solution of a non-linear system around a critical point is topologically equivalent to the solution of the corresponding linearized system at that point. Mathematically this statement is expressed by the Hartman-Grobman Theorem [66].

After summarizing the general solution of a linear system, we will specifically consider vector fields in $\mathbb{R}^2$ and $\mathbb{R}^3$. Examples of different topological cases will be presented, again, by making use of the visualization techniques developed later in this work.

2.2.1 General Solution

A homogenuous linear stationary vector field is defined by a constant $n \times n$ matrix A, i.e., $\boldsymbol{f}(\boldsymbol{x}) = A\,\boldsymbol{x}$. Since for any such field the Lipschitz condition (2.8) holds, the corresponding linear system always has a unqiue solution for every $\boldsymbol{x} \in \mathbb{R}^n$. In order to specify this solution analytically, the exponential of the linear operator A need to be defined. This is done by

$$e^{A\,t} = \sum_{k=0}^{\infty} \frac{A^k\,t^k}{k!}, \quad t \in \mathbb{R}. \tag{2.12}$$

It turns out that the exponential of a linear operator can be differentiated in the same way as the expontential of a real number, i.e.,

$$\frac{d}{dt} e^{A\,t} = A\,e^{A\,t}. \tag{2.13}$$

With this background we can state the fundamental theorem for linear systems.

Theorem 2. Let A be an $n \times n$ matrix. Then for a given $\boldsymbol{x}_0 \in \mathbb{R}^n$ the initial value problem

$$\frac{d\boldsymbol{x}}{dt} = A\,\boldsymbol{x}, \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{2.14}$$

has the unique solution

$$\boldsymbol{x}(t) = e^{A\,t}\,\boldsymbol{x}_0. \tag{2.15}$$

In order to specify this solution more precisely we note that if A can be decomposed into PBP^{-1} with an invertable linear operator P, then

$$e^{A\,t} = \sum_{k=0}^{\infty} \frac{(PBP^{-1})^k\,t^k}{k!} = P\,e^{B\,t}\,P^{-1}. \tag{2.16}$$

Consequently, if A can be diagonalized the solution is given by

$$x(t) = P\,\mathrm{diag}[e^{\lambda_i t}]\,P^{-1}\,\boldsymbol{x}_0, \tag{2.17}$$

where λ_i are the (real) eigenvalues of A and the matrix P is build from the corresponding eigenvectors $\boldsymbol{v}_i$, i.e., $P = [\boldsymbol{v}_1 \cdots \boldsymbol{v}_n]$. However, in general A may not only have real eigenvalues but also pairs of complex conjugate ones. Moreover, the matrix may even be defective, i.e., there may not exist a set of n orthonormal eigenvectors. In both cases, A cannot be diagonalized. In this case, it is convenient to convert A into Jordan canonical form,

$$P^{-1}AP = \begin{bmatrix} B_1 & & \\ & \ddots & \\ & & B_r \end{bmatrix}. \tag{2.18}$$

The elementary Jordan blocks B_j are associated with the eigenvalues λ_j of A. For real eigenvalues λ and for pairs of complex conjugate eigenvalues $a \pm ib$ respectively, the Jordan blocks B_j are of the form

$$\begin{bmatrix} \lambda & 1 & & 0 \\ & \ddots & \ddots & \\ & & \lambda & 1 \\ 0 & & & \lambda \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} D & I & & 0 \\ & \ddots & \ddots & \\ & & D & I \\ 0 & & & D \end{bmatrix}, \tag{2.19}$$

with

$$D = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \quad \text{and} \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{2.20}$$

The matrix P in Eq. (2.18) is built from the generalized eigenvectors of A. A generalized eigenvector $\boldsymbol{v}$ of A is a solution of $(A - \lambda I)^k \boldsymbol{v} = 0$ with $1 \leq k \leq m$, where m denotes the multiplicity of the eigenvalue λ.

Instead of Eq. (2.17) the general solution of a linear system now can be written as

$$x(t) = P\,\mathrm{diag}[e^{B_j t}]\,P^{-1}\,\boldsymbol{x}_0. \tag{2.21}$$

By analysing the exponentials of the elementary Jordan blocks one finds that the solution will be a linear combination of functions of the form

$$t^k\,e^{at}\,\cos bt \qquad \text{or} \qquad t^k\,e^{at}\,\sin bt, \tag{2.22}$$

where $\lambda = a + ib$ is an eigenvalue of the matrix A and $0 \leq k \leq n-1$. If the matrix possesses a full set of ordinary orthonormal eigenvectors, all k will be zero and the polynomials terms in t vanish.

An important concept is the subdivision of $\mathbb{R}^n$ into stable, unstable, and center subspaces of the linear system. Stable subspaces are spanned by generalized eigenvectors corresponding to real negative eigenvalues. In these subspaces the solution decays exponentially and approaches the origin. Similarly, unstable subspaces are spanned by generalized eigenvectors corresponding to real positive eigenvalues, while center subspaces are spanned by generalized eigenvectors corresponding to complex eigenvalues. In center subspaces the solution spirals around the origin as suggested by the sines and cosines in Eq. (2.22). The sign of the real part of a corresponding eigenvalue determines whether it moves towards the origin or away from it.

2.2.2 Topological Classification in 2D

We now want to consider the special case of 2D linear systems:

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{2.23}$$

Here the dot denotes differentiation with respect to t. The eigenvalues of a $n \times n$ matrix A are obtained by solving the characteristic equation $\det(A - \lambda I) = 0$. For a 2×2 matrix this is a quadratic equation and the solutions are given by

$$\lambda_{1,2} = \frac{1}{2}\left(\tau \pm \sqrt{\tau^2 - 4\delta}\right), \tag{2.24}$$

with

$$\tau = a_{11} + a_{22} = \mathrm{tr}A \quad \text{and} \quad \delta = a_{11}a_{22} - a_{12}a_{21} = \det A. \tag{2.25}$$

Depending on the values of τ and δ the eigenvalues of A may be real or complex, leading to a particular type of integral curves. In the following we shall dicuss the different topological cases in detail. Figure 2.2 presents a summary of this discussion.

Saddles. If $\delta < 0$ the matrix A has two real eigenvalues of opposite sign. This means that the solution is exponentially decaying in the direction of one eigenvector while it is exponentially increasing in the direction of the other. The situation is shown in Figure 2.2 (f) and (g). The system is said to have a *saddle* at the origin. In the direction of the eigenvectors there are four trajectories approaching the origin as $t \to \pm\infty$. These trajectories are called *separatrices* of the system.

Nodes. If $0 < 4\delta < \tau^2$ the matrix A either has two real negative eigenvalues ($\tau < 0$) or two real positive eigenvalues ($\tau > 0$). The origin is said to be a *node*. The solution curves are either exponentially approaching it or are moving away from it, c.f. Figure 2.2 (e). In the first case the node is called a *sink* while in the second case it is called a *source*. Moreover, a critical point which has only real eigenvalues, i.e., which is either a saddle or a node, is called a *hyperbolic critical point*.

If $4\delta = \tau^2$ the system has two identical eigenvalues. If the matrix can be diagonalized, this corresponds to a symmetric star-shaped node, c.f. Figure 2.2 (d). However, if the matrix is defective, in one direction a term proportional to $t\, e^{\lambda t}$ arises. The flow pattern then becomes twisted as illustrated in Figure 2.2 (c). The origin is called a *logarithmic critical point*.

Foci. If $4\delta > \tau^2$ the matrix A has a pair of complex conjugate eigenvalues, i.e., $\lambda = a \pm ib$. If the real part a doesn't vanish the system is said to have a *focus* at the origin, c.f. Figure 2.2 (b). If $a < 0$ the solution approaches the origin on a spiral path. If $a > 0$ it moves away from it. Therefore, like for nodes the line $\tau = 0$ separates foci into *stable* ones and *unstable* ones.

Centers. In the special case $\tau = 0$ and $\delta > 0$ the matrix A has a pair of pure imaginary eigenvalues. The solution of the corresponding linear system is comprised by closed *periodic orbits*. The system is said to have a *center* at the origin. This situation is illustrated in Figure 2.2 (a).

2.2.3 Topological Classification in 3D

In its individual subspaces the solution of a higher-dimensional homogeneous linear system will exhibit the same topological behaviour we already know from $\mathbb{R}^2$. For example, the vector field might show a circular motion in a plane defined by two eigenvectors belonging to a pair of pure imaginary eigenvalues. Perpendicular to this plane the flow might be exponentially decaying or increasing. In this section we want to classify the solutions for the important class of 3D homogeneous linear vector fields. As outlined in [12] such a classification helps to interprete the structure of more complex general 3D vector fields. Let us consider the following linear system

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}. \tag{2.26}$$

The characteristic equation $\det(A - \lambda I) = 0$ can be written as

$$\lambda^3 + P\,\lambda^2 + Q\,\lambda + R = 0, \tag{2.27}$$

with

$$P = -(a_{11} + a_{22} + a_{33}) = -\mathrm{tr}\,A, \tag{2.28}$$

$$Q = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} + \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} + \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}, \tag{2.29}$$

$$R = -\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = -\det A. \tag{2.30}$$

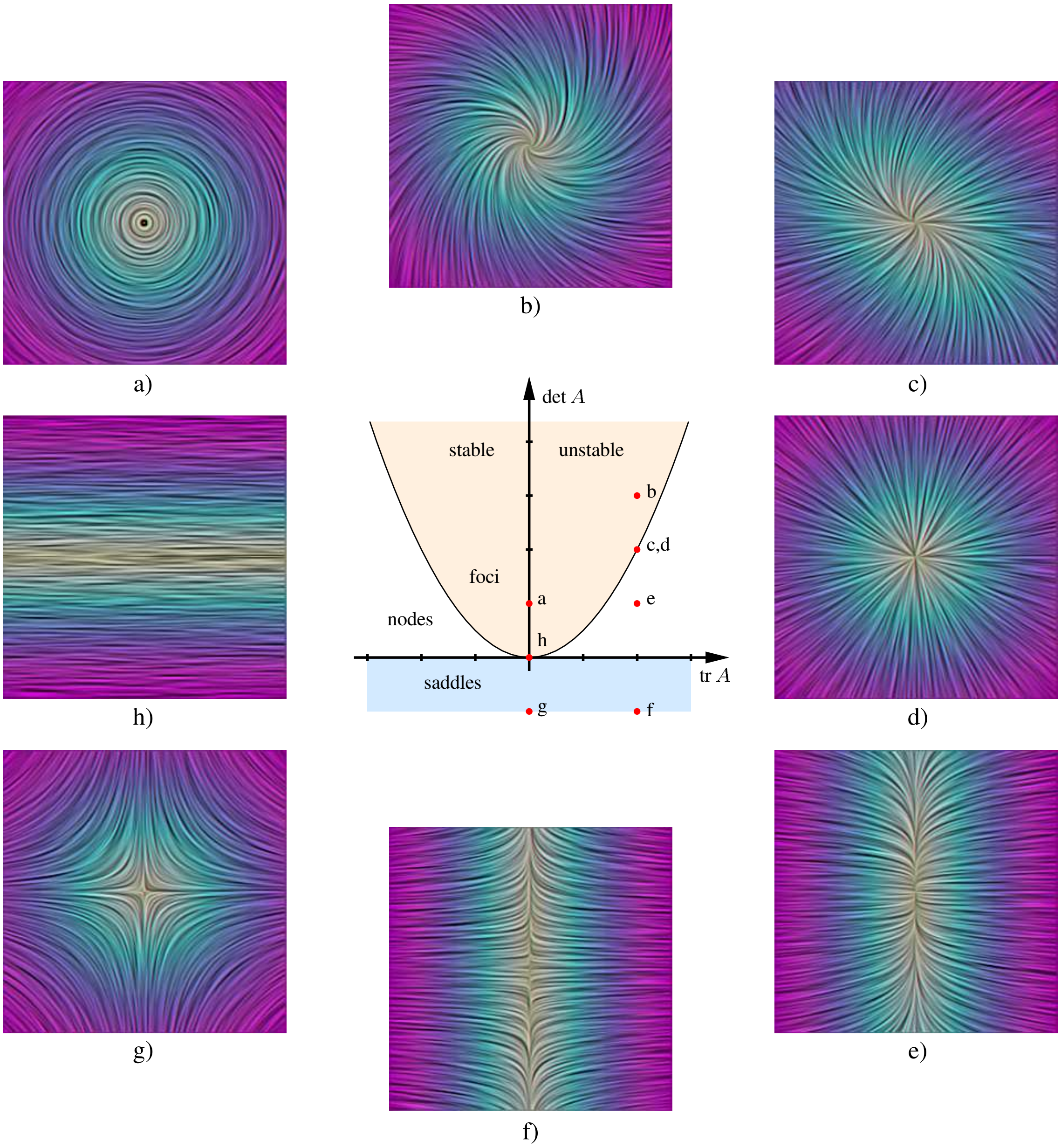


Figure 2.2: Solutions of a homogeneous linear system $\dot{x} = A\,x$ in $\mathbb{R}^2$ can be classified into saddles, nodes, foci, and centers. The topology of a solution is determined by the eigenvalues of A. For example, real eigenvalues of opposite sign correspond to a saddle (g,f). Different solution classes can be depicted in a diagram defined by tr A and det A.

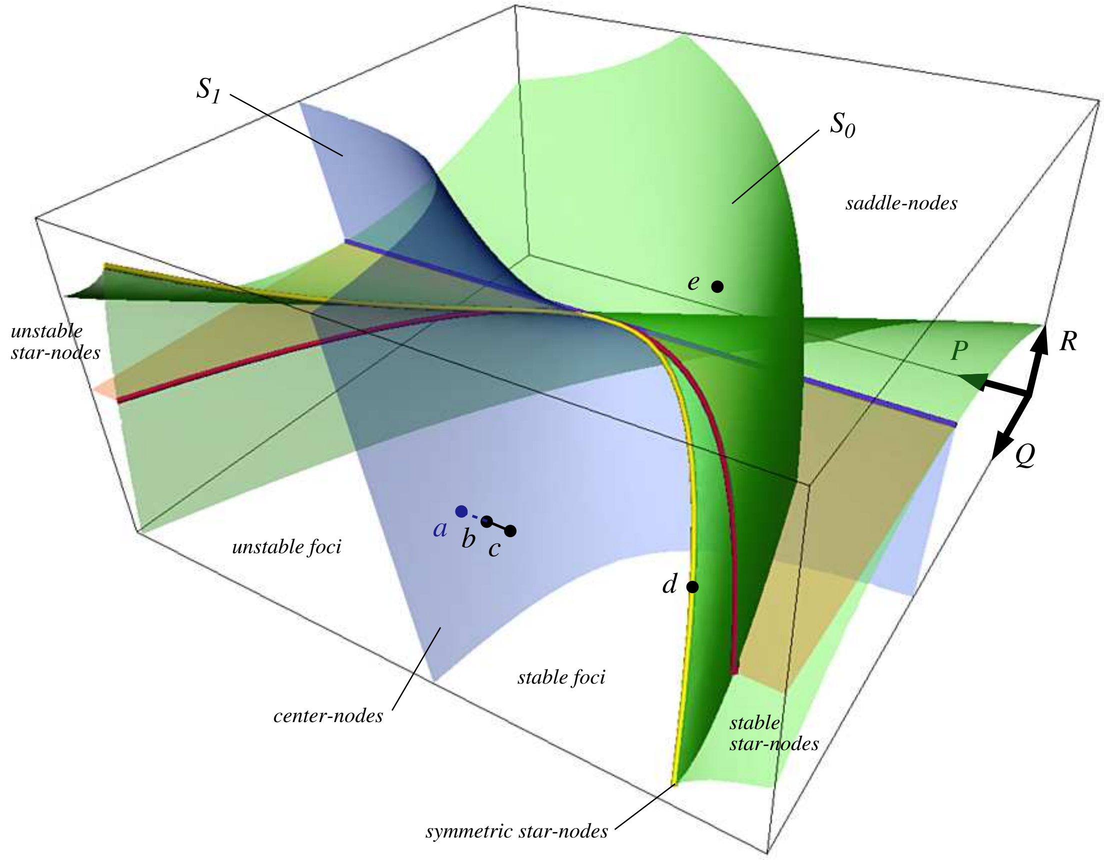


Figure 2.3: Configuration diagram for homogeneous linear systems in $\mathbb{R}^3$. The axes correspond to the coefficients P, Q, and R of the characteristic equation (2.27). The points a to e indicate the position of the topological configurations shown in Figure 2.4.

The roots of the cubic equation (2.27) can be expressed in closed form by means of Cardano's formula [7]. The sign of the discrimination number

$$D = 27R^2 + (4P^3 - 18PQ)R + (4Q^3 - P^2Q^2) \tag{2.31}$$

determines whether the roots are all real or whether complex roots occur. If $D > 0$ there are three real distinct roots. If $D = 0$ there are also three real roots, but at least two of them are equal. Finally, if $D < 0$ there will be one real root and a conjugate pair of complex roots. The condition $D = 0$ defines a surface S_0 in PQR-space separating the real solutions from the complex ones. This surface is shown in Figure 2.3, where all possible configurations are summarized.

Symmetric Star-Nodes. The surface S_0 exhibits a cusp for $3Q = P^2$ and $27R = P^3$. On this curve there are three identical real roots $\lambda = -P/3$. Provided the matrix can be diagonalized, this corresponds to a symmetric star-shaped node. An example is shown in Figure 2.4 d).

Star-Nodes. If $D > 0$, $Q > 0$, and $PR > 0$ there are three real eigenvalues of equal sign. Thus field lines are either approaching the origin from all directions ($P < 0$) or are

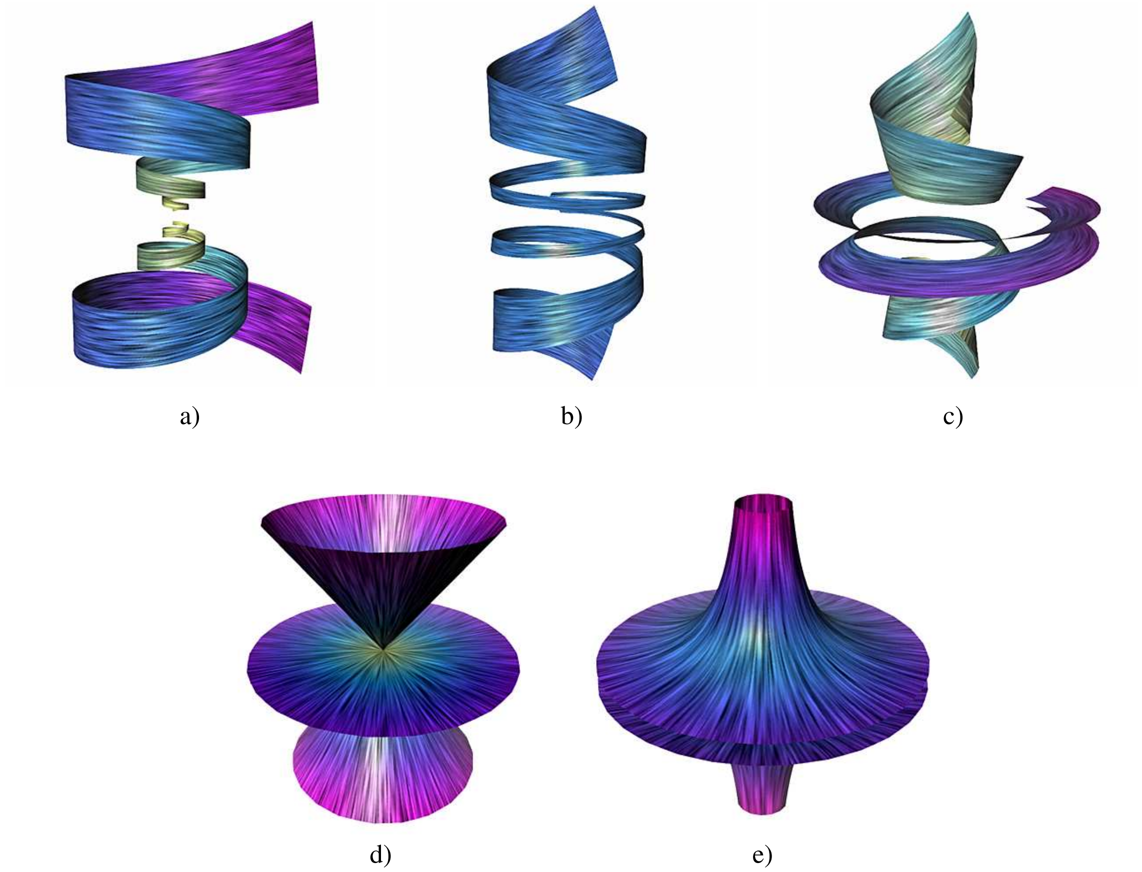


Figure 2.4: Typical topologies of homogeneous linear vector fields in $\mathbb{R}^3$ are a stable focus (a), a center (b), an unstable focus (c), a star node (d), and a saddle node (e). In each case stream surfaces in a corresponding vector field are displayed. The flow on a surface is visualized by means of a LIC texture.

moving away from it ($P > 0$). The two regions in configuration space corresponding to stable and unstable star-nodes are indicated in Figure 2.3.

Saddle-Nodes. If $D > 0$, but $Q < 0$ or $PR < 0$, then there are three real eigenvalues of opposite sign, i.e., two positive ones and a negative one or vice versa. The flow exhibits a node-type topology in the plane spanned by the two eigenvectors corresponding to equal sign. In the planes spanned by the other two eigenvector combinations a saddle-type topology arises. An example is shown in 2.4 e).

Foci. If $D < 0$ a conjugate pair of complex eigenvalues $\lambda = a \pm ib$ occurs. Therefore in the plane spanned by the corresponding eigenvectors the flow is spiraling around the origin. The sign of the real part a determines whether it approaches the origin or whether it moves away from it. Both cases are separated by the surface $PQ - R = 0$, denoted S_1 in Figure 2.3. Two examples for positive and negative real part are shown in Figure 2.4 a) and c).

Center-Node. If $D < 0$ and $PQ - R = 0$ the real part of the two complex eigenvalues vanishes. In this case the flow exhibits a true center-type topology in the corresponding eigenvector plane, c.f. Figure 2.4 b). Perpendicular to this plane the flow may either decay or increase exponentially.

Degenerate Cases. If $R = 0$ at least one eigenvalue will be zero. We then obtain just the same configurations as in the 2D case. In fact, we can interprete a 2D matrix as a 3D matrix with $a_{i3} = a_{3j} = 0$. Then $P = -\tau$ and $Q = \delta$ as defined in Eq. (2.25). Therefore the curve $4Q = P^2$, $R = 0$ is equivalent to the parabola from Figure 2.2, separating real-valued solutions from complex ones in $\mathbb{R}^2$.

2.3 Survey of Vector Field Visualization Methods

So far we introduced various types of curves and surfaces in a vector field and we gained insight into the topology of two- and three-dimensional fields. We are now in a position to review several existing approaches to vector field visualization. Primiarily we distinguish between *local* and *global* methods. While in local methods directional properties at different locations are visualized directly and independently from each other, in global methods some kind of field line integration process is performed. Global methods differ in what kind of mathematical curves or surfaces are computed and how these objects are represented graphically. After considering "standard" visualization methods we focus on texture-based techniques. Such methods may either be local or global ones. Finally we shortly address visualization of instationary data in a separate section. We try to judge different approaches with respect to the three design goals mentioned in the introduction, accuracy, performance, and cognition.

2.3.1 Local Methods

Arrow Plots. Arrows probably provide the most simple way for vector field visualization. An arrow can be used to depict field direction at a unique point. Proper scaling and coloring allows one to encode a scalar quantity, e.g. vector magnitude, as well. The main advantages of arrow plots are that they are easy to implement and easy to interprete. On the other hand, they are restricted to a rather coarse spatial resolution. If individual symbols overlap each other the image rapidly gets cluttered. Even more than in 2D in 3D it is important to restrict the placement of arrows to certain regions. For examples, often a set of two-dimensional slices of a 3D field is extracted. In general arrows are not well suited to resolve small details of a field.

Another potential problem with arrows are artifacts introduced by a regular arangement of the individual symbols. Patterns caused by such an arangement may distract from the structure of the vector field itself, thus making the image quite unintuitive. In order to avoid such problems Dovey proposed to distribute symbols in a jittered way [25]. Turk and Banks [87] suggest a similar strategy. By means of a relaxation algorithm they even try to find an optimized placement of arrows. Thus overlapping symbols as well as large empty regions are effectively avoided.

Vector Field Probes. A straight-forward extension of simple arrows has been proposed by de Leeuw and van Wijk [19]. Their *vector field probe* not only indicates field direction and magnitude, but also properties obtained from a first order field expansion. At a particular point the first derivative $\partial f_i / \partial x_j$ is computed. This matrix is decomposed into characteristic components like curvature and torsion, which are then represented by a set of geometric symbols.

An example of the de Lueew–van Wijk probe is shown in Figure 2.5. Because of its size and geometric complexity only few icons can be displayed in a single image. In order to investigate global features of the field some kind of mental field line integration or extrapolation has to be performed. This makes the method quite unintuitive and inaccurate. Like with standard arrows problems can occur if geometric scaling is used to indicate vector field magnitude. For example, electrostatic fields may contain point charges where magnitude diverges. In this case logarithmic scaling might be used or field strength might be encoded by color.

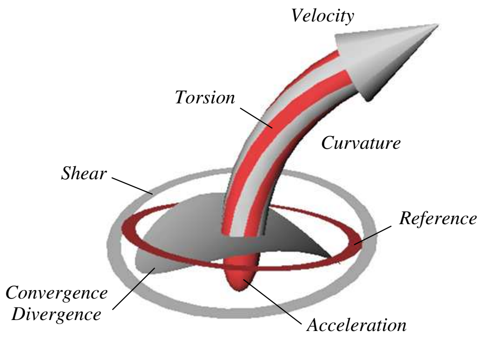


Figure 2.5: The vector field probe of de Lueew and van Wijk depicts characteristic components of the first derivative $\partial f_i / \partial x_j$.

Another type of vector field icon has been designed by Rumpf and Happe [72]. They determine critical points in a vector field automatically and indicate position and topological type of these points by means of specifically designed symbols.

Color Coding. Color coding is a standard technique for visualizing scalar quantities. However, color can also be used to depict directional information. This is most obvious for vector fields in $\mathbb{R}^2$. It is a well-known matter of fact that color space has three dimensions. One way to parametrize this space is to decompose colors into hue, saturation, and luminance. For constant saturation and luminance colors can be arranged on a color circle (red-yellow-green-cyan-blue-magenta-red). Thus each color uniquely defines an angle or vector direction. In addition, the magnitude of a vector can be encoded by varying saturation or luminance. Color-coded directional images are fast to compute and can be used to resolve fine details of a field. However, they are not very intuitive and are usually difficult to interprete.

A similar technique has been proposed for visualizing three-dimensional vector fields. A standard way for visualizing three-dimensional scalar fields is so-called direct volume rendering [39]. Each point in space is assigned some light emission and light absorption coefficient. Then light transport inside the volume is simulated. Though computationally expensive, the resulting cloudy image can reveal some insight into the data. Uselton [88] mapped the x-, y-, and z-components of a vector field to the red, green, and blue components of color. However, often the colors accumulate to medium grey and the directional structures of the field is hard to understand. Frühauf used a reflective illumination model for raycasting vector fields [33]. He mapped the angle between the tangent vector and some light vector to color. Its is argued that vectors pointing towards or away from the light source can be identified by rotating the whole volume and observing the corresponding color changes.

Yet another vector field visualization technique based on color coding has been proposed by Theisel [86]. He showed that a vector field in $\mathbb{R}^2$ or on a 2D manifold can be expressed by two scalar curvature fields which can be found analytically. Visualizing both of these scalar fields by means of pseudo-coloring reveals information about the vector field itself. For example, discontinuities in the field's first derivative become readily visible. However, again the overall flow structure of the field is hard to understand.

2.3.2 Global Methods

Stream Lines. While arrows or icons as well as all color coding techniques only reveal local information of a vector field stream lines encode global information. Integral curves like stream lines have been used to analyse vector fields since the introduction of the field concept in science by Faraday more than hundred years ago. Not surprisingly, many computer-based visualization techniques for stationary vector fields also rely on the display of stream lines. In particular this is true if fields in $\mathbb{R}^3$ are to be depicted. The various techniques mainly differ in the way how individual curves are rendered or how seed points for the curves are determined.

High quality images can be obtained if individual curves are represented by small cylindrical tubes. These tubes can be illuminated by taking into account ambient, diffuse, and specular light reflection. In this way spatial impression of the resulting images can be improved significantly. Usually tubes are created by extruding an n-sided polygon along the path [17]. The polygons might also be scaled or rotated in order to depict the local deformation characteristics of the vector field [76]. Stream ribbons are a variation of this concept. Brill et. al. [6] positioned so-called stream balls along a stream line. The stream balls define an implicit function which can be used to approximate stream tubes. Individual stream balls may merge or split along a curve. In this way velocity information can be depicted in an intuitive way. Although nice images can be obtained, representing field lines polygonally is computationally quite expensive. Therefore only a very limited number of lines can be displayed at interactive frame rates on a standard graphics workstation. As an alternative simple scan-converted lines, i.e., truely one-dimensional primitives, can be drawn. In order to combine the efficiency of simple line rendering with the perceptional benefits of an illuminated polygonal scene we developed a hardware-accelerated technique for illuminating one-dimensional lines correctly [83].

As the term "stream line" already indicates many visualization techniques based on the display of integral curves were developed in the context of flow visualization. However, most of these methods can be applied to general vector fields as well. For example, MacLeod, Johnson, and Matheson [58] report about the visualization of bioelectrical fields. In the general case stream lines better should be called field lines.

Stream Surfaces. Stream surfaces are similar to stream ribbons. They are defined by a dense set of neighbouring stream lines. In contrast to stream ribbons stream surfaces usually have a larger extent perpendicular to the flow. They provide intuitive means for revealing structures in a 3D vector field, but are more difficult to compute compared to simple stream lines.

Hultquist [44] described an advancing front algorithm based on tracing inidvidual stream lines in a vector field. Between neighbouring lines triangles are created as necessary. The algorithm keeps track of diverging and converging surface parts. If necessary, additional stream lines are inserted or stream lines are terminated. We will investigate this algorithm in detail in Section 5.1.

If a vector field has vanishing divergence then a scalar-valued stream function can be defined. The gradient of this function is always perpendicular to the flow. An iso-contour of a stream function represents a stream surface. Noting that any linear combination of two stream functions is again a stream function, van Wijk [89] proposed to generate stream surfaces by determining a dual pair of stream functions, superposing these functions in a suitable way, and finally extracting stream surfaces by means of a standard isosurface algorithm such as marching cubes [54]. Moreover, for certain kinds of incompressible flows principal stream surfaces can be defined [11]. Such surfaces are perpendicular to the curvature vector of a stream line at every point. They are supposed to be especially nice and informative. Thus they can be used to simplify the problem of defining suitable constraints or initial values for stream surface generation. Often such constraints had to be defined manually. We reconsider this point in Section 5.1, too.

Eventually, Löffelmann et. al. [53] suggested to improve display of ordinary stream surfaces by making arrow-shaped parts of the surface semi-transparent. In this way the direction of the flow within the surface becomes more clear. In addition it is possible to investiagte inner parts of a curly shaped stream surface which would be occluded otherwise.

Topological Analysis. An alternative global approach to vector field visualization relies on the extraction of topological features such as critical points and separatrices. Recall that a separatrix is a line (in 2D) or a surface (in 3D) separating different hyperbolic sectors of a saddle-type critical point. Separatrices connect critical points as well as attachment and detachment points on boundaries. They define the *skeleton* of a vector field, thus summarizing the topology of the field. For example, recirculation areas are easily identified by closed separatrices. Helman and Hesselink [43, 42] describe methods for automatically extracting the topology of 2D and 3D flow fields. Globus, Levit, and Lasinski developed a tool for identifying and classifying critical points in vector fields [35]. Scheuermann et. al. [74] extended these methods for the analysis of higher order critical points. At these points linear terms in a field expansion vanish, i.e., one or more eigenvalues of the linearized field will be zero.

Another important topological feature of flow fields are vortices or vortex lines. A vortex line is loosely defined as the center of a swirling flow region. A number of different algorithms have been developed to detect such features in a vector field. Among the more popular approaches is a method due to Banks and Singer which relies on integrating stream lines in a vortex core [2]. In order to make stream line tracing more stable after each step the current position is corrected by determining a local pressure minimum.

Sujudi and Haimes identify vortex lines by determining flow regions where just one real eigenvalue occurs and the associated eigenvector is oriented parallel to the flow [84].

Visualization methods based on the extraction of topological features can be very powerful. However, they require a sound understanding of the nature of the vector field being investigated. Not all critical points or vortex lines are relevant in practice. For example, many of them simply are noise artifacts. Probably it is a good idea to combine topological methods with visualization approaches aiming on a more intuitive display of flow structures.

2.3.3 Texture-based Methods

Standard methods for vector field visualization are based on the display of arrow-like symbols or stream lines. We already pointed out that a common problem with these methods, both local and global ones, is limited spatial resolution. In order to obtain an intuitive image revealing as much information of a vector field as possible many symbols or many stream lines should be used. However, if these objects are placed too densely, the image rapidly gets cluttered and no information at all is revealed anymore. To tackle this problem so-called texture-based visualization methods have been proposed. The idea is to synthesize a texture or a textured surface in order to densely represent the directional information contained in a vector field. In principle the resolution of a directional texture is only limited by the size of a single pixel.

The first texture-based method designed for vector field visualization was the spot noise technique conceived by van Wijk in 1991 [90]. The method generates a directional texture by superposing individual spots oriented according to local vector field direction. Since no integral curves are computed spot noise is a local visualization method. However, for texture-based methods the distinction between local and global methods is not as significant as for other methods. In fact, enhancements of spot noise have been proposed which actually do compute integral curves [20]. In this paper we focus on a technique called line integral convolution or LIC. First introduced by Cabral and Leedom in 1993 [9], LIC is closely related to spot noise. Details will be discussed in Chapter 4. In contrast to original spot noise LIC relies on computing stream lines of a vector field, thus making the visual results more accurate. Although from the cognitive point of view LIC is very favourable the method is computationally quite expensive. Therefore techniques for efficiently synthezising directional textures as described in this paper are very important. They make texture-based methods almost an ideal tool for vector field visualization.

Since its introduction in 1993 LIC has been extended in a number of ways. These extensions include animation of LIC textures in order to reveal the sign of vector field direction as well as vector magnitude [9, 81, 31, 48], design of parallel implementations [10, 96], application of LIC on surfaces [31, 85, 3, 59], or direct volume rendering of 3D LIC textures [78, 45]. Some of these issues will be discussed in detail later on. For a review of the other topics we refer to the SIGGRAPH'97 course notes on *Texture Synthesis with Line Integral Convolution*.

Beside spot noise and line integral convolution there are a number of other visualization techniques which also produce some kind of directional texture. In a technique called FROLIC [93] individual strokes or droplets are rendered like in spot noise. However, these strokes are not symmetric and therefore allow one to recognize the sign of vector field direction more easily. Max et. al. [61] encoded 3D directional information near contour surfaces by rendering stochastically distributed line bundles or semi-transparent ellipses. The texture splats technique by Crawfis and Max [16] can be used for direct volume rendering of vector fields. The method works by rendering a large number of oriented non-isotropic semi-transparent polygons. Compared to line integral convolution in all these methods the correlation length of the directional textures being produced is limited. This makes these methods somewhat less accurate since it is harder to visually trace individual stream lines along a larger distance.

2.3.4 Instationary Visualization Methods

Visualization methods based on the display of stream lines or stream surfaces are hard to apply for instationary data. This is also true for texture-based techniques such as LIC. The reason is that correlation between stream lines in two successive time steps is lost. Small changes of the vector field result in big displacements of a stream line. Therefore, in order to analyse time-dependent data either streak lines or path lines are computed. The display of streak lines is quite intuitive and very similar to dye injection experiments in flow visualization. In contrast, path line patterns are much harder to interprete. Therefore, often individual particles are simply traced in time rather than showing their accumulated trajectory. By releasing particles continuously essentially a streak line visualization is obtained.

Particle tracing systems have a number of applications in computer graphics and animation. Reeves discribes an early system for geneating the illusion of fire or water falls [71]. Similar techniques can also be used in scientific visualization. Many computer-based flow visualization systems like NASA's virtual windtunnel software [8], the Flow Analysis Software Toolset (FAST) [1], or the pv3 system [36] provide particle tracing modules. Usually, individual particles are rendered as points or arrows.

Van Wijk [91] describes an interesting variation of a particle system. Instead of rendering point primitives he uses little triangles aligned in flow direction. The triangles are illuminated by a light source. In this way for stationary vector fields the illusion of stream surfaces can be obtained at low computational costs.

In another class of path tracing algorithms individual streak lines or particles aren't rendered at all. Instead, dense volumes are being displayed using direct volume rendering methods. Examples are Volume Seeds [56], Virtual Smoke [57], or Flow Volumes [60, 4]. It is argued that methods based on volume rendering cannot miss interesting features of a vector field as easily as if single streak lines are being displayed. Another advantage is, that an additional scalar quantity like pressure can be displayed in an intuitive way by adjusting the opacity of the volume.

Chapter 3

Underlying Numerical Methods

In this chapter we want to discuss some basic numerical algorithms which we will utilize later on. In particular, we will consider the problem of numerically integrating field lines in a vector field, as well as methods for representing and evaluating vector fields on a computer. Special emphasis will be put on requirements imposed by our particular visualization algorithms. For example, the computation of continuously defined solutions of an ordinary differential equation will be an issue (dense output), as well as straight line approximations, treatment of discontinuous vector fields, or fast evaluation of vector fields defined on discrete grids.

3.1 Field Line Integration

Given an initial value $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$, how can we numerically compute an integral curve $\boldsymbol{x}(t)$ in a vector field? In the literature a great variety of algorithms is known for solving this kind of problem. Many of the advanced methods either make special assumptions about the field, e.g., assume that the integrand can be continuously differentiated several times, or are devoted to special cases like for example stiff problems. For an overview we refer to standard textbooks like Deuflhard and Bornemann [21] or Hairer, Nørsett, and Wanner [37]. In our context vector fields are often defined on a discrete grid with some low-order (often linear) interpolation scheme being applied inside the grid cells. In this situation many high-order methods do not really pay. For this reason we have chosen to utilize simple and robust integrators of Runge-Kutta type. Equipped with error monitoring and adaptive step size control these methods turn out to be an excellent choice for our purposes. In addition, they allow us to construct continuously defined interpolants in a simple way. Before we discuss specific methods in detail, let us introduce the basic concept of a discrete numerical integrator.

3.1.1 Discrete Approximations

Our goal is to find a solution $\boldsymbol{x}(t)$ of the initial value problem

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \quad \text{with } (\boldsymbol{x}_0, t_0) \in E \times J. \tag{3.1}$$

In order to simplify the notation let us assume $t_0 = 0$. We further want to to remove the explicit time dependence of $\boldsymbol{f}(\boldsymbol{x}, t)$ on the right hand side. Formally, this can be achieved by turning Eq. (3.1) into the following system:

$$\frac{d}{dt}\begin{pmatrix} \boldsymbol{x} \\ s \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}, s) \\ 1 \end{pmatrix}, \quad \begin{pmatrix} \boldsymbol{x}(0) \\ s(0) \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_0 \\ 0 \end{pmatrix} \tag{3.2}$$

In contrast to Eq. (3.1) this equation represents an *autonomous* system of ODE's. Silently incorporating the variable s into the state vector $\boldsymbol{x}$ and introducing $\Omega = E \times J$, we may denote it for short by

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \quad \text{with } \boldsymbol{x}_0 \in \Omega. \tag{3.3}$$

From Section 2.1.2 recall that the solution of Eq. (3.3) can be expressed by the evolution ϕ^t. This operator describes how a point $\boldsymbol{x}_0$ evolves in time:

$$\phi^t \boldsymbol{x}_0 = \boldsymbol{x}(t), \quad \left.\frac{d}{dt}\phi^t \boldsymbol{x}\right|_{t=0} = \boldsymbol{f}(\boldsymbol{x}_0). \tag{3.4}$$

In general, it is not possible to determine ϕ^t exactly. However, given some local approximation $\hat{\phi}^t$ we can try to find an approximate solution $\hat{\boldsymbol{x}}$ iteratively. Usually, $\hat{\boldsymbol{x}}$ will be defined on a discrete grid $\{t_i\}$, i.e.,

$$\hat{\boldsymbol{x}} : \{t_i\} \mapsto \mathbb{R}^n, \quad \hat{\boldsymbol{x}}(t_i) \equiv \boldsymbol{x}_i \approx \boldsymbol{x}(t_i). \tag{3.5}$$

In the following we want to concentrate on one-step methods. For these methods the value $\boldsymbol{x}_{i+1}$ solely depends on $\boldsymbol{x}_i$. Consequently we obtain a two-term recursive relationship:

$$\boldsymbol{x}_{i+1} = \hat{\phi}^{h_i} \boldsymbol{x}_i, \quad i = 0, 1, 2, ..., N. \tag{3.6}$$

As indicated by the subscript, the time increment $h_i = t_{i+1} - t_i$ between sucessive points $\boldsymbol{x}_{i+1}$ and $\boldsymbol{x}_i$ may vary.

The local error of a discrete evolution is defined by

$$\epsilon_i = \phi^h \boldsymbol{x}_i - \hat{\phi}^h \boldsymbol{x}_i. \tag{3.7}$$

An integration method is said to be of order p if its local error is of order $O(h^{p+1})$. The Taylor series expansion of the exact solution around $\boldsymbol{x}$ is given by

$$\phi^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f} + \tfrac{1}{2}h^2 \boldsymbol{f}'\boldsymbol{f} + \tfrac{1}{6}h^3\left(\boldsymbol{f}''\boldsymbol{f}\,\boldsymbol{f} + \boldsymbol{f}'\boldsymbol{f}'\boldsymbol{f}\right) + O(h^4). \tag{3.8}$$

Here, the elementary differentials of $\boldsymbol{f}$ are to be evaluated at $\boldsymbol{x}$. The most simple discrete evolution is given by Euler's method,

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f}(\boldsymbol{x}). \tag{3.9}$$

Comparing this expression with the Taylor series expansion of the exact solution reveals that the leading error term is of order $O(h^2)$. Therefore, Euler's method is of order 1.

Before we discuss how to construct methods of higher order, let us shortly mention an important theorem regarding the convergence of a discrete evolution. For a detailed discussion we refer to [21].

Theorem 3. Assume the initial value problem (3.3) has a unique solution $\boldsymbol{x}(t)$ and the discrete evolution $\hat{\phi}^h \boldsymbol{x}$ can be expressed in terms of an increment function ψ,

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\psi(\boldsymbol{x}, h). \tag{3.10}$$

If $\psi(x, h)$ is locally Lipschitz with respect to the state vector $\boldsymbol{x}$, then for $h \to 0$ the sequence of points $\{\boldsymbol{x}_i\}$ defined by Eq. (3.6) converges to $\boldsymbol{x}(t)$, i.e., the discretization error $||\,\epsilon\,||_\infty = \max|\,\epsilon_i\,|$ approaches zero.

From Eq. (3.10) it follows that the discrete evolution obeys

$$\frac{d}{dh}\hat{\phi}^h \boldsymbol{x}\bigg|_{h=0} = \boldsymbol{f}(\boldsymbol{x}_0). \tag{3.11}$$

Of course, the same expression is also true for the exact evolution, compare Eq. (3.4). A discrete evolution is said to be *consistent* if it fullfills this kind of minimal requirement.

3.1.2 Runge-Kutta Methods

In order to improve the order of a single Euler step we can take an intermediate step, evaluate the right hand side at this intermediate location, and compute a solution as

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f}(\boldsymbol{x} + \tfrac{1}{2}\,h\,\boldsymbol{f}(\boldsymbol{x})). \tag{3.12}$$

This method, proposed by Runge in 1895 [73], is known as midpoint rule. Its Taylor series expansion is

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + h\boldsymbol{f} + \tfrac{1}{2}\,h^2 \boldsymbol{f}'\boldsymbol{f} + \tfrac{3}{24}\,h^3 \boldsymbol{f}''\boldsymbol{f}\boldsymbol{f} + O(h^4). \tag{3.13}$$

Comparing this with Eq. (3.8) shows that the method is of order 2. In particular, the local error is given by

$$\epsilon = \phi^h \boldsymbol{x} - \hat{\phi}^h \boldsymbol{x} = \tfrac{1}{24}\,h^3(\boldsymbol{f}''\boldsymbol{f}\boldsymbol{f} + 4\boldsymbol{f}'\boldsymbol{f}'\boldsymbol{f}) + O(h^4) \tag{3.14}$$

By taking more intermediate steps and combining all these to obtain the final solution, methods of higher order can be constructed. This is the basic idea of so-called explicit Runge-Kutta methods [51]. Given a set of coefficients a_{ij}, s derivatives are computed,

$$\boldsymbol{k}_i = h\boldsymbol{f}(\boldsymbol{x} + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j), \quad i = 1, ..., s. \tag{3.15}$$

The intermediate $\boldsymbol{k}_i$ are combined to give the final solution, i.e.,

$$\hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + \sum_{i=1}^{s} b_i \, \boldsymbol{k}_i. \tag{3.16}$$

Note, that the coefficients b_i must satisfy $\sum_i b_i = 1$ in order for Eq. (3.10) to be fulfilled. Similarly, there are other so-called order conditions, which must also be fulfilled if error terms of higher order are to vanish.

A well-known example established in this way is the classical fourth-order Runge-Kutta formula, which requires four evaluations of the right hand side:

$$\left.\begin{aligned} \boldsymbol{k}_1 &= h\boldsymbol{f}(\boldsymbol{x}) \\ \boldsymbol{k}_2 &= h\boldsymbol{f}(\boldsymbol{x} + \tfrac{1}{2}\boldsymbol{k}_1) \\ \boldsymbol{k}_3 &= h\boldsymbol{f}(\boldsymbol{x} + \tfrac{1}{2}\boldsymbol{k}_2) \\ \boldsymbol{k}_4 &= h\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{k}_3) \end{aligned}\right\} \quad \hat{\phi}^h \boldsymbol{x} = \boldsymbol{x} + \frac{\boldsymbol{k}_1}{6} + \frac{\boldsymbol{k}_2}{3} + \frac{\boldsymbol{k}_3}{3} + \frac{\boldsymbol{k}_4}{6} + O(h^5) \tag{3.17}$$

For higher order formulas more $\boldsymbol{f}$-evaluations are needed. In many cases higher order formulas are advantageous because the additional computational expense can be more than compensated by a correspondingly bigger step size. Note, however, that this is not necessarily the case. Especially, if the right hand side is not very smooth, the step size is limited and lower order methods are favourable.

For non-autonomous ODE's it is often convenient to treat the time variable separately. Instead of (3.15) we then have

$$\boldsymbol{k}_i = h\boldsymbol{f}(t + c_i h, \boldsymbol{x} + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j), \quad \text{with} \quad c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, ..., s. \tag{3.18}$$

It became customary to summarize the coefficients of a particular Runge-Kutta method in a so-called Butcher tableau,

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \vdots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline \hat{\phi} & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array} \tag{3.19}$$

Using this notation the fourth-order Runge-Kutta formula (3.17) can be summarized in the following way:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline \hat{\phi} & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{3.20}$$

3.1.3 Error Estimation and Step Size Control

Adaptive step size control is an important issue for serious numerical integration. In the following suppose that we are able to estimate the local integration error by some number Δ. In order to maximize performance we would like to choose the step size h as large as possible. On the other hand, to guarantee the quality of the approximation the error estimate should not exceed a user-defined limit Δ^*.

For a p-th order integration method we know that the leading error term scales as h^{p+1}. Therefore, if a step size h results in some error Δ, we can compute an optimized step size h^* corresponding to the user-defined error limit Δ^* by

$$h^* = h \sqrt[p+1]{\frac{\rho \cdot \Delta^*}{\Delta}}. \tag{3.21}$$

Here we have introduced a safety factor $\rho < 1$. Also, since Δ may become arbitrarily small, h^* should be limited by an upper bound. Eq. (3.21) can be exploited for step size control as follows: If in the current step Δ was less than Δ^*, then h^* computed from Eq. (3.21) is used during the next iteration. Otherwise the current step is repeated with $h = h^*$. It should be noted that the real discretization error not only consists of a local component due to the evolution $\hat{\phi}^h \boldsymbol{x}_n$, but also of the accumulated error from all previous steps. However, controlling the accumulated error may require recomputation of the entire field line. Therefore, usually only the local error is considered.

A straight-forward way to obtain some local error estimate Δ is to compare two discrete evolutions of different order, an accurate one $\hat{\phi}^h \boldsymbol{x}$ and a less acurate one $\bar{\phi}^h \boldsymbol{x}$. The difference between the two solutions, $\hat{\phi}^h \boldsymbol{x} - \bar{\phi}^h \boldsymbol{x} = \hat{\epsilon} - \bar{\epsilon}$, is an approximation of the error of the less accurate evolution. However, it can be shown [21] that this approximation can safely be used to control the step size of the more accurate evolution via equation (3.21). Consequently, it has become common practice to proceed with $\hat{\phi}^h \boldsymbol{x}$ in the next iteration instead of $\bar{\phi}^h \boldsymbol{x}$, an approach known as *local extrapolation*.

Note, that the quantity $\hat{\epsilon} - \bar{\epsilon}$ represents an error vector, while in Eq. (3.21) a single scalar value Δ is required. This value is obtained by computing the magnitude of the error vector – preferably using a smooth norm like the Euclidean distance, i.e., $\Delta = |\hat{\epsilon} - \bar{\epsilon}|$ [70]. In many applications the individual error components have to be scaled properly before computing the norm. However, in visualization usually all components refer to space dimensions and thus are of same order of magnitude. The spatial interpretation of the solution curves also implies that it is adequate to consider absolute errors instead of relative ones.

Instead of computing two completely different discrete solutions some of the intermediate steps $\boldsymbol{k}_i$ in a Runge-Kutta method may be reused. Corresponding methods are known as embedded Runge-Kutta formulas. Another trick due to Fehlberg is to use the derivative $\boldsymbol{f}(\boldsymbol{x}_{i+1})$ in both the current and the following step. In this way from the classical fourth-order Runge-Kutta formula (3.17) a third-order approximation can be constructed,

$$\bar{\phi}^h \boldsymbol{x} = \boldsymbol{x} + \frac{\boldsymbol{k}_1}{6} + \frac{\boldsymbol{k}_2}{3} + \frac{\boldsymbol{k}_3}{3} + \frac{\boldsymbol{f}(\hat{\phi}^h \boldsymbol{x})}{6} + O(h^4). \tag{3.22}$$

The error estimate is given by

$$\hat{\phi}^h \boldsymbol{x} - \bar{\phi}^h \boldsymbol{x} = \frac{1}{6}(\boldsymbol{k}_4 - \boldsymbol{f}(\hat{\phi}^h \boldsymbol{x})). \tag{3.23}$$

The resulting adaptive integrator, denoted RK4(3), turns out to be very robust and well suited for many general-purpose applications. We took this integrator as the default method for most field line integration tasks in this work.

Beside RK4(3) we have also implemented a third-order method due to Bogacki and Shampine [5] and a fifth-order method due to Dormand and Prince [24]. We will call these methods RK3(2) and RK5(4), respectively. Both methods make use of local extrapolation, i.e., they compute an error estimate by means of a second-, respectively forth-order approximation, but proceed with the higher order solution. The third-order method is advantageous for rough vector fields as they may occur for example in case of coarse computational grids (c.f. Sec. 3.2). The fifth-order method is advantageous for very smooth data, e.g., for analytically defined vector fields or dynamical systems. Although of fifth-order, this method requires six intermediate $\boldsymbol{f}$-evaluation, including the one at the final position. It can be shown that this is the minimum number of stages for a fifth-order method. In general, higher order methods require a rapidly increasing number of stages. Since this overhead usually cannot be compensated by a larger step size, many high-order methods are of limited practical importance.

The Butcher tableaus of RK3(2), RK4(3), and RK5(4) are listed in Table 3.1. Of course, these methods are not the only ones of that order. However, it is argued that particular sets of coefficients like the ones in Table 3.1 are optimal with respect to certain design criteria. Usually, these design criteria include things like magnitude of the coefficient of the leading error term or ratio of this coefficient and the coefficient of the second-largest error term. However, the significance of these criteria with respect to pratical applications is not always clear.

3.1.4 Interpolation and Dense Output

Adaptive Runge-Kutta methods like the ones discussed above allow us to quickly compute an ODE on a discrete grid $\{t_i\}$ at guaranteed high accuracy. However, in many situations

RK3(2)

$$
\begin{array}{c|ccccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
\hat{\phi} & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
\hline
\bar{\phi} & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$

RK4(3)

$$
\begin{array}{c|ccccc}
0 & & & & & \\
\frac{1}{2} & \frac{1}{2} & & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & & \\
1 & 0 & 0 & 1 & & \\
1 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & \\
\hline
\hat{\phi} & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 0 \\
\hline
\bar{\phi} & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{6}
\end{array}
$$

RK5(4)

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
\hat{\phi} & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
\hline
\bar{\phi} & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

Table 3.1: This table summarizes the Butcher tableaus of some particularly "good" embedded Runge-Kutta methods we used for field line integration. The methods are of order three, four, and five, respectively. They all make use of the Fehlberg trick, i.e., they evaluate the right hand side at the final point of a step.

solution values have to be evaluated at prescribed locations τ, which in general are not part of the set $\{t_i\}$. In particualar, this is true for many visualization methods such as the ones discussed below. Restricting the step size of the integrator to the desired value would result in a tremendous overhead of work. Therefore, special interpolation strategies have been developed allowing one to approximate the solution anywhere between two break points $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$.

In this section we will concentrate on local methods, i.e., methods where an interpolation polynomial $\boldsymbol{z}(t)$ is constructed from the information of a single step only. The polynomial $\boldsymbol{z}(t)$ is said to be of order m (or degree $m-1$) if it has the form

$$\boldsymbol{z}(t) = \boldsymbol{c}_0 + \boldsymbol{c}_1\, t + \boldsymbol{c}_2\, t^2 + \ldots + \boldsymbol{c}_{m-1}\, t^{m-1}. \tag{3.24}$$

The order m of an interpolation polynomial should be related to the order p of the Runge-Kutta method being used. Any continuous interpolant must suffice at least $\boldsymbol{z}(t_i) = \boldsymbol{x}_i$ and $\boldsymbol{z}(t_{i+1}) = \boldsymbol{x}_{i+1}$.

Let $\tilde{\boldsymbol{z}}(t)$ be a polynomial of the same order as $\boldsymbol{z}(t)$ interpolating the exact solution $\boldsymbol{x}(t)$ instead of the discrete one. We may then split the local error $\boldsymbol{x}(t) - \boldsymbol{z}(t)$ into the following two terms:

$$\boldsymbol{x}(t) - \boldsymbol{z}(t) = (\boldsymbol{x}(t) - \tilde{\boldsymbol{z}}(t)) + (\tilde{\boldsymbol{z}}(t) - \boldsymbol{z}(t)), \quad t_i \le t \le t_{i+1} \tag{3.25}$$

The first term can be regarded as an *interpolation error*. It doesn't depend on the integration formula being used. It is a standard result of interpolation theory that the interpolation error is of order $O(h^m)$, where m denotes the order of the interpolant and $h = t_{i+1} - t_i$

[18]. The second term in Eq. (3.25) can be regarded as a *data error*. Shampine [77] shows that this term is of order $O(h^{p+1})$ provided a pth-order integration formula has been used.

From these results it follows that the order of the integration formula is preserved if we choose $m > p$. While for $m > p+1$ the data error dominates, for $m = p+1$ the interpolation error becomes equally important. Unfortunately, in many situations interpolants with $m > p$ cannot be obtained for free, i.e., without requiring additional $\boldsymbol{f}$-evaluations. Instead, often the order of the interpolation polynomial is chosen to be *equal* to the order of the integration formula. In this case we have

$$\boldsymbol{x}(t) - \boldsymbol{z}(t) \sim h^p + O(h^{p+1}). \tag{3.26}$$

The reason why this approach is still acceptable is that the interpolant is used to produce single output values only, i.e., interpolation errors don't accumulate like local errors of the integrator. Therefore, it is reasonable to allow the interpolation error to be of the same order of magnitude as the *global error* of the integrator, namely $O(h^p)$. Moreover, if local extrapolation is used the numerical error estimate Δ will also be of order $O(h^p)$. Thus, by controlling Δ also the interpolation error (3.26) will be controlled.

An elegant way of establishing an interpolation polynomial $\boldsymbol{z}(t)$ is provided by so-called *continuous* Runge-Kutta formulas. In these formulas the coefficients b_i are replaced by functions $b_i(u)$, so that the resulting approximation

$$\boldsymbol{z}(t_i + uh) = \boldsymbol{x}_i + h \sum_{i=1}^{s} b_i(u)\, \boldsymbol{k}_i, \quad 0 \le u \le 1, \tag{3.27}$$

smoothly interpolates between $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$. Although it is not obvious how to find suitable functions $b_i(u)$ in general, for some pth-order formulas a pth-order interpolant can be constructed. In particular, the RK5(4) method of Dormand and Prince listed in Table 3.1 can be extended in the following way [37]:

$$\begin{aligned}
b_1(u) &= u - \tfrac{1337}{480}\, u^2 + \tfrac{1039}{360}\, u^3 - \tfrac{1163}{1152}\, u^4 \\
b_2(u) &= 0 \\
b_3(u) &= \tfrac{100}{3}\, u^2 \left(\tfrac{1054}{9275} - \tfrac{4682}{27825}\, u + \tfrac{379}{5565}\, u^2\right) \\
b_4(u) &= -\tfrac{5}{2}\, u^2 \left(\tfrac{27}{40} - \tfrac{9}{5}\, u - \tfrac{83}{96}\, u^2\right) \\
b_5(u) &= \tfrac{18225}{848}\, u^2 \left(-\tfrac{3}{250} + \tfrac{22}{375}\, u - \tfrac{37}{600}\, u^2\right) \\
b_6(u) &= -\tfrac{22}{7}\, u^2 \left(-\tfrac{3}{10} + \tfrac{29}{30}\, u - \tfrac{17}{24}\, u^2\right)
\end{aligned} \tag{3.28}$$

A simple and elegant way to obtain a fourth-order C^1-continuous interpolant is provided by cubic Hermite interpolation. In addition to the locations $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$ also the derivatives $\boldsymbol{f}_i \equiv \boldsymbol{f}(\boldsymbol{x}_i)$ and $\boldsymbol{f}_{i+1} \equiv \boldsymbol{f}(\boldsymbol{x}_{i+1})$ are interpolated. As can be verified by Newton's interpolation formula, the corresponding interpolation polynomial is given by

$$\begin{aligned}
\boldsymbol{z}(t_i + uh) = \boldsymbol{x}_i &+ u\, h\boldsymbol{f}_i + u^2\, (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i - h\boldsymbol{f}_i) \\
&+ u^2(u-1)\, (h\boldsymbol{f}_{i+1} - 2(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) + h\boldsymbol{f}_i), \quad 0 \le u < 1.
\end{aligned} \tag{3.29}$$

We use this interpolant for both the fourth-order method RK4(3) and the third-order method RK3(2) listed in Table 3.1.

3.1.5 Forward Differences on Equidistant Grids

The fast LIC algorithm discussed in Chapter 4 requires field lines to be sampled at a large number of intermediate points, i.e., interpolation polynomials need to be evaluated quite often. In this case efficiency becomes an important issue.

Consider an mth-order interpolation polynomial $\boldsymbol{z}(u)$. Of course, we need not to compute every power in u separately, but may apply Horner's rule, i.e.,

$$\begin{aligned} \boldsymbol{z}(u) &= \boldsymbol{c}_0 + \boldsymbol{c}_1\, u + \boldsymbol{c}_2\, u^2 + \ldots + \boldsymbol{c}_{m-1}\, u^{m-1} \\ &= \boldsymbol{c}_0 + u\,(\boldsymbol{c}_1 + u\,(\boldsymbol{c}_2 + u\,(\ldots + u\,\boldsymbol{c}_{m-1})\ldots)). \end{aligned} \tag{3.30}$$

However, this approach still requires $m-1$ additions and $m-1$ multiplications per vector component. Substantial speed-ups can be achieved if the polynomial is to be evaluated at evenly spaced locations only. This will exactly be the case in our fast LIC algorithm. Evenly spaced samples can be efficiently computed using a forward differencing scheme, reducing the cost to just $m-1$ additions and no multiplication at all after initialization.

Assume we want to evaluate the interpolation polynomial $u_k = u_0 + k\delta$, $k = 0, 1, 2, ..,$ where δ denotes the sampling distance. We then may define forward differences in the following way

$$\begin{aligned} \Delta^1 \boldsymbol{z}(u_k) &= \boldsymbol{z}(u_{k+1}) - \boldsymbol{z}(u_k) \\ \Delta^{i+1} \boldsymbol{z}(u_k) &= \Delta^i \boldsymbol{z}(u_{k+1}) - \Delta^i \boldsymbol{z}(u_k), \quad i > 1. \end{aligned} \tag{3.31}$$

Like for ordinary derivatives, it turns out that for any polynomial of order m the difference $\Delta^{m-1}\boldsymbol{z}(u_k)$ will be constant for all k. Higher order differences vanish completely. Thus, after initially computing $\Delta^i \boldsymbol{z}(u_0)$, $0 < i < m$, at location u_0 we may obtain subsequent values by means of the following update scheme:

$$\begin{aligned} \boldsymbol{z}(u_{k+1}) &= \boldsymbol{z}(u_k) + \Delta^1(u_k) \\ \Delta^i \boldsymbol{z}(u_{k+1}) &= \Delta^i \boldsymbol{z}(u_k) + \Delta^{i+1} \boldsymbol{z}(u_k), \quad i > 1. \end{aligned} \tag{3.32}$$

In the particular case of a fourth-order polynomial like the cubic Hermite interpolant defined in Eq. (3.29) the initial differences are given by

$$\begin{aligned} \Delta^1 \boldsymbol{z}(u_0) &= (3\delta u_0^2 + 3\delta^2 u_0 + \delta^3)\, \boldsymbol{c}_3 + (2\delta u_0 + \delta^2)\, \boldsymbol{c}_2 + \delta^3 \boldsymbol{c}_1 \\ \Delta^2 \boldsymbol{z}(u_0) &= (6\delta^2 u_0 + 6\delta^3)\, \boldsymbol{c}_3 + 2\delta^2 \boldsymbol{c}_2 \\ \Delta^3 \boldsymbol{z}(u_0) &= 6\delta^3 \boldsymbol{c}_3 \end{aligned} \tag{3.33}$$

For a fifth-order polynomial like the interpolant of the RK5(4) method defined by Eq. (3.27) and (3.28) we have to add the following terms:

$$\begin{aligned} \Delta^1 \boldsymbol{c}_4 u_0^4 &= (4\delta u_0^3 + 6\delta^2 u_0^2 + 4\delta^3 u_0 + \delta^4)\, \boldsymbol{c}_4 \\ \Delta^2 \boldsymbol{c}_4 u_0^4 &= (12\delta^2 u_0^2 + 24\delta^3 u_0 + 14\delta^4)\, \boldsymbol{c}_4 \\ \Delta^3 \boldsymbol{c}_4 u_0^4 &= (24\delta^3 u_0 + 36\delta^4)\, \boldsymbol{c}_4 \\ \Delta^4 \boldsymbol{c}_4 u_0^4 &= 24\delta^4\, \boldsymbol{c}_4 \end{aligned} \tag{3.34}$$

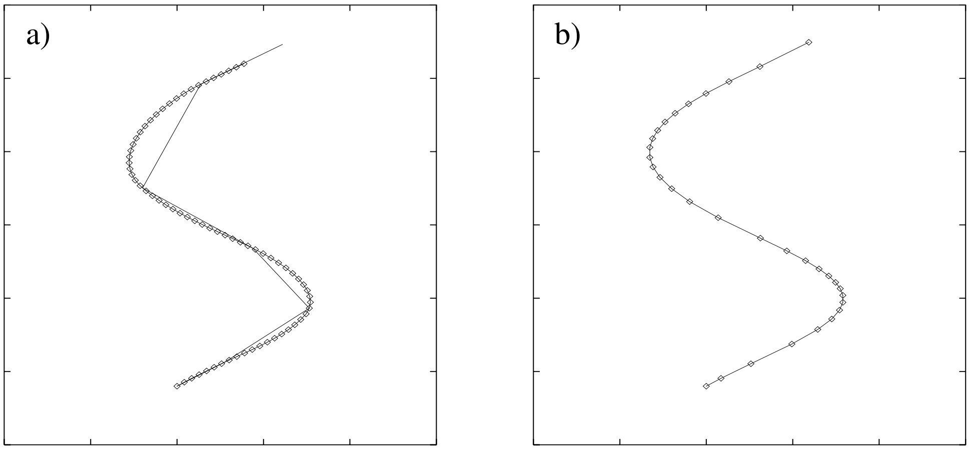


Figure 3.1: Intermediate points on a field line can be efficiently computed using an appropriate interpolation polynomial. In Figure (a) a cubic Hermite polynomial has been used to interpolate equi-distant samples from the results of the RK4(3) integrator. Figure (b) illustrates a curvature-adapted straight line approximation of a field line.

Note, that these expressions become much simpler if we take $u_0 = 0$. However, our goal is to sample multiple integration intervals with equal increments. Since the length of a single interval in general will not be an integer multiple of the sampling distance, the remaining fractional distance (which just doesn't fit into the current interval anymore) has to be taken as an offset for the next interval. Therefore, non-vanishing inital values u_0 arise.

An example of an evenly sampled field line is shown in Figure 3.1 a). The points $\boldsymbol{x}_i$ computed by the integrator itself are connected by straight line segments. The figure clearly indicates that without a proper interpolant higher-order integrators are unsuitable for graphical output.

3.1.6 Straight Line Approximations

Beside the sampling of a field line at many equi-distant locations, another common requirement in visualization is the approximation of a curve by a minimal number of straight line segments. These line segments need not to be of equal length. Instead they should approximate the shape of the curve as close as possible. Obviously, this requires that the sampling distance is somehow related to the curve's curvature.

Sometimes curvature-adapted straight line approximations are computed in a post-processing step by simplifying a densely sampled intermediate polyline. However, this is clearly not an optimal approach with respect to both performance and quality. Other authors tried to modify the step size control mechanism of the integrator, so that step size is adapted to curvature [17, 49]. This may lead to a highly increased number of

$\boldsymbol{f}$-evaluations. In addition, the accuracy of the approximation is not optimal if simple error criteria are used such as the angle between two sucessive line segments. A better approach would be to use the full information provided by the polynomials $\boldsymbol{z}(t)$ in order to compute a straight line approximation.

The difference between an arbitrary function $g(t)$ and an mth-order interpolant $p_n(t)$ constructed from m points τ_k such that $g(\tau_k) = p_n(\tau_k)$ can be expressed by

$$g(t) = p_n(t) + (t - \tau_1)...(t - \tau_m)\,[\tau_1, ..., \tau_m, t]\,g, \tag{3.35}$$

where $[\tau_1, ..., \tau_m, t]g$ denotes the mth divided difference of g [18]. Furthermore, it can be shown that the divided difference is related to the mth derivative of g evaluated at some intermediate location $\tau \in [\tau_1, \tau_m]$:

$$[\tau_1, ..., \tau_m, t]\,g = g^{(m)}(\tau)/m! \tag{3.36}$$

In the following we want to create a piecewise linear approximation $\bar{\boldsymbol{z}}(t)$ to the higher-order interpolant $\boldsymbol{z}(t)$ of the integration formula. Introducing a set of breakpoints $\{\tau_k\}$ (different from the discrete mesh $\{t_i\}$ used by the integrator itself), this approximation is given by

$$\bar{\boldsymbol{z}}(t) = \boldsymbol{z}(\tau_k) + \frac{t - \tau_k}{\tau_{k+1} - \tau_k}\,(\boldsymbol{z}(\tau_{k+1}) - \boldsymbol{z}(\tau_k))\,, \quad \tau_k \le t < \tau_{k+1}. \tag{3.37}$$

According to Eq. (3.35) the difference between $\boldsymbol{z}$ and $\bar{\boldsymbol{z}}$ involves a second divided difference. Applying Eq. (3.36) in every interval $[\tau_k, \tau_{k+1}]$ we obtain the following error estimate:

$$|\boldsymbol{z}(t) - \bar{\boldsymbol{z}}(t)\,| \le \left(\frac{\tau_{k+1} - \tau_k}{2}\right)^2 \max_{\tau_k \le t < \tau_{k+1}} \left|\frac{\boldsymbol{z}''(t)}{2}\right|, \quad \tau_k \le t < \tau_{k+1} \tag{3.38}$$

Given an arbitrary point τ_k our goal is to find the next point τ_{k+1} such that the distance $\tau_{k+1} - \tau_k$ is as large as possible but, at the same time, the total error of the approximation given by (3.38) does not exceed a user-defined limit. Given an analytic expression for the second derivative $\boldsymbol{z}''(t)$ then, in principal, we could find such a point directly. However, since $\boldsymbol{z}''(t)$ itself is a piecewise polynomial function and because of the maximum operation in (3.38) this is not a trivial task.

As an alternative we may rely on stepping along the field line with small equi-distant increments δ. This can be efficiently achieved using the forward difference scheme described in the previous section. At each step we compute an estimate of $\boldsymbol{z}''$ and keep track of the maximum value $|\boldsymbol{z}''|_{\max}$. Eventually, after n steps the expression $(n\delta)^2|\boldsymbol{z}''|_{\max}/8$ will exceed a predefined error limit. We then accept the current sample as a breakpoint for the straight line approximation, reset n as well as $|\boldsymbol{z}''|_{\max}$, and continue stepping along the field line.

The only remaining question is how to compute the value of $\boldsymbol{z}''$ at a sample point. However, since $\boldsymbol{z}$ is a polynomial function of order m, $\boldsymbol{z}''$ will be a polynomial of order

$m-2$. In particluar, in case of the cubic Hermite interpolant $\boldsymbol{z}''$ is a linear function, while for the RK5(4) interpolant it is a qudratic function. In any case, like for $\boldsymbol{z}$ itself we can apply the forward differences technique to evaluate $\boldsymbol{z}''$ efficiently for equi-distant samples.

Figure 3.1 b) shows an example of a field line approximated by straight lines. Note, how the distance between subsequent breakpoints nicely adapts to the curvature of the curve. Of course, one can ask why the integrator itself couldn't be modified so that it directly computes the correct breakpoints. The difference between the true integral curve and a straight line approximation is of order $O(h^2)$. An appropriate error estimate would be obtained by comparing the result of Eq. (3.12) (midpoint rule) with the result of a simple Euler step. Indeed, the RK2(1) formula established in this way can be used to produce a similar result as in Figure 3.1 b). However, this method of course affects the *global error* of the solution because fairly large local errors are accumulated.

3.1.7 Discontinuous Problems

From the discussion in the previous sections we learned that smooth local interpolants of a numerical integration formula are a valuable and versatile tool. They can be used to produce a large number of samples on a field line as well as to ease the construction of curvature-adapted straight line approximations. Yet another important application is the integration of discontinuous vector fields. For example, discontinuities frequently occur at domain boundaries, e.g., at the boundries of an underlying computational grid. Another example are electric fields as shown in Figure 1.2 b). These fields exhibit discontinuities at the boundary of materials with different dielectric properties.

When encountering a discontinuity on the right hand side any robust numerical integrator will automatically reduce its step size in order to ensure the user-defined error limit. While in this way the quality of the approximation is guaranteed, performance heavily breaks down due to a large number of small steps. In addition, discontinuities usually lead to a huge number of *rejected* steps. Typically, after every sucessful small step just before a discontinuity a much bigger step will be suggested for the next iteration. The bigger step is likely to cross the discontinuity, and thus will be rejected again.

Interpolants provide an elegant way to circumvent these problems. The general idea is based on the observation that an interpolant not only approximates the true solution in the interval $[t_i, t_i + h]$ but also in the interval $[t_i + h, t_i + 2h]$. In fact, it can be shown that the order of the approximation remains the same, although the associated error constants are larger in general [27]. This means that we can use the interpolant of the last sucessful step in order to locate a discontinuity ahead and to restart integration afterwards.

Enright et. al. [28] proposed an algorithm where the occurence of a discontinuity is detected fully automatically. Evaluation of the right hand side of the ODE can take place in a standard way, i.e., the occurence of a discontinuity need not to be indicated explicitely. Instead, it is suposed that there exists a discontinuity somewhere in the range $[t_i{+}h, t_i{+}2h]$ if the step from t_i to $t_i{+}h$ was immediately preceded by a rejected attempted step and the proposed step size for the next step is $h^* \geq 2h$. The authors reported this automatic criterium to be quite reliable. After the occurence of a discontinuity has been signaled its exact location is determined by sampling the interpolant $\boldsymbol{z}(t)$ in $[t_i+h, t_i+2h]$. In particular, a bracketing strategy is applied, assuming that a location t is right from the discontinuity if the ratio of the so-called defect $\delta(t)$ of the interpolant at t and the defect $\delta(t^*)$ at a specially chosen location $t^* \in [t_i, t_i + h]$ exceeds a certain value. The defect of the interpolant is defined by

$$\delta(t) = \boldsymbol{z}'(t) - \boldsymbol{f}(\boldsymbol{z}(t)). \tag{3.39}$$

After the location of the discontinuity has been determined up to the required tolerance, integration is restarted using the same step size as in the last successful step before the discontinuity event.

The main advantage of the described automatic algorithm is its simplicity from the user's point of view. It is argued that it might be difficult to provide user code which is able to signal discontinuities under all circumstances. However, in our case discontinuities are usually associated with certain domain boundaries in a computational grid. Each domain can be identified by a unique label. Therefore, we need not to apply an automatic criterium to identify discontinuities, but may merely compare domain IDs. Whenever a new domain is encountered, we reject a step and sample the interpolant like in Enright's algorithm in order to determine the exact location of the boundary. Of course, instead of analysing the defect of the interpolant we again look at the domain ID to decide whether we are left or right from the boundary. In some rare situations it might occur that the interpolant does not cross a domain boundary, although such a boundary was seen before. Similar to [28], in this case we simply continue with a reduced step size as suggested by the step size control mechanism.

The resulting algorithm turns out to be quite efficient for integrating discontinuous vector fields. If a vector field is defined on a discrete grid, we commonly apply this technique to trace field lines exactly up to the boundary of the grid.

3.2 Data Representation and Evaluation

In the following we want to discuss how to represent and evaluate the right hand side of an ODE. Usually in scientific visualization a vector field $\boldsymbol{f}(\boldsymbol{x})$ will not be given by a mathematical expression which can be evaluated for an arbitrary location $\boldsymbol{x}$. Instead, the data is commonly defined on a discrete grid. In the following we shortly summarize different grid types and sketch associated interpolation methods. In addition, point location techniques will be described. These techniques are necessary for evaluating fields on a discrete grid at arbitrary locations.

3.2.1 Computational Grids

A discrete computational grid or mesh subdivides the domain Ω on which a field $\boldsymbol{f}(\boldsymbol{x})$ is defined into elementary grid cells. Computational grids can be classified according

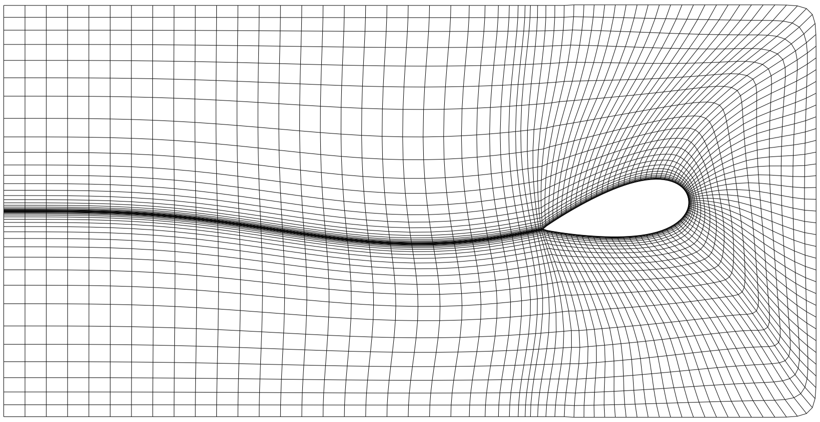


Figure 3.2: Regular two-dimensional grid with curvilinear coordinates. The grid smoothly bends around a wing profile with two boundary sides being glued together behind the wing.

to the shape of the elementary cells. The most common cell types are triangular and quadrilateral cells in two dimensions as well as tetrahedral and hexahedral cells in three dimensions. Other cell types like prisms can be used as well. It is also possible to combine cells of different types in a single grid. For an overview we refer to standard text books on finite element analysis, e.g. [95]. Conceptually, at least *regular*, *block-structured*, *irregular* and *hybrid* grids can be distinguished. In our discussion we concentrate on simple regular and irregular examples.

Regular Grids. A regular grid consists solely of quadrilateral cells in 2D or hexahedral cells in 3D. These cells are logically arranged in an array, so that they can be addressed via an index pair (i, j) in 2D or via an index triple (i, j, k) in 3D, respectively. Regular grids are further distinguished according to the kind of coordinates being used. The most simple case comprises so-called *uniform* coordinates, where all cells are assumed to be rectangular and axis-aligned. Moreover, the grid spacing is constant along each axis. In case of *rectilinear* coordinates the cells are still aligned to the axes, but the grid spacing may vary from cell to cell. Finally, in case of *curvilinear* coordinates each node of the grid may have arbitrary coordinates. Grids with curvilinear coordinates are often used in fluid dynamics because they have a simple structure but still allow for accurate modeling of complex shapes like rotor blades or airfoils. A 2D example of a regular grid with curvilinear coordinates is shown in Figure 3.2.

Irregular Grids. In contrast to regular grids irregular grids are usually built from triangles or tetrahedrons. This provides a much greater flexibility, allowing one to model almost any type of geometric object. The cells are not arranged in an array anymore, but are more or less irregularly distributed. A big advantage of triangular and tetrahedral cells is that they can be refined easily in a consistent way without changing the overall structure of the grid. This makes it possible to adapt the resolution of the grid locally. In addition, simple linear interpolants can be defined inside triangular and tetrahedral cells. An example of a triangular grid with locally varying resolution is shown in Figure 3.3.

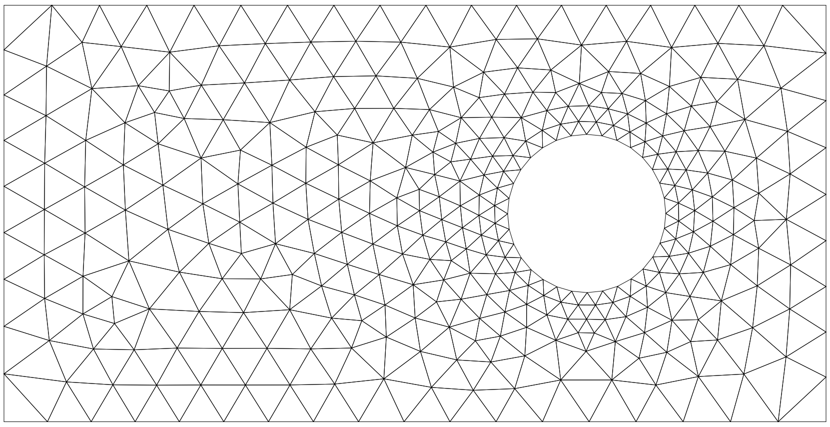


Figure 3.3: Triangular grid around a circle. Irregular grids are well suited to represent computational domains of arbitrary topology. In addition, the grid resolution can be adapted locally.

3.2.2 Interpolation and Local Coordinates

In order to define a function $\boldsymbol{f}(\boldsymbol{x})$ on a discrete grid, data values have to be specified at certain points of the grid. In addition, an interpolation rule has to be provided allowing one to evaluate the function at any point inside a given grid cell. Often the interpolated function should at least be continuous across the boundaries of adjacent cells. Therefore, data values are commonly specified at the corners of a cell. In case of triangular or tetrahedral cells then a linear interpolant can be defined, while for quadrilateral or hexahedral cells usually bilinear or trilinear interpolants are applied.

In the following, let us first consider the special case of quadrilateral cells to point out some general concepts. The four data values at the corners of a quadrilateral cell provide just enough information to determine a bilinear interpolant, i.e., an interpolant of the form

$$\boldsymbol{f}(\boldsymbol{p}) = \boldsymbol{\alpha}_0 + \boldsymbol{\alpha}_1 x + \boldsymbol{\alpha}_2 y + \boldsymbol{\alpha}_3 xy, \quad \boldsymbol{p} = (x, y)^T. \tag{3.40}$$

The coefficients $\boldsymbol{\alpha}_i$ have to be chosen such that the data values $\boldsymbol{f}_i$ at the cell's corners are reproduced. This can be achieved by solving a 4×4 linear system for each vector component. Higher-order interpolation methods can be established by providing derivative information or by defining data values at additional points inside a cell or at the cell's boundaries.

It turns out, that direct evaluation of the coefficients $\boldsymbol{\alpha}_i$ of an interpolation polynomial is numerically quite unfavourable. Especially, for non-linear interpolants the solution of the associated linear system may not be trivial. A more robust and stable alternative is based on the introduction of local coordinates inside a cell. The interpolant can then be expressed directly as a linear combination of the orginal data values $\boldsymbol{f}_i$ with the corresponding weights being computed as a function of the local coordinates. For example, in case of a quadrilateral cell local coordinates u and v can be defined as shown in Figure 3.4. By means of these variables the bilinear interpolant can be written as

$$\boldsymbol{f} = (1-u)(1-v)\,\boldsymbol{f}_0 + u(1-v)\,\boldsymbol{f}_1 + v(1-u)\,\boldsymbol{f}_2 + uv\,\boldsymbol{f}_3. \tag{3.41}$$

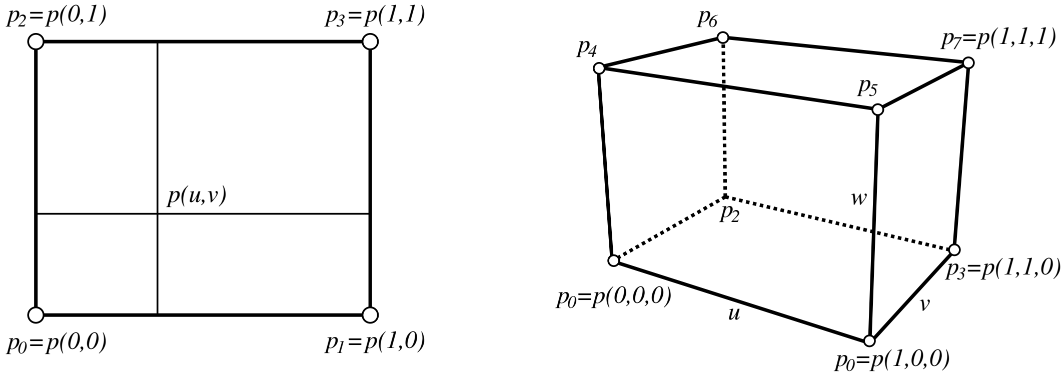


Figure 3.4: Local coordinates u, v in a quadrilateral cell (left) or u, v, w in a hexahedral cell (right) vary from 0 to 1 along the cell's edges. These coordinates are also defined for distorted cells, e.g., in case of curvilinear coordinates.

It should be mentioned that suitable local coordinates can be defined for virtually any type of grid cell. While hexahedral cells are parametrized similarly to quadrilateral cells in 2D, in case of triangular and tetrahedral cells so-called barycentric coordinates are used. These will be discussed in Section 3.2.5. Of course, higher-order interpolants can be expressed in terms of local coordinates as well. In finite element analysis the particular combinations of local coordinates like $(1-u)(1-v)$, $u(1-v)$, $v(1-u)$, and uv in Eq. (3.41) are called *shape functions*. A grid cell together with a given set of shape functions is called an *element*.

3.2.3 Tensor Product Interpolants

Bilinear interpolation as defined in the previous section can be regarded as the tensor product of two linear interpolations. From a computational point of view it is advantagous to exploit this structure. Thus, instead of Eq. 3.41 we have:

$$\left.\begin{aligned} \boldsymbol{f}_{01} &= \boldsymbol{f}_0 + u\,(\boldsymbol{f}_1 - \boldsymbol{f}_0) \\ \boldsymbol{f}_{23} &= \boldsymbol{f}_2 + u\,(\boldsymbol{f}_3 - \boldsymbol{f}_2) \end{aligned}\right\} \quad \boldsymbol{f} = \boldsymbol{f}_{01} + v\,(\boldsymbol{f}_{23} - \boldsymbol{f}_{01}) \tag{3.42}$$

The same result $\boldsymbol{f}$ would be obtained if the order of the linear interpolations in u and v is reversed, i.e., intermediate values $\boldsymbol{f}_{02}$ and $\boldsymbol{f}_{13}$ instead $\boldsymbol{f}_{01}$ and $\boldsymbol{f}_{23}$ are computed.

The three-dimensional analogon to bilinear interpolation is trilinear interpolation. Using the local coordinates u, v, and w of a hexahedral cell a trilinear interpolant can be computed in the following way:

$$\left.\begin{aligned} \boldsymbol{f}_{01} &= \boldsymbol{f}_0 + u\,(\boldsymbol{f}_1 - \boldsymbol{f}_0) \\ \boldsymbol{f}_{23} &= \boldsymbol{f}_2 + u\,(\boldsymbol{f}_3 - \boldsymbol{f}_2) \\ \boldsymbol{f}_{45} &= \boldsymbol{f}_4 + u\,(\boldsymbol{f}_5 - \boldsymbol{f}_4) \\ \boldsymbol{f}_{67} &= \boldsymbol{f}_6 + u\,(\boldsymbol{f}_7 - \boldsymbol{f}_6) \end{aligned}\right\} \quad \left.\begin{aligned} \boldsymbol{f}_{23}^{01} &= \boldsymbol{f}_{01} + v\,(\boldsymbol{f}_{23} - \boldsymbol{f}_{01}) \\ \boldsymbol{f}_{67}^{45} &= \boldsymbol{f}_{45} + v\,(\boldsymbol{f}_{67} - \boldsymbol{f}_{45}) \end{aligned}\right\} \quad \boldsymbol{f} = \boldsymbol{f}_{23}^{01} + w\,(\boldsymbol{f}_{67}^{45} - \boldsymbol{f}_{23}^{01}) \tag{3.43}$$

Although linear tensor product interpolants are by far the most popular choice for regular grids, in some cases smoother results are required. In particular, this might be necessary for low resolution input grids. In order to construct a C^1-continuous interpolant an estimate of the partial derivatives of $\boldsymbol{f}$ at the cell boundaries is required. Then for example the tensor product of multiple one-dimensional cubic Hermite interpolants can be computed. However, often the partial derivatives will not be provided explicitely but have to be approximated by central differences. Suppose, we want to compute a cubic interpolant between $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$ on a one-dimensional grid. Then

$$\boldsymbol{f}'_i \approx (\boldsymbol{f}_{i+1} - \boldsymbol{f}_{i-1})/2, \quad \text{and} \quad \boldsymbol{f}'_{i+1} \approx (\boldsymbol{f}_{i+2} - \boldsymbol{f}_i)/2. \tag{3.44}$$

Computing a cubic Hermite interpolant using these two expressions yields a so-called Catmull-Rom spline. It can be written in matrix form as

$$\boldsymbol{f}(u) = [u^3\ u^2\ u\ 1] \begin{bmatrix} -1/2 & 3/2 & -3/2 & 1/2 \\ 1 & -5/2 & 2 & -1/2 \\ -1/2 & 0 & 1/2 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{f}_{i-1} \\ \boldsymbol{f}_i \\ \boldsymbol{f}_{i+1} \\ \boldsymbol{f}_{i+2} \end{bmatrix} \tag{3.45}$$

In order to construct a bicubic tensor product interpolant in a regular 2D grid we first apply the Catmull-Rom spline to four rows in the grid. We then interpolate the obtained intermediate values again in column direction to get the final result. Like in the linear case, the order of interpolation doesn't matter, i.e., rows and columns may be interchanged.

The same method can be applied to 3D hexahedral cells in a straight-forward way. In this case the one-dimensional interpolant (3.45) has to be evaluated $16+4+1$ times. Though the results of tricubic interpolation are of high quality, it is clear that the approach is computationally much more expensive compared to simple trilinear interpolation.

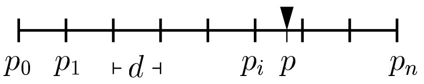
Finally, it should be mentioned that in finite element analysis another class of interpolants is commonly used for quadrilateral and hexahedral elements. Instead of m^d input values as for an mth-order tensor product in d dimensions these so-called Serendipity elements [95] are built from fewer data values, all of which are located at the boundaries of a cell. The interpolants do not contain so-called spurious powers of high order like $(uvw)^{m-1}$, but at most uvw^{m-1}, $uv^{m-1}w$, and $u^{m-1}vw$. However, unless all required data is directly available, e.g. from a finite element simulation, in our case the tensor product approach is usually preferable. It requires less memory (derivatives are computed on the fly) and can be implemented more easily.

A general drawback of all tensor product interpolants including the linear ones is that they are not rotationally invariant. For example, along the diagonal of a grid cell the order of the interpolation polynomial is higher then along its edges. Therefore, certain artefacts can be introduced depending on how the grid is oriented. The effect is less dominant for the mentioned Serendipity elements, although these are not fully isotropic, too.

3.2.4 Point Location in Regular Grids

In order to evaluate a field defined on a discrete grid at an arbitrary point $\boldsymbol{p}$ first the grid cell containing $\boldsymbol{p}$ has to be found and second the local coordinates of $\boldsymbol{p}$ in this cell have to be computed. In the following we illustrate how this can be done for regular grids with various types of coordinates.

Uniform coordinates. For a regular grid with uniform coordinates the task of locating an arbitrary point, i.e., determining the grid cell it is contained in, is very simple. In each dimension the corresponding index i of the grid cell as well as the associated local coordinate u can be computed as indicated in the following figure:



$$i = \left\lfloor \frac{p - p_0}{d} \right\rfloor, \quad u = \frac{p - p_i}{d}$$

Rectilinear coordinates. In case of rectilinear coordinates computations still can be performed independently in each dimension. However, since the grid spacing along the major axes need not to be constant, the index of the grid cell containing p cannot be determined by simple divison anymore. Instead, a more complex search strategy has to be applied. In our implementation we utilized a bisection approach characterized by complexity $O(\log n)$, where n denotes the number of grid cells in the relevant dimension. Let p_i be the coordinates of the grid's vertices and assume $p_0 \leq p \leq p_n$, i.e., the searched point lies inside the grid. Starting with $i = \lfloor n/2 \rfloor$, we replace i by $\lfloor i/2 \rfloor$ if $p > p_{i+1}$. If $p < p_i$ we replace i by $n - \lfloor (n-i)/2 \rfloor$. This procedure is repeated until $p_i \leq p \leq p_{i+1}$, i.e., until the correct cell index is found. The local coordinate u is then given by $(p - p_i)/(p_{i+1} - p_i)$.

The outlined bisection algorithm performs a *global search* in the grid. Thus, it can be used to locate an arbitrary point $\boldsymbol{p}$. However, often the desired point $\boldsymbol{p}$ will not be too far away from some other point $\boldsymbol{q}$ the cell index of which is already known. For example, during field line integration the locations at which the field has to be evaluated will be closely correlated. In this situation it may be favourable to apply a *local search* instead of a global one. For rectilinear coordinates this can be implemented by testing whether cells in the vicinity of a seed cell contain the desired point or not. In our software implementation we provided both global and local search strategies whenever appropriate.

Curvilinear coordinates. Point location in a regular grid with curvilinear coordinates is much more difficult than for axis-aligned uniform or rectilinear coordinates. At least in 3D hexahedral cells even the test whether a point is contained in a particular grid cell or not is not trivial. This is because the cell's faces need not to be planar. Instead, they are usually defined by bilinear patches. However, if we were able to compute the local coordinates of a point with respect to an arbitrary grid cell then of course we can also decide whether the point lies in the cell or not. For a point inside the cell all local coordinates must be in the range $0...1$. In the following, we first present a method for computing local coordinates. Afterwards we outline, how efficient global and local search strategies can be established.

Consider a point $\boldsymbol{p}^* \in \mathbb{R}^2$. We are looking for the local coordinates u and v of $\boldsymbol{p}^*$ in a quadrilateral grid cell formed by the four vertices $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ (c.f. Figure 3.4).

Suppose, the physical coordinates of the cell depend bilinearly on u and v. Then

$$\boldsymbol{p}(u, v) = (1-u)(1-v)\,\boldsymbol{p}_0 + u(1-v)\,\boldsymbol{p}_1 + (1-u)v\,\boldsymbol{p}_2 + uv\,\boldsymbol{p}_3. \tag{3.46}$$

We need to determine u and v such that $\boldsymbol{p}(u, v) = \boldsymbol{p}^*$. Instead of solving the resulting non-linear system explicitly, it is usually much easier and often more accurate to compute a solution by means of numerical techniques. In particular, Newton's method provides a suitable iterative algorithm with excellent convergence. Given some initial guess of u and v we look for increments δu and δv to improve the approximation. A Taylor series expansion of (3.46) yields

$$\boldsymbol{p}(u{+}\delta u, v{+}\delta v) = \boldsymbol{p}(u, v) + \frac{\partial \boldsymbol{p}}{\partial u}\,\delta u + \frac{\partial \boldsymbol{p}}{\partial v}\,\delta v + O(\delta u^2) + O(\delta v^2), \tag{3.47}$$

with

$$\frac{\partial \boldsymbol{p}}{\partial u} = (1-v)(\boldsymbol{p}_1 - \boldsymbol{p}_0) + v\,(\boldsymbol{p}_3 - \boldsymbol{p}_2) \tag{3.48}$$

and

$$\frac{\partial \boldsymbol{p}}{\partial v} = (1-u)(\boldsymbol{p}_2 - \boldsymbol{p}_0) + u\,(\boldsymbol{p}_3 - \boldsymbol{p}_1). \tag{3.49}$$

Consequently, the desired increments δu and δv can be computed by solving the following 2×2 linear system:

$$\frac{\partial \boldsymbol{p}}{\partial u}\,\delta u + \frac{\partial \boldsymbol{p}}{\partial v}\,\delta v = \boldsymbol{p}^* - \boldsymbol{p}. \tag{3.50}$$

Usually, only 2-4 iterations are required to obtain a sufficiently accurate solution. The resulting algorithm sometimes is called *stencil walk*. The choice of the initial values u and v is arbitrary. We may simply take $u = v = 1/2$. A more accurate initial guess could be obtained for example by computing the projection of $\boldsymbol{p}^* - \boldsymbol{p}_0$ on $\boldsymbol{p}_1 - \boldsymbol{p}_0$ and $\boldsymbol{p}_2 - \boldsymbol{p}_0$, respectively. The stencil walk method can be easily generalized to 3D hexahedral cells. Then in each iteration a 3×3 linear system has to be solved in order to find increments δu, δv, and δw. It is also possible to apply it in cases where the cell coordinates are higher-order functions of the local coordinates, e.g. tensor products of cubic splines.

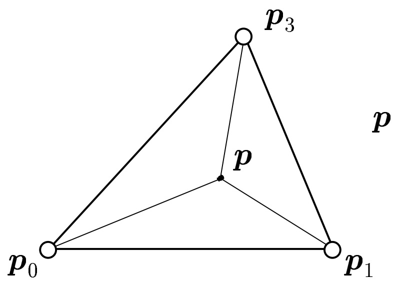
In order to implement a global point search in a regular grid with curvilinear coordinates in principal for all grid cells it has to be tested whether a cell contains $\boldsymbol{p}^*$ or not. This can only be done efficiently if the cells are inserted into some kind of spatial data structure. By means of such a data structure a small number of candidate cells in the neighbourhood of the given point can be extracted. For these candidate cells local coordinates are computed by means of the outlined stencil walk approach. If all coordinates are in the range $0...1$ the correct cell has been found. The design of suitable spatial data structures will be discussed in Section 3.2.6.

A local point search can be implemented by applying the stencil walk method to all cells intersected by a straight line connecting point $\boldsymbol{p}$ and some other known point $\boldsymbol{q}$. The cells can be accessed sucessively by tracing the line through the cell's faces, determining entry and exit points in each case.

3.2.5 Triangular and Tetrahedral Grids

In the previous two sections we discussed interpolation methods for regular grids as well as means of computing local coordinates in quadrilateral and hexahedral cells. We know want to consider another important class of grids, namely irregular grids built from triangular and tetrahedral cells.

In a triangular cell it is most convenient to introduce a redundant set of three so-called *barycentric coordinates* ξ_0, ξ_1, ξ_2. These are defined in the following way:



$$
\begin{aligned}
\boldsymbol{p} &= \boldsymbol{p}_0 + \xi_1(\boldsymbol{p}_1 - \boldsymbol{p}_0) + \xi_2(\boldsymbol{p}_2 - \boldsymbol{p}_0) \\
&= \xi_0\boldsymbol{p}_0 + \xi_1\boldsymbol{p}_1 + \xi_2\boldsymbol{p}_2, \quad \xi_0 = 1 - \xi_1 - \xi_2
\end{aligned}
\tag{3.51}
$$

It is easy to see that the ξ_i are proportional to the area of the subtriangles defined by point $\boldsymbol{p}$ and an edge opposite to $\boldsymbol{p}_i$. Therefore, barycentric coordinates are sometimes also called *area coordinates*. In particular, we have

$$
\xi_0 = \frac{\text{Area } \boldsymbol{p}\boldsymbol{p}_1\boldsymbol{p}_2}{\text{Area } \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2}, \quad \xi_1 = \frac{\text{Area } \boldsymbol{p}_0\boldsymbol{p}\boldsymbol{p}_2}{\text{Area } \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2}, \quad \xi_2 = \frac{\text{Area } \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}}{\text{Area } \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2}. \tag{3.52}
$$

Given data values $\boldsymbol{f}_0$, $\boldsymbol{f}_1$, and $\boldsymbol{f}_2$ at the vertices of a triangle, a linear interpolant can be defined by means of barycentric coordinates in a straight-forward way. The data values are just combined in the same way as the coordinates itself, i.e.,

$$
\boldsymbol{f}(\boldsymbol{p}) = \xi_0\boldsymbol{f}_0 + \xi_1\boldsymbol{f}_1 + \xi_2\boldsymbol{f}_2. \tag{3.53}
$$

Barycentric coordinates of a point $\boldsymbol{p} = (x, y)^T$ can be computed via Eq. (3.51). Writing $\boldsymbol{p}_i = (x_i, y_i)^T$ and noting that the area of a planar triangle can be expressed by means of a cross product we obtain

$$
\begin{aligned}
\xi_1 &= \left((x - x_0)(y_2 - y_0) - (y - y_0)(x_2 - x_0)\right)/\Delta \\
\xi_2 &= \left((y - y_0)(x_1 - x_0) - (x - x_0)(y_1 - y_0)\right)/\Delta
\end{aligned}
\tag{3.54}
$$

with

$$
\Delta = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = 2 \text{ Area } \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2. \tag{3.55}
$$

The third coordinate ξ_0 is given by $1 - \xi_1 - \xi_2$. Whenever one of the three barycentric coordinates ξ_i is negative point $\boldsymbol{p}$ lies outside the triangle. Thus, computing these variables provides an efficient way of performing a point-in-triangle test.

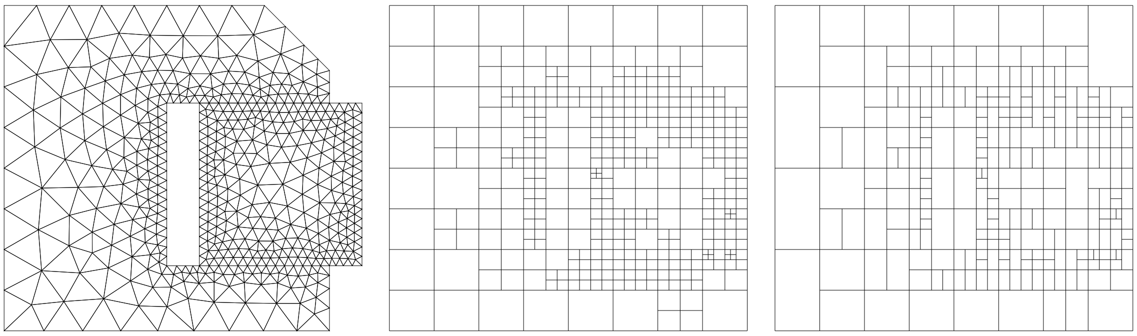
The concept of barycentric coordinates can be easily generalized to three-dimensional tetrahedral cells. In this case a fourth coordinate ξ_3 is required. The ξ_i are proportional to the volume of the subtetrahedra defined by point $\boldsymbol{p}$ and the face opposite to a vertex $\boldsymbol{p}_i$. Therefore, barycentric coordinates of a tetrahedron are sometimes also called *volume coordinates*. Moreover, of course also higher-order interpolation polynomials can be defined for both triangles and tetrahedrons. As with regular cells this requires some additional information, e.g. values provided at edges or faces of the cells [95].

Global and local point location strategies in triangular and tetrahedral grids are quite similar to the strategies described for regular grids with curvilinear coordinates. For a global method it is important to insert all cells into a suitable spatial data structure. This allows one to extract a small number of candidate cells in the neighbourhood of an arbitrary point. Then for all these candidate cells barycentric coordinates are computed until the correct cell is found. If the exact location of some other nearby point is known a local search strategy can be implemented by testing cells in the neighbourhood of the given cell. In particular, the next cell to be tested can be found by computing which face of the current cell a line segment connecting both points intersects.

3.2.6 Spatial Data Structures

At this point we shortly want to sketch data structures which can be used to speed up global point location in an unstructured or curvilinear grid. We already described how to check whether an arbitrary point $\boldsymbol{p}$ is contained in a given grid cell or not. The task of spatial data structures is to provide a small set of candidate cells in the neighbourhood of $\boldsymbol{p}$, thus accelerating point location significantly.

A large number of spatial data structures have been described in the literature [69]. Popular choices comprise quadtrees or octrees, kd-trees, or binary space partioning trees (BSP-trees). Often these data structures are designed to store a set of points. The general idea is to subdivide a contiguous space domain into smaller hierarchically ordered regions. Regions which are not subdivided anymore are called leafs. Usually points are inserted into exactly one leaf. However, in our case we need to process grid cells rather than



a) triangular grid b) quadtree c) $2d$-tree

Figure 3.5: Elements of an irregular or unstructured grid should be inserted into spatial data structure to speed up point location. In two dimensions for example quadtrees or $2d$-trees can be used.

points. Therefore we insert a cell into all leafs which possibly intersect a cell. For an arbitrary point $\boldsymbol{p}$ there is exactly one leaf $\boldsymbol{p}$ is contained in. The grid cells stored in this leaf provide the desired set of candidate cells for point location.

In our implementation we decided to insert grid cells into quadtrees or octrees, respectively. These data structures are simple and require little memory. They work by dividing a quadrilateral or hexahedral domain into four respectively eight subdomains of equal size. Whenever the number of cells stored in a particular domain exceeds a certain number this domain is subdivided further, causing the cells to be distributed into the newly created subdomains. Of course, subdivison only makes sense if the average size of the grid cells is much smaller than the size of the quadtree or octree domains. An example of a resulting 2D space partioning is shown in Figure 3.5 b). In this example at most 10 grid cells were allowed to be contained in a leaf. For case of simplicity we consider the bounding box of a grid cell in order to decide whether the cell should be inserted into a leaf or not. This makes testing much simpler but implies that some cells actually don't intersect the leaf's domain. The method is used for both triangular or curvilinear cells.

Instead of a quadtree or an octree also a binary kd-tree, i.e., a $2d$-tree, can be used. In this case a leaf is merely divided into two subregions instead of four or eight. Subdivison is performed perpendicular to the largest side of the subregion. kd-trees are advantagous if the size of the original domain along the major axes is very different. Moreover, kd-trees usually require slightly less memory since fewer leafs are produced. However, depth of the tree as well as average number of cells contained in a leaf are increased. Therefore point location is somewhat more expensive. An example of a $2d$-tree is shown in 3.5 c). While the domains of a quadtree, octree, or a kd-tree are aligned along the major axes for BSP-trees this is not the case. Instead, subregions are divided into two parts along an arbitrary plane. Because determining good bisections planes is not a simple task and because the memory required to explicitely store these planes is quite large we didn't investigate BSP-trees further.

Chapter 4

Line Integral Convolution in 2D

In this chapter we are going to discuss one particular kind of texture-based vector field visualization method, namely line integral convolution. We are using the term *texture* to indicate that directional information is not encoded by individual symbols – arrows or contour lines – but is inherently being contained in a pattern generated by some sort of stochastic process. Texture-based visualization methods are closely related to experimental techniques in flow visualization. We will investigate this relationship in more detail below. In contrast to a physical experiment, on a computer much more parameters can be changed yielding to a large variety of visual effects and concrete algorithms. The discussion of these concepts and the development of particular efficient algorithms will be the main subject of this chapter.

4.1 Experimental Techniques in Flow Visualization

Although directional quantities play an important role in many areas of science and engineering, the imaging of vector fields is of particular interest in the domain of experimental fluid mechanics. Van Dyck's *Album of Fluid Motion* presents an impressive collection of photographs and results [26]. In fact, flow visualization has a long tradition in physics. Pioneers like Prandtl already took photographs of flow patterns at the beginning of the century and used them for both qualitative and quantitative investigations. Since then the experimental setups became more sophisticated, and more accurate results could be obtained. However, the principal techniques remained the same until today [15]. In this section we will briefly sketch these methods, since they provide valuable background information for the development of computer-based visualization methods.

4.1.1 Randomly Dispersed Particles

Most experimental techniques in flow visualization are based on the insertion of external tracers into the fluid, i.e., particles or even molecules. A particularly useful method for studying laminar flows and wakes consists in recording lit solid particles randomly dispersed in the fluid. Typically tiny aluminium or magnesium cuttings are taken, but other

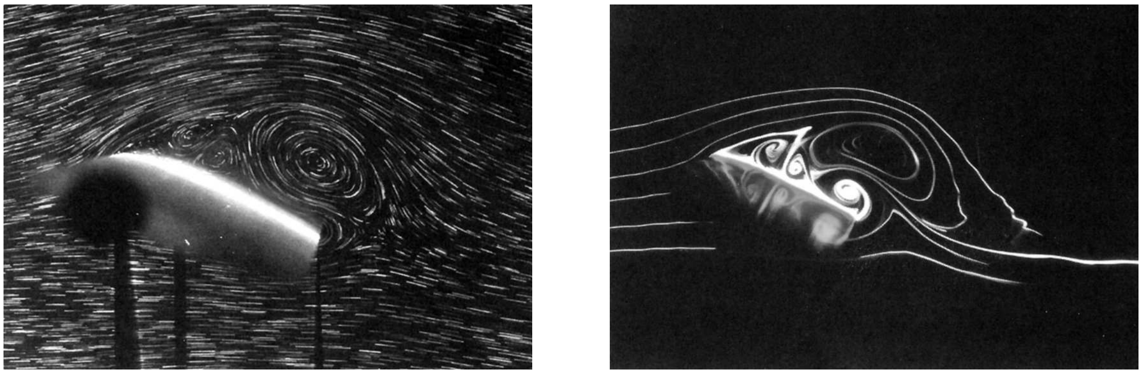


Figure 4.1: Comparative photographs of the starting flow past an airfoil. The left image shows streaks from small magnesium cuttings dispersed in the flow. The right image shows streak lines from an electrochemical dye continuously released into the flow (Images by P. Meyer and G. Pineau).

tracers like air or hydrogen bubbles can be used as well. The motion of the particles is captured during a short amount of time. If the flow field doesn't change too rapidly, a particle streak image is obtained which represents the velocity field of the flow at one particular instance of time. From this image the field's stream line pattern can be reconstructed by visual or numerical integration. The left part of Fig. 4.1 illustrates this approach. The image shows the flow around a wing profile rendered visible by individual particle streaks. It should be noted, that for more rapidly changing or even turbulent flows particle streak images are much harder to interpret. In this case an instantaneous stream line pattern cannot be obtained anymore. Instead of stream lines, actually particle trajectories are recorded which may cross each other in a complex way.

4.1.2 Continuous Dye Injection

A second commonly used technique in experimental flow visualization is based upon the continuous release of tracer material into the flow. Possible tracers are for example dye, electrochemical material, or smoke. In this way images of streak lines are obtained. While particle streak visualization yields an instananeous image of the flow, dye injection methods inform us about the convective transfer of vorticity occuring in the flow. The underlying reasons are given by Helmholtz's laws of vortex motion of 1858. In particular, the second of these laws states that – for an ideal inviscid fluid – particles on a vortex line at any instant will stay on a vortex lines at all subsequent times [13]. Thus, vorticity is convected in the same way as a material element in the fluid. In order to reveal the vortex structure of a flow, the general methodology is to ensure that the tracers become associated or entrained into the vorticity-bearing parts of the flow. The right part of Figure 4.1 illustrates the dye injection technique.

Even more than with particle streak images, much care has to be taken when actually analysing streak line photographs. There are several complications which may lead to a misinterpretation. First of all, real fluids are not, in general, inviscid. Therefore, the rate of diffusion of vorticity may be different from the rate of diffusion of the tracer.

Consequently, the presence or absence of vorticity cannot always be concluded from the distribution of tracer material. A second problem is, that the concentration of tracer material cannot cancel, whereas vortices of opposite sign could. Moreover, even if there is a positional correlation of vorticity and tracer material, in general there will be no correlation between the amount of vorticity and the concentration of tracer material.

4.1.3 Other Methods

Beside particle streak visualization, i.e., the imaging of stream lines, and dye injection, i.e., the imaging of streak lines, there are other techniques in flow visualization, which should be mentioned, too. The development of so-called material lines or time lines can be investigated by releasing tracer particles instantaneously from a line source, e.g. a wire, and taking photographs at regular intervals. This method is particularly useful for obtaining velocity profiles of the flow. In principle, by taking photographs with long exposure times also pathlines or trajectories of individual particles may be recorded. However, for unsteady flows this method is rarely applied because the resulting images are very complex and difficult to understand.

In addition to the volumetric methods mentioned above, the analysis of surface flow patterns represents a valueable source of information, too. Such patterns can be recorded by coating the surface of a solid body in the flow with a thin layer of material, e.g. an oil film. In this way separation lines and reattachment points become visible, and the transition from laminar flow to turbulence can be studied. In Chapter 5 we discuss, how surface flow visualization techniques can be simulated on a computer.

While all of the techniques discussed so far rely on the insertion of forgein material into the flow, optical flow visualization methods exploit variations of optical properties of the fluid itself. These methods are mainly applied for studying high-speed compressible flows. In particular, the refractive index in such flows is correlated to mass density, and thus can be used to obtain information about pressure and velocity. Variations of the refractive index can be detected either by so-called shadow graphs, by analysing schlieren patterns, or by using interferometric methods. Shock waves in the supercritical flow regime are nicely shown by these methods.

Finally, we would like to point out that not all vector fields are flow fields and that there are also experimental imaging techniques outside the domain of flow visualization. Perhaps the first visualization experiments have been performed to explore the nature of electricity. From the physics lessons in school we know that small grain particles on an oil film arrange in characteristic line patterns in the neighbourhood of charged metallic objects. In fact, experiments like this inspired Faraday to introduce the concept of a field in science [64].

4.2 Particle Streak Visualization

In this section we want to look at particle streak experiments in fluid mechanics from a mathematical point of view. From all the experimental techniques outlined above, particle streak visualization gives the most intuitive understanding of field structure. In particular, it is well suited to reveal topological information about a vector field. By looking at particle streak images we will find a motivation for two important classes of digital vector field visualization algorithms, namely spot noise and line integral convolution.

4.2.1 Image Formation

Let us consider a single particle dispersed in a fluid flow. For an infinitesimal small exposure time dt the particle will produce an image of intensity $A(\boldsymbol{x}-\boldsymbol{x}_0)\,dt$ on a photographic film. Here $\boldsymbol{x}_0$ denotes the position of the particle at that time instance. Noting, that intensity is measured per area the function A must have the dimension of a flux density $[\mathrm{m}^{-2}\,\mathrm{sec}^{-1}]$. We call A a *particle spread function*, since it is very similar to the point spread function used in microscopy []. In our notation we neglect, that this function may also depend on parameters other than the particle's position, e.g. orientation or velocity.

A camera will capture a particle while it is moving on a path $\boldsymbol{x}_i(t), t \in [t_1, t_2]$. The time difference $t_2 - t_1$ just corresponds to the total exposure time. Thus, the image of a single particle can be written as

$$h_i(\boldsymbol{x}) = \int_{t_1}^{t_2} A(\boldsymbol{x} - \boldsymbol{x}_i(t))\,dt. \tag{4.1}$$

Ignoring occlusion effects, the image intensity obtained by recording many particles simultaneously is simply given by the superposition of the individual intensities, i.e.,

$$I(\boldsymbol{x}) = \sum_i h_i(\boldsymbol{x}). \tag{4.2}$$

4.2.2 Spot Noise

Of course, the total image intensity (4.2) is proportional to the number of particles in the fluid. In practice, we would rather like to consider a normalized intensity independent from this number, but scaled so that the resulting image has certain contrast. Physically such a scaling can be implemented by adjusting the amount of incoming light, the camera aperture, the film sensitivity, or the kind of film processing. Thus we have

$$\bar{I}(\boldsymbol{x}) = I_0 + c \sum_i (h_i(\boldsymbol{x}) - \langle h \rangle), \tag{4.3}$$

where I_0 denotes the mean image intensity, c is a constant factor controlling the contrast of the resulting image, and $\langle h \rangle$ is the mean value of h_i over all particles. Assuming that the particles are distributed equally in space, $\langle h \rangle$ does not depend on $\boldsymbol{x}$. Further assuming that the values $h_i(\boldsymbol{x})$ for different particles are statistically independent, it follows from the central limit theorem of statistics that the intensity distribution at a given location $\boldsymbol{x}$ is a Gaussian one. The same distribution can also be obtained by

$$\bar{I}(\boldsymbol{x}) = I_0 + \sum_i a_i\, h_i(\boldsymbol{x}) = I_0 + \sum_i a_i \int_{t_1}^{t_2} A(\boldsymbol{x}_0 - \boldsymbol{x}_i(t))\, dt, \tag{4.4}$$

where a_i denotes a particle weighting factor obeying $\langle a \rangle = 0$. Since these weighting factors do not depend on $\boldsymbol{x}$ the correlation of the final intensity values will be controlled by $h_i(\boldsymbol{x})$ alone. Thus, from a statistical point of view (4.3) and (4.4) are equivalent.

Eq. (4.4) can be used to synthesize artifical particle streak images on a computer. In vector field visualization this approach is known as *spot noise* [90]. The term *spot* refers to the image or streak h_i of an individual particle. Spots from many particles distributed in a random way are combined using positive or negative weights a_i. The function h_i may be approximated in different ways, giving rise to a large variety of slightly different spot noise algorithms. With more general functions h_i the method can be used to synthesize general stochastic textures, e.g. clouds or marble patterns.

4.2.3 Line Integral Convolution

In the following we want to derive a different algorithm which can also be used to approximate particle streak images, namely line integral convolution. We start from the observation that if many particles are present the discrete set of weights a_i can be replaced by a function $a(\boldsymbol{x})$ defined on a 2D domain Ω. Instead of Eq. (4.4) we thus have

$$I(\boldsymbol{x}_0) = \int_\Omega a(\boldsymbol{y}) \int_{t_1}^{t_2} A(\boldsymbol{x}_0 - \boldsymbol{x}_{\boldsymbol{y}}(t))\, dt\, d\boldsymbol{y}. \tag{4.5}$$

Here $\boldsymbol{x}_{\boldsymbol{y}}(t)$ denotes the path of a particle initially being located at $\boldsymbol{y}$, i.e., $\boldsymbol{x}_{\boldsymbol{y}}(0) = \boldsymbol{y}$. In order to simplify this equation let us assume that the flow field doesn't change in time. In this case the particle paths don't intersect. In fact, the set of integral curves $\boldsymbol{\sigma}$ of the flow field completely covers Ω in a disjunct manner. We may therefore introduce a local coordinate system aligned with the integral curves and parametrize a region of interest by $\boldsymbol{y} = \boldsymbol{\sigma}(u; v)$. Here u represents ordinary time, while v parametrizes the domain in the direction perpendicular to the flow. Using this parametrization, we have

$$\boldsymbol{x}_0 = \sigma(u_0; v_0) \quad \text{and} \quad \boldsymbol{x}_{\boldsymbol{y}}(t) = \boldsymbol{\sigma}(u + t; v). \tag{4.6}$$

In the most simple case the particle spread function A is just a delta function. This yields to

$$A(\boldsymbol{x}_0 - \boldsymbol{x}_{\boldsymbol{y}}(t)) = A(\boldsymbol{\sigma}(u_0; v_0) - \boldsymbol{\sigma}(u + t; v)) = \delta(u_0 - u - t)\, \delta(v_0 - v).$$

We insert this expression into Eq. (4.5) and obtain

$$I(\boldsymbol{x}_0) = \iint a(\boldsymbol{\sigma}(u; v)) \int_{t_1}^{t_2} \delta(u_0 - u - t)\, \delta(v_0 - v)\, dt\, dv\, dv. \tag{4.7}$$

Introducing $\boldsymbol{\sigma}(u) \equiv \boldsymbol{\sigma}(u; v_0)$, this expression can be simplified to

$$I(\boldsymbol{x}_0) = \int g(u_0 - u)\, a(\boldsymbol{\sigma}(u))\, du, \quad \text{with} \quad g(u_0 - u) = \int_{t_1}^{t_2} \delta(u_0 - u - t)\, dt. \tag{4.8}$$

This equation states that the intensity $I(\boldsymbol{x}_0)$ in a particle streak image is given by a one-dimensional convolution of two functions g and a along an integral curve of the field passing through $\boldsymbol{x}_0$. More precisely, g turns out to be a box-shaped filter kernel, while the function $a(\boldsymbol{x})$ describes the particle distribution within the fluid. This function has been defined as the continuous version of the discrete particle weights a_i occuring in the spot noise. Usually, $a(\boldsymbol{x})$ will be some sort of random noise function. Again, Eq. (4.8) can be used to generate synthetic particle streak images on a computer. In scientific visualization this approach is known as *line integral convolution* (LIC).

Line integral convolution can be interpreted as a special kind of spot noise technique associated with a δ-shaped particle spread function. In effect, the image intensity at a given location can be computed by a one-dimensional convolution instead of a general two-dimensional one, which would be computationally much more expensive. Later on we describe how to design efficient algorithms for computing LIC image very quickly. While the δ-character of the particle spread function perpendicular to the flow is an inherent feature of line integral convolution, the method may well handle more general functions parallel to the flow. As a result, the box-filter g in Eq. (4.8) will be replaced by more complex kernel shapes.

4.3 LIC as an Image Operator

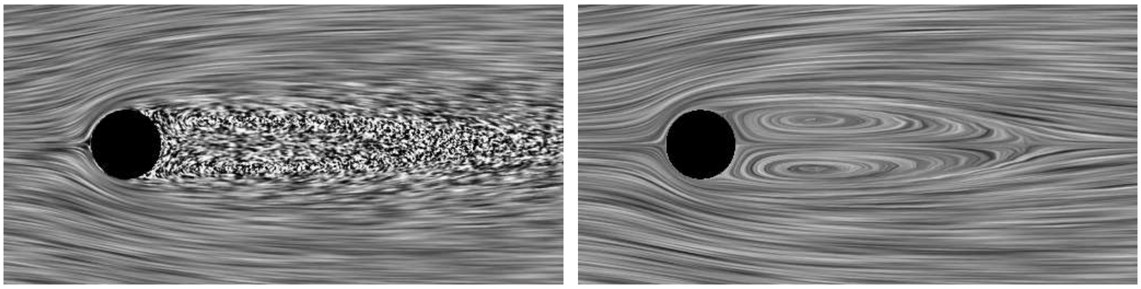
In the previous section we introduced line integral convolution (LIC) as a method for generating artificial particle streak images on a computer. However, LIC is not only restricted to flow visualization but can be used to synthesize directional textures in general. In this respect, it is of interest in areas like image processing or digital art. Therefore, we now want to reconsider LIC under a more general point of view.

4.3.1 Definition

Formally, LIC may be defined as an operator which takes an input image T, a vector field $\boldsymbol{f}$, and a filter kernel g as input and produces an output image I. It does so by computing the one-dimensional convolution of T and g along the integral curves $\boldsymbol{\sigma}$ of $\boldsymbol{f}$:

$$I(\boldsymbol{x}_0) = \int g(u_0 - u)\, T(\boldsymbol{\sigma}(u))\, du = g * T(\boldsymbol{\sigma}), \qquad \boldsymbol{x}_0 = \boldsymbol{\sigma}(u_0) \tag{4.9}$$

Of course, this definition requires that the integral curves $\boldsymbol{\sigma}(u)$ exist and can be defined in an unambiguous way. If this is not the case, some special treatment is necessary. For example, at singularities we may simply truncate the convolution interval or we may extend the integral curves artificially. Notice also, that in general the integral curves $\boldsymbol{\sigma}(u)$



a) time parametrization b) arc-length parametrization

Figure 4.2: On the left a LIC image with field lines parametrized by time is shown. The result resembles a photograph from a real particle streak experiment. However, the directional structure of the field in areas of low velocity remains unclear. This disadvantage can be circumvented if field lines are parametrized by arc-length, as shown on the right.

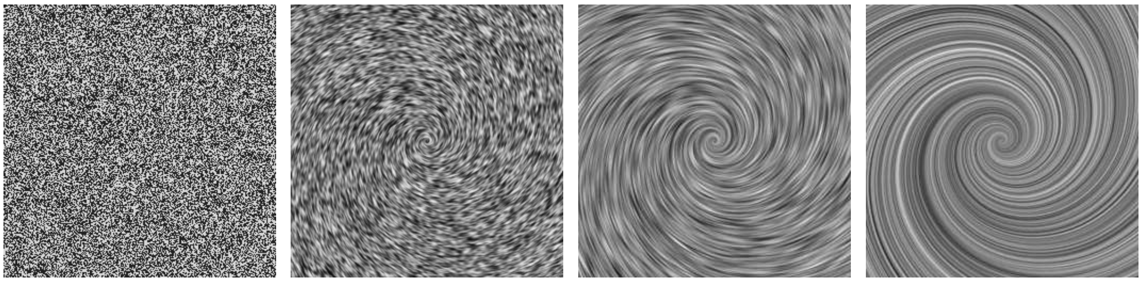
will leave the domain Ω on which the output image is to be computed. Therefore, the input image T and the vector field V need to be defined on a bigger domain Ω^+. In practice, we will usually determine T and V outside Ω by some kind of continuation method, for example periodic tiling, clamping, or bilinear extrapolation.

Line integral convolution was proposed for the first time by Cabral and Leedom in 1993 [9]. Although their pioneering work did not contain a general equation like (4.9), it showed how to apply LIC in scientific visualization and digital painting. The LIC algorithm described by Cabral and Leedom relied on a simple numerical integration method and was rather slow and inflexible. We will review this approach in Section 4.3.4 before developing a more efficient way to compute LIC images.

In order to simulate particle streak images, the input image T should comprise some sort of random noise image. Due do the convolution process, in the final LIC image the pixel correlation in field line direction will be much higher than in perpendicular direction. Thus, the directional structure of the field becomes visible. However, other input images may be processed as well. In this case, LIC may be used for example to simulate motion blur effects. Because of the variety of possible effects and the generality of the input T and $\boldsymbol{f}$ LIC can be interpreted as a general image proessing operator instead of a special flow visualization technique.

4.3.2 Parametrization

An important aspect of line integral convolution is the way how the integral curves $\boldsymbol{\sigma}(u)$ are parametrized. If we want to imitate real world particle streak images the natural parametrization is adequate where u equals time t and the vector value $\boldsymbol{f}(\boldsymbol{x})$ denotes the velocity of a particle at location $\boldsymbol{x}$. This natural parametrization means that the spatial extent of the convolution interval gets larger in areas with large vector magnitude, i.e., high velocity. This is evident since fast particles produce long streaks, while slow particles produce short streaks. However, as a consequence the directional structure of a flow field is often hard to understand in regions of low velocity. In real experiments this shortcoming



$L = 0$ $L = 4$ $L = 16$ $L = 250$

Figure 4.3: This figure illustrates the effect of varying the length of the LIC filter kernel. In the left-most image the original input noise image is shown. The larger the filter length the more coherent the structures in the final LIC image. Contrast has been adjusted so that the variance of the input noise is maintained. In all examples arc-length parametrization has been used.

can hardly be overcome. However, on a computer we may simply choose a different parametrization. For many purposes it is convenient to choose arc-length s as a parameter. Arc-length of a curve is defined by

$$s = \int ds, \quad \text{with} \quad ds = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}\, dt. \tag{4.10}$$

Since this is a strictly monotonously increasing function, the reparametrization of $\boldsymbol{\sigma}(t)$ is well-defined, except in areas of vanishing velocity. Noting that $dt/ds = |\boldsymbol{f}|^{-1}$ we have

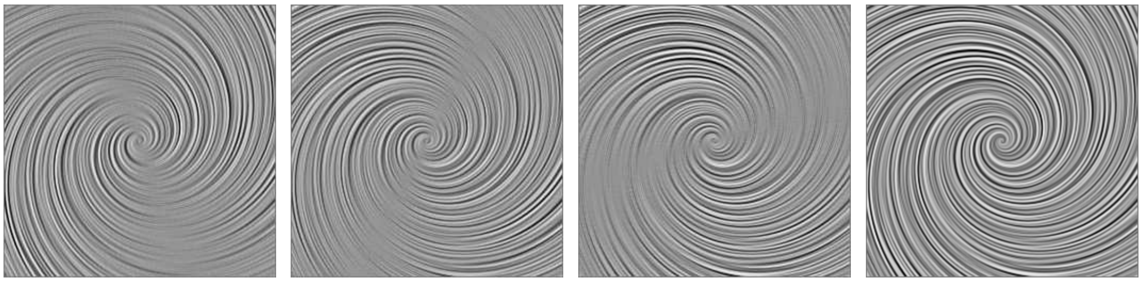
$$\frac{d}{ds}\boldsymbol{\sigma}(s) = \frac{d\boldsymbol{\sigma}}{dt}\frac{dt}{ds} = \frac{\boldsymbol{f}}{|\boldsymbol{f}|}. \tag{4.11}$$

Using this parametrization the spatial extent of the convolution interval is constant across the whole image. In Figure 4.2 the effects of using time t or arc-length s as a parameter are compared. While in the first case the image closely resembles a real particle streak photograph, in case of arc-length parametrization the structure of the field in areas of low velocity becomes more obvious. In order to visualize vector magnitude simultaneously other techniques like color coding or texture animation can be employed. Details will be discussed below.

4.3.3 Additional Remarks

Line integral convolution can be used to produce a variety of different effects. At this point we want to give an overview of how these effects can by obtained, either by some sort of pre- or post-processing, or by changing parameters of the LIC operator directly.

Filter Length. Asume that we compute LIC images with arc-length parametrization. The characteristic feature size in the resulting images is directly proportional to the spatial extent of the filter kernel being used. For vector field visualization usually a random noise image is taken as input. Very small filter lengths modify the noise image only little. The



$\phi = 0^\circ$ $\phi = 45^\circ$ $\phi = 90^\circ$ adaptive

Figure 4.4: The appearance of raw LIC images can be improved in a post-processing step. In the first three examples directional derivatives along a predefined direction have been computed. In the image on the very right this direction has been locally adapted so that it is always perpendicular to the vector field.

directional structure of the field becomes more evident if a larger filter length is chosen. However, in this case also contrast of the resulting LIC image is reduced. More precisely, the standard deviation of the greylevel histogram is decreased. In the limit of an infinitely large filter kernel a constant grey image would be obtained. For large but finite filter lengths contrast of the final LIC image can be preserved by scaling the intensity values in the right way. Details about contrast adjustment as well as an intensity correlation analysis are presented in the following sections. Examples of LIC images computed using varying filter lengths are shown in Figure 4.3.

Embossing. LIC images computed from random noise input textures clearly reveal the structure of the underlying vector field. However, the appearance of these results can be significantly improved by applying additional digital filters in a post-processing step. In particular, so-called *relief filtering* or *embossing* provides good results. Embossing essentially relies on computing directional derivatives of image grey values along a predefined direction. A fast and simple implementation of an emboss filter is described in [75]. Examples of embossed LIC images are shown in Figure 4.4. One disadvantage of relief filtering is that the directional structure of the LIC image gets suppressed in areas aligned parallel to the direction of the emboss operator. To prevent this drawback derivatives can always be computed perpendicular to vector field direction. This is illustrated in the rightmost image shown in Figure 4.4. Notice, that this image doesn't look three-dimensional anymore. Instead, contrast of the original LIC image is emphasized in all areas.

Color Images. The LIC operator can be easily defined for color images, too. In this case the convolution operation (4.9) simply has to be applied separately to each channel of a color image. Instead of a single value I three independent intensities are computed, representing the red, green, and blue components of a color pixel, respectively. In order to emphasize the directional structure of the resulting color LIC image, noise can be added to the input image. Examples are shown in Figure 4.5.

Another way to compute colored LIC images is to combine a three-component RGB image with a standard grey value LIC texture in a post-processing step. For example, the color image can be used to visualize an additional scalar quantity via pseudo-coloring.

a) original image

b) LIC vector field derived from 2 point charges

c) original image with noise added

d) LIC vector field derived from image gradient

Figure 4.5: Instead of a random noise image also arbitrary color images can be processed usng the LIC operator. Suitable vector fields may either be designed manually or may be computed from the input image itself. If noise is added to the input image the resulting image looks more like conventional LIC images.

Examples have already been shown in the introductory chapter. In order to combine a color image and a simple LIC texture it is usually sufficent to multiply each component of the color image with a factor proportional to the intensity of the single component LIC texture. Modern paintbox programs like Adobe Photoshop provide many alternative methods of combining two such inputs in a meaningful way. However, for scientific visualization these methods are usually not adequate.

Vector Field Design. Our last remark is concerned to the type of vector field being used for the LIC operation. In scientific visualization this field is obtained from a numerical simulation or a measurement. However, if LIC is applied in digital art, usually interesting vector fields have to be designed manually. For example, in Figure 4.5 b) the vector field has been computed from point charges placed manually in the image. The charges gave rise to a Coulomb field, which then has been rotated by a predefined angle.

Another approach is to extract the vector field from the input image T itself. For example, the gradient of the image function T might be computed. The gradient vectors can again be rotated by some arbitrary angle ϕ. A rotation of 90 degrees means that T can be interpreted as a stream function or Hamiltonian function. The resulting vector field then contains closed orbits. Figure 4.5 d) shows a LIC image with a vector field computed from the gradient of a blurred version of the input image T. Blurring is necessary because otherwise a very noisy discontinuous vector field would result.

4.3.4 The Algorithm of Cabral and Leedom

Before discussing our own algorithm for computing LIC images, let us briefly sketch the original approach of Cabral and Leedom. Their algorithm requires the vector field V to be defined on the same uniform grid as the input image T. Consequently, for each pixel p of the input image both an intensity I_p as well as a vector $\boldsymbol{f}_p$ are stored. Both, intensity and vector field are assumed to be constant within a pixel. LIC intensities are computed separately for all pixels. Starting from the center of a pixel, stream lines $\boldsymbol{\sigma}(s)$ are followed up to a distance L in positive and negative direction. Each curve $\boldsymbol{\sigma}(s)$ is decomposed into a number of straight line segments of length Δs_i. The breakpoints of the straight line segments are determined such that they are located exactly at the pixel boundaries.

For each straight line segment the corresponding contribution to the convolution integral is computed. Since intensity is constant within a pixel, merely the integral of the filter kernel along the line segment has to be determined, i.e.,

$$g_i \equiv \int\limits_{s_i}^{s_i+\Delta s_i} g(s)\,ds, \quad \text{with } s_0 = 0 \text{ and } s_{i+1} = s_i + \Delta s_i. \tag{4.12}$$

Let N_b be the number of segments in backward direction, N_f the number of segments in forward direction, and T_i the input intensity corresponding to the i-th segment. Then the final normalized LIC intensity for a pixel is given by

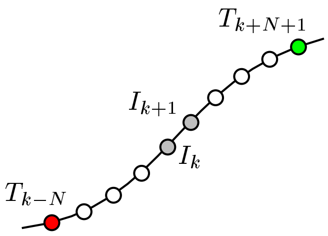
$$I_{\text{LIC}} = \frac{\sum\limits_{-N_b \le i \le N_f} g_i\, T_i}{\sum\limits_{-N_b \le i \le N_f} g_i}. \tag{4.13}$$

While this algorithm is conceptually very easy, Cabral and Leedom pointed out that some care has to be taken in the actual implementation. In particular, the computation of entry and exit points of a stream line through a pixel has to be performed in a robust and consistent way. Likewise, the detection of singularities, i.e., sinks and sources, requires special care since the vector field is assumed to be constant inside a pixel. Another restriction of the outlined algorithm is that vector field and input image have to be defined on the same grid. Finally, from the algorithmic point of view it should be noted that processing each pixel separately implies a large amount of redundant work. Typical computing times of the algorithm are in the range of tens of seconds and more.

4.3.5 Exploiting Intensity Coherences

Compared to other vector field visualization techniques line integral convolution produces highly intuitive results well suited to be processed by the human visual system. However, in order to be really competitive to other methods it is very desireable to improve the performance of Cabral and Leedoms's original algorithm. This can be done by strictly avoiding redundant work when computing LIC images. In the original algorithm there are two types of redundancies. First, for neighbouring pixels very similar field lines are computed. Second, almost identical convolution sums are recomputed again and again. Both types of redundancies are closely related.

The basic idea of the fast LIC algorithm presented below is to exploit intensity coherences along a field line. This can be achieved most easily for a simple box filter. In this case LIC intensity can be simply approximated by an average of $2N + 1$ different samples of the input image located on a common field line. Now LIC intensity I_k at one sample location is almost identical to LIC intensity I_{k+1} at a neighbouring sample. In order to obtain the correct value we may simply *update* I_k instead of recomputing I_{k+1} from scratch. This can be done in the following way:



$$I_k = 1/(2N+1) \sum_{-N \le i \le N} T_i$$

$$I_{k+1} = I_k + (T_{k+N+1} - T_{k-N})/(2N+1)$$

Intensity update for a box filter involves just one addition and one subtraction, ignoring normalization of LIC intensity. In the following we first show how similar intensity updates can be performed for more general filter kernels. In order to compute a full 2D LIC image we will then compute a dense set of field lines distributed homogeneously throughout the whole input image. Details of this approach will be presented below. In summary, by exploiting intensity coherences, i.e., by performing intensity updates, a fast LIC algorithm can be formulated which turns out to be one order of magnitude faster than the original approach of Cabral and Leedom [81].

4.4 Rewriting the Convolution Integral

In order to derive a general method for performing intensity updates we need to rewrite the convolution integral (4.9). A particular efficient update scheme can be designed for the class of piecewise polynomial filter kernels. Piecewise polynomial functions provide a very general function class, allowing us to represent a wide variety of kernel shapes. In the following we will first briefly introduce piecewise polynomial functions formally and introduce a basis of the corresponding linear space. In our discussion we closely follow the one presented in [18]. Afterwards we state an important convolution theorem and consider particular filter kernels in detail.

4.4.1 Piecewise Polynomial Functions

Given a strictly increasing sequence $\xi \equiv (\xi_i)_{i=1,...,l}$ of knots $\xi_i \in \mathbb{R}$ and polynomials P_i, $i = 1, ..., l$, of order k (i.e., of degree $< k$), a *piecewise polynomial function* of *order* k can be defined by

$$f(x) := \begin{cases} 0, & x < \xi_1 \\ P_i(x), & \xi_i \leq x < \xi_{i+1}\,, \quad i = 1 \ldots l-1 \\ P_l(x), & x \geq \xi_l\,. \end{cases} \tag{4.14}$$

The function and its derivatives may or may not be continuous at the knots ξ_i. The definition is illustrated in Figure 4.6. It is easy to see that the set of piecewise polynomial functions of order k for a fixed knot sequence ξ generates a linear space. We will call this space $\mathbb{P}_{k,\xi}$. In order to simplify further discussion we allow the function f to take non-vanishing values right from the last knot ξ_l, while it vanishes identically left from the first knot ξ_1. Later, when using piecewise polynomial functions as filter kernels, we require the right-most polynomial P_l (dashed in Figure 4.6) to be zero, since the kernels must have finite support.

A basis for the space of piecewise polynomial functions can be built from so-called truncated power functions. These functions are ordinary monomials for $x \geq 0$, but vanish for negative x. We denote them by

$$(x)_+^r = \begin{cases} 0, & x < 0 \\ x^r, & x \geq 0 \end{cases} \qquad \text{for } r \geq 0. \tag{4.15}$$

The truncated power basis of $\mathbb{P}_{k,\xi}$ is given by the double sequence of functions

$$\phi_{ij}(x) = \frac{(x - \xi_i)_+^{(j-1)}}{(j-1)\,!}, \quad i = 1, ..., l \text{ and } j = 1, ..., k. \tag{4.16}$$

The ϕ_{ij} are piecewise polynomial functions of order j with just one knot ξ_i. Obviously they are elements of $\mathbb{P}_{k,\xi}$. Note, that this would not be the case if we had required $P_l(x)$ to be zero in Eq. (4.14). Plots of some ϕ_{ij} are shown in Figure 4.7. The derivatives of ϕ_{ij} obey the relation

$$\frac{d}{dx}\,\phi_{ij} = \phi_{i,j-1}, \qquad j > 1. \tag{4.17}$$

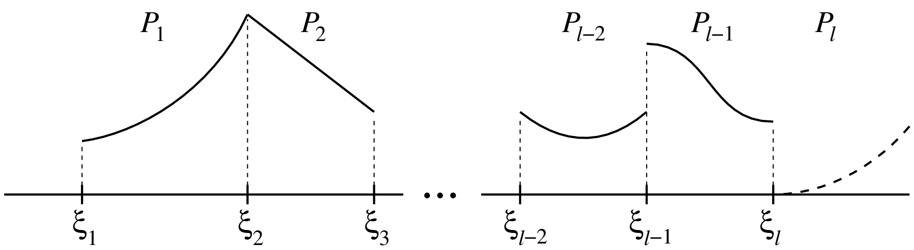


Figure 4.6: Piecewise polynomial functions are defined by a set of l knots ξ_i and l polynomials P_i. When constructing LIC filter kernels we require the right-most polynomial P_l to be zero.

For later use we note that the derivative of the step functions $\phi_{i,1}$ can be defined in the sense of distribution theory by means of Dirac's delta function. Keeping this in mind and applying Eq. (4.17) recursively, we find the following important relationship:

$$\frac{d^j}{dx^j}\,\phi_{i,j}(x) \;=\; \delta(x-\xi_i), \qquad j \geq 1 \tag{4.18}$$

Before we continue, let us prove that the set of functions ϕ_{ij} really forms a basis of $\mathbb{P}_{k,\xi}$. We first look at the linear functionals λ_{ij} that take the difference of the $(j-1)$th derivative of f at a breakpoint ξ_i:

$$\lambda_{ij} f = \text{jump}_{\xi_i}\, f^{(j-1)} = f^{(j-1)}(\xi_i^+) - f^{(j-1)}(\xi_i^-), \tag{4.19}$$

with $i = 1, ..., l$ and $j = 1, ..., k$. Applying λ_{ij} to ϕ_{ij} yields

$$\lambda_{ij}\phi_{rs} = \text{jump}_{\xi_i}\, \frac{d^{(j-1)}}{dx^{(j-1)}} \frac{(x-\xi_r)_+^{(s-1)}}{(s-1)\,!} = \delta_{ir}\delta_{js}. \tag{4.20}$$

As is obvious from Figure 4.7 this expression vanishes always except if the knot indices i and r as well as the exponents j and s are equal. We can use this result to show that the ϕ_{ij} are linear independent. Linear independence means that the equation $\sum c_{ij}\phi_{ij} = 0$ must imply $c_{ij} = 0$. In fact, this can be shown by applying λ_{ij} to an arbitrary linear combination,

$$\lambda_{ij} \sum c_{rs}\,\phi_{rs} = \sum c_{rs}\,\lambda_{ij}\,\phi_{rs} = \sum c_{rs}\,\delta_{ir}\,\delta_{js} = c_{ij} = 0. \tag{4.21}$$

The dimension of the space $\mathbb{P}_{k,\xi}$ is kl, which is equal to the number of functions ϕ_{ij}. Together with the linear independence this dimensional argument shows that the ϕ_{ij} really comprise a basis for piecewise polynomial functions. Consequently, every $f \in \mathbb{P}_{k,\xi}$ has a unique representation of the form

$$f = \sum_{ij} (\lambda_{ij} f)\,\phi_{ij} = \sum_{ij} (\text{jump}_{\xi_i}\, f^{(j-1)})\,\frac{(x-\xi_i)_+^{(j-1)}}{(j-1)\,!}. \tag{4.22}$$

For many applications a more suitable and numerically advantageous basis of $\mathbb{P}_{k,\xi}$ is given by so-called B-splines [18, 22]. However, as we will see, the truncated power basis allows us to rewrite convolution integrals in an elegant way, thus facilitating the design of a general fast LIC algorithm. Therefore we will expand all filter kernels, even B-splines, in the truncated power basis.

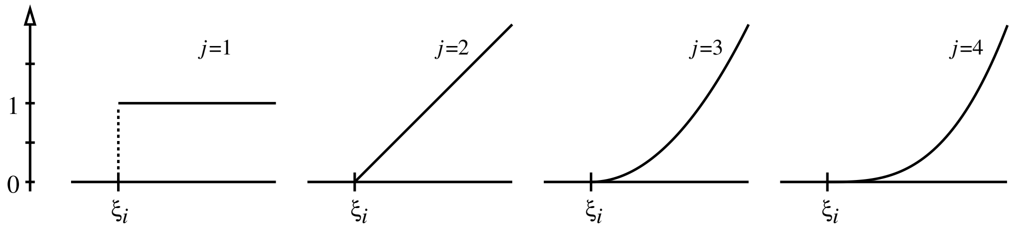


Figure 4.7: The truncated power basis ϕ_{ij} consists of oridinary monomials clipped to be zero left from a breakpoint ξ_i. Note, that the derivative of ϕ_{ij} is just given by $\phi_{i,j-1}$.

4.4.2 A Convolution Theorem

The single important theorem which allows us to formulate our fast LIC algorithm is the following. The convolution of two functions f and h, where h has finite support, is equal to the convolution of the integral F of f and the derivative h' of h, i.e.,

$$f*h = \int_{-\infty}^{\infty} f(y)\, h(x-y)\, dy \;=\; \int_{-\infty}^{\infty} F(y)\, h'(x-y)\, dy = F*h', \tag{4.23}$$

for all $x \in \mathbb{R}$. To show this we consider a definite integral with finite bounds first, apply integration by parts, let the integration bounds move to infinity, and then use the fact that the filter kernel h has finite support:

$$\int_{-z}^{z} F(y)\, h'(x-y)\, dy = F(y)\, h(x-y)\,\Big|_{y=-z}^{y=z} + \int_{-z}^{z} f(y)\, h(x-y)\, dy \;\underset{z\to\infty}{\longrightarrow}\; f*h \tag{4.24}$$

This is valid for every finite x. Assuming that f is at least n times integrable and that h is at least n times differentiable, we may apply the above theorem n times repeatedly. In order to simplify notation we introduce $F^{(0)} \equiv f$ and define *repeated integrals* of f recursively by

$$F^{(n)}(x) = \int_{\infty}^{x} F^{(n-1)}(y)\, dy, \qquad \text{for } n \geq 1. \tag{4.25}$$

Using this notation we find the important relation

$$f * h = F^{(n)} * h^{(n)}\,. \tag{4.26}$$

By means of suitable test functions it can be shown that this expression is also valid if the n th derivative $h^{(n)}$ is a delta distribution.

Suppose the filter kernel h is a piecewise polynomial function written in truncated powers, i.e., $h = \sum_{ij} c_{ij}\phi_{ij}$. Recall from Eq. (4.18) that the j-th derivative of ϕ_{ij} is a delta function. Thus, by applying equation (4.26) with $n=j$ the convolution integral can be expressed in terms of repeated integrals as follows:

$$\begin{aligned} f*h &= \int_{-\infty}^{\infty} f(y)\, h(x-y)\, dy \;=\; \sum_{ij} c_{ij} \int_{-\infty}^{\infty} f(y)\, \phi_{ij}(x-y)\, dy \\ &= \sum_{ij} c_{ij} \int_{-\infty}^{\infty} F^{(j)}(y)\, \delta(x-\xi_i-y)\, dy \;=\; \sum_{ij} c_{ij}\, F^{(j)}(x-\xi_i) \end{aligned} \tag{4.27}$$

4.4.3 Filter Kernels of B-Spline Type

Before we discuss how to build a fast LIC algorithm by discretizing Eq. (4.27), let us first look at some particular filter kernels. In principle we may apply our technique to any piecewise polynomial filter kernel. However, for line integral convolution the filter kernel should be as simple as possible. There is no reason why to consider unsymmetric kernels. Furthermore, smoother results are obtained if the kernel is less weighted at its boundaries. We will make the notion of "smoothness" more precise by analysing intensity correlation later on.

A nice class of filter kernels fulfilling these conditions are centered B-splines with uniform knot sequences. The most simple element of this class is a box filter b_1 with just two knots at $-L$ and L, normalized such that $\int b_1\,dx = 1$. Higher order filters can be obtained by repeatedly convolving box filters. Convolving two box filters results in a triangle or hat filter $b_2 = b_1 * b_1$. In contrast to the box the triangle filter is continuous but still has discontinuous derivative. Even smoother filters are obtained by convolving the triangle with a box again, and so on. In this way we get filter kernels of B-spline type. In particular, an n-th order B-spline filter is given by

$$b_n = \frac{1}{(2L)^n} \sum_{i=0}^{n} (-1)^i \binom{n}{i} \frac{\big(x + (n-2i)L\big)_+^{n-1}}{(n-1)\,!}, \quad n \geq 1. \tag{4.28}$$

Let us prove this expression by induction. For $n = 1$ we obtain the box filter itself:

$$b_1 = \frac{1}{2L}\,\big((x+L)_+^0 - (x-L)_+^0\big) \tag{4.29}$$

Assuming that Eq. (4.28) is valid for n we get for $b_{n+1} = b_n * b_1$:

$$\begin{aligned} b_{n+1} &= \frac{1}{(2L)^{n+1}} \sum_{i=0}^{n} (-1)^i \binom{n}{i} \int_{-\infty}^{\infty} \frac{\big(x+(n-2i)L\big)_+^{n-1}}{(n-1)\,!} \big((x-y+L)_+^0 - (x-y-L)_+^0\big)\,dy \\ &= \frac{1}{(2L)^{n+1}} \sum_{i=0}^{n} (-1)^i \binom{n}{i} \Big(- \frac{\big(x+L+(n-2i)L\big)_+^n}{n!} + \frac{\big(x-L+(n-2i)L\big)_+^n}{n!} \Big) \\ &= \frac{1}{(2L)^{n+1}} \sum_{i=0}^{n} (-1)^i \binom{n}{i} \Big(- \frac{\big(x+(n+1-2(i+1))L\big)_+^n}{n!} + \frac{\big(x+(n+1-2i)L\big)_+^n}{n!} \Big) \end{aligned}$$

Here we have used theorem Eq. (4.23) to replace the box filter by a delta distribution. Relabeling the index in the first term of the sum and using the equality $\binom{n}{i-1} + \binom{n}{i} = \binom{n+1}{i}$ we obtain

$$b_{n+1} = \frac{1}{(2L)^{n+1}} \sum_{i=0}^{n+1} (-1)^i \binom{n+1}{i} \frac{\big(x+(n+1-2i)L\big)_+^n}{n!}, \tag{4.30}$$

in accordance with Eq. (4.28). All filters b_n are normalized since $\int f\,dx = \int g\,dx = 1$ implies $\int f * g\,dx = 1$, and $\int b_1\,dx = 1$. The functions b_n have support $[-nL, nL]$ as can be seen by inspection of Eq. (4.28).

In Table 4.1 some filter kernels of B-spline type are listed. These filters have been rescaled in order to achieve $supp\ h = [-L, L]$. They are still normalized to unity. The table also contains an expression for $f * h$. The convolution has been decomposed in terms of repeated integrals using Eq. (4.27).

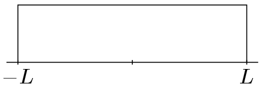
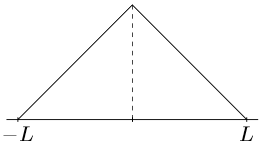
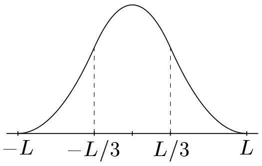
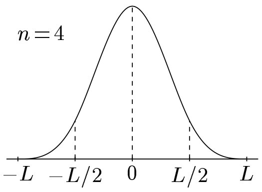
Filter kernel h	Definition	Convolution $f * h$
	$P_1 = \frac{1}{2L}$	$\frac{1}{2L}\big(F(x+L) - F(x-L)\big)$
	$P_1 = \frac{1}{L^2}(x+L)$ $P_2 = \frac{1}{L^2}(L-x)$	$\frac{1}{L^2}\big(F_2(x+L) - 2F_2(x) + F_2(x-L)\big)$
	$P_1 = \frac{27}{16L^3}(x+L)^2$ $P_2 = \frac{9}{8L} - \frac{27}{8L^3}x^2$ $P_3 = \frac{27}{16L^3}(L-x)^2$	$\frac{27}{8L^3}\big(F_3(x+L) - 3F_3(x+L/3)$ $+3F_3(x-L/3) + F_3(x-L)\big)$
	General nth-order B-Spline: $f * h = \left(\frac{n}{2L}\right)^n \sum_{i=0}^{n} (-1)^i \binom{n}{i} F_n\big(x + \frac{n-i}{n}L\big)$	

Table 4.1: Filter kernels of B-spline type. The filter of order $n-1$ can be obtained by convolving the nth order filter with a box filter. The filters in this table have support $[-L, L]$ and are normalized to one.

The construction of B-splines from repeated convolutions has first been described by Curry and Schoenberg in 1947. In modern spline theory B-splines are usually defined in a more general context as divided differences for an arbitrary nondecreasing knot sequence. For more information we refer to deBoor [18].

A nice property of a nth order filter kernel of B-spline type is that all derivatives except of the $n-1$th one are continuous, i.e. the filter is built from exactly $n+1$ truncated power functions of degree $n-1$. Consequently, the evaluation of $f * h$ involves nth order integrals only. For general piecewise polynomial filter kernels usually integrals of mixed orders have to be considered. This makes the computation of $f * h$ more expensive.

4.5 The Fast LIC Algorithm

We are now in a position to formulate an algorithm for fast computation of LIC images. The key idea of this algorithm is to exploit the coherence of pixel intensities in flow direction. More precisely, this means that the LIC intensity for a given point on a field line is *updated* to obtain the intensity of a neighbouring point. Such an intensity update turns out to be much less expensive than evaluating the convolution integral from scratch.

4.5.1 Discretization

We start the construction of the fast LIC algorithm by distributing discrete sample points $\{s_k\}$ on a field line $\boldsymbol{\sigma}(s)$. The sample points should be equidistant in parameter space, i.e.,

$$s_k = k\Delta s, \quad k \in \mathbb{Z}. \tag{4.31}$$

We are looking for a discrete approximation of the repeated integrals $F^{(j)}$ on the discrete mesh $\{s_k\}$. The approximation itself should be constructed only from samples taken at $\{s_k\}$. This is an important requirement, which facilitates the recursive evaluation of repated integrals of increasing order and which allows us to compute $F^{(j)}$ efficiently for many samples. The most simple solution is to choose left-handed Riemann sums, yielding

$$F^{(j)}(s_k) = \int\limits_0^{k\Delta s} F^{(j-1)}(x)\,dx \approx \begin{cases} \Delta s \sum\limits_{i=0}^{k-1} F^{(j-1)}(s_i), & k > 0 \\ 0, & k = 0 \\ \Delta s \sum\limits_{i=k}^{-1} F^{(j-1)}(s_i), & k < 0 \end{cases} \tag{4.32}$$

for $j \geq 1$. For convenience we introduce the abbreviation $F_k^{(j)} \equiv F^{(j)}(s_k)$. Using this notation, the relation of $F^{(j)}$ between neighbouring sample points is given by

$$F_{k+1}^{(j)} = F_k^{(j)} + \Delta s\, F_k^{(j-1)}, \quad j \geq 1. \tag{4.33}$$

On the first sight it might seem that the use of left-handed Riemann sums in Eq. (4.32) is a very crude approximation. However, the relative error of the approximation decreases with $1/N$, where N is the number of sample points. The final convolution integral is typically approximated by about 50-200 samples. It turns out that the corresponding intensity deviations do hardly produce any visual artifacts.

Nevertheless, in principle more accurate integration formulas may be applied, too. Centered Riemann sums are not suitable, because they are constructed from samples located at $(k + 1/2)\Delta s$. Therefore, they prevent recursive evaluation of $F^{(j)}$. However, trapezoid rule is a valid alternative. In this case the integral $F_k^{(j)}$ would be approximated by $|k| + 1$ symmetrically distributed samples, where the first one and the last one – located at the boundaries of the integration interval – are being weighted with a factor 1/2. Instead of Eq. (4.33) the relation of $F^{(j)}$ between neighbouring sample points turns out to be given by a slightly more complex formula, namely

$$F_{k+1}^{(j)} = F_k^{(j)} + \frac{\Delta s}{2}\left(F_k^{(j-1)} + F_{k+1}^{(j-1)}\right), \quad j \geq 1. \tag{4.34}$$

After discrete approximations of the repeated integrals $F^{(j)}$ have been computed, we can immediately express the final LIC intensity I_k at a sample point s_k using Eq. (4.27). In order to avoid any computational overhead we just have to ensure that the knots ξ_i

of the piecewise polynomial filter kernel coincide with sample points s_k. We therefore introduce discrete offsets n_i and require $\xi_i = n_i \Delta s$. The final LIC intensity is then given by

$$I_k = \sum_{\substack{i=1,\dots,l \\ j=1,\dots,m}} c_{ij}\, F^{(j)}_{k+n_i}. \tag{4.35}$$

Recall, that the numbers l and m are usually quite small. For example, in case of a box filter we have $l = 1$ and $m = 2$. The sum (4.35) therefore includes only a few terms.

4.5.2 Intensity Updates

The basic idea of the proposed algorithm is to compute intensities I_k for many sample points s_k on a single field line in one pass. It should be clear that this goal can be easily achieved by using update formulas like Eq. (4.33) or Eq. (4.34). Let us suppose, that the LIC filter kernel is of order m and has an extent of N samples in both directions. In order to obtain the initial intensity I_0, the values of the repeated integrals $F^{(j)}_k$ with $k = -N, ..., N$ and $j = 1, ..., m$ have to be computed and stored in an appropriate array. From these values I_0 can be determined directly using Eq. (4.35).

To obtain additional intensity values I_{k+1}, a straight-forward algorithmic approach would be to compute $F^{(j)}_{k+N}$ for $j = 1, \dots, m$ via Eq. (4.33) or Eq. (4.34) and to append these values to the array of repeated integrals. Then the intensity at the next sample point can again be determined directly using Eq. (4.35). Of course, the same strategy might be applied to step along the field line in negative direction by computing I_{k-1}.

However, the explicit computation of repeated integrals $F^{(j)}_k$ for larger and larger values of k is numerically not favourable, because it may cause overflow problems. Usually, the input image contains by 8-bit integer values in the range of 0 to 255. For the actual computation we make use of 32-bit integer arithmetic. Assuming constant input values $f = 128$, for example the 4-th order integral $F^{(4)}$ would cause an overflow at $k \approx 100$. Since such distances might well occur in practice, the LIC intensities should be evaluated in a numerically stable way.

In order to find a stable evaluation scheme, we rewrite Eq. (4.35) as

$$I_k = \sum_{j=1,\dots,m} I^{\langle j\rangle}_k, \quad \text{with} \quad I^{\langle j\rangle}_k = \sum_{i=1,\dots,l} c_{ij}\, F^{(j)}_{k+n_i}. \tag{4.36}$$

Let us first consider the case of left-handed Riemann sums. We use Eq. (4.33) to obtain an update formula for the components $I^{\langle j\rangle}_k$, namely

$$I^{\langle j\rangle}_{k+1} = \sum_{i=1,\dots,l} c_{ij} \left(F^{(j)}_{k+n_i} + \Delta s F^{(j-1)}_{k+n_i} \right) = I^{\langle j\rangle}_k + \Delta I^{\langle j\rangle}_k, \tag{4.37}$$

where we have introduced *intensity differences*

$$\Delta I_k^{\langle j\rangle} = \Delta s \sum_{i=1,\dots,l} c_{ij} F_{k+n_i}^{(j-1)}. \tag{4.38}$$

While for $j = 1$ the corresponding intensity difference is a linear combination of the original function values $F_{k+n_i}^{(0)} \equiv f_{k+n_i}$, higher-order differences still involve repeated integrals, namely integrals of order $j-1$. These integrals can be eliminated using a nested evaluation scheme. We generalize Eq. (4.38) and define *repeated intensity differences* by

$$\Delta^p I_k^{\langle j\rangle} = (\Delta s)^p \sum_{i=1,\dots,l} c_{ij} F_{k+n_i}^{(j-p)}, \quad p = 0, \dots, j. \tag{4.39}$$

Note, that $\Delta^0 I_k^{\langle j\rangle} \equiv I_k^{\langle j\rangle}$. It is easy to see that the repeated intensity differences obey the relation

$$\Delta^p I_{k+1}^{\langle j\rangle} = \Delta^p I_k^{\langle j\rangle} + \Delta^{p+1} I_k^{\langle j\rangle}. \tag{4.40}$$

We are now able to update the intensity I_k by maintaining a set of $\frac{1}{2}\,m\,(m+1)$ intermediate variables $\Delta^p I_k^{\langle j\rangle}$, which are being updated according to Eq. (4.40). The resulting evaluation scheme can be sumarized in the following way:

$$\begin{array}{lllll}
I_k^{\langle 1\rangle} & \leftarrow \ \Delta I_k^{\langle 1\rangle} & & & \\
I_k^{\langle 2\rangle} & \leftarrow \ \Delta I_k^{\langle 2\rangle} & \leftarrow \ \Delta^2 I_k^{\langle 2\rangle} & & \\
\dots & & & & \\
I_k^{\langle m\rangle} & \leftarrow \ \Delta I_k^{\langle m\rangle} & \leftarrow \ \Delta^2 I_k^{\langle m\rangle} & \dots & \leftarrow \ \Delta^m I_k^{\langle m\rangle} \\
\hline
I_k & & & &
\end{array} \tag{4.41}$$

The rightmost expressions in this schemes represent linear combinations of the original function values f_{k+n_i}. These expressions are added to the second last intensity difference in each row in order to perform the update for $k+1$. Similarly, the other intensity differences are updated by adding the expression to the right. Finally, the resulting LIC intensity is given as the sum of the terms in the left-most column of the table. The advantage of the above evaluation scheme is, that *repeated intensity differences* are being updated instead of *repeated integrals*. In contrast to integrals differences do not get larger as the sample index k increases. Thus, the scheme is robust and not harmed by numerical overflow problems. At the same time, the number of operations is identical for both approaches, since the update formulas Eq. (4.33) and (4.40) are of identical structure.

It should be noted that the evaluation scheme Eq. (4.41) is not restricted to left-handed Riemann sums. In order to use more sophisticated integration rules, merely the update formula Eq. (4.40) has to be modified. For example, trapezoid rule would yield

$$\Delta^p I_{k+1}^{\langle j\rangle} = \Delta^p I_k^{\langle j\rangle} + \frac{1}{2}\left(\Delta^{p+1} I_k^{\langle j\rangle} + \Delta^{p+1} I_{k+1}^{\langle j\rangle}\right). \tag{4.42}$$

In contrast to Eq. (4.40) this expression is symmetric in k and $k+1$. This means, that the update formula in backward direction looks the same as in forward direction.

4.5.3 Special Filter Kernels

We now want to apply the intensity update scheme outlined in the previous section to some simple filter kernels of practical interest. In this way, also the general procedure should become more clear.

Box Filter. The most simple filter kernel for LIC is definitely a box filter. It is of first order and has two knots, i.e., $m = 1$ and $l = 2$. Let us assume the filter width is $N\Delta s$ in both directions. Applying Eq. (4.35) we find that the normalized LIC intensity for a box filter is

$$I_0 = \frac{1}{2N}\,(F_N - F_{-N}) = \frac{1}{2N}\sum_{i=-N}^{N-1} f_i. \tag{4.43}$$

The last expression has been derived by approximating F_N and F_{-N} by means of left-handed Riemann sums. Additional intensities in forward direction can be obtained via

$$I_{k+1} = I_k + \Delta I_k^{\langle 1\rangle} = I_k + \frac{\Delta s}{2N}\,(f_{k+N} - f_{k-N}). \tag{4.44}$$

In contrast, the update formula in backward direction is

$$I_{k-1} = I_k - \Delta I_{k-1}^{\langle 1\rangle} = I_k - \frac{\Delta s}{2N}\,(f_{k+N-1} - f_{k-N-1}). \tag{4.45}$$

The box filter has minimal computational costs. The update essentially involves just two additions. An additional multiplication is necessary to ensure proper normalization. The unsymmetry of the update formulas is caused by the use of left-handed Riemann sums. Trapezoid rule yields symmetric expressions. However, these expressions would contain samples of the input function f at four different locations instead of just two. An alternative is to compute the LIC intensity from $2N + 1$ samples instead of $2N$. In this case the width of the box filter actually is $(N + 1/2)\Delta s$, and we have

$$I_0 = \frac{1}{2N+1}\sum_{i=-N}^{N} f_i, \tag{4.46}$$

which is accompanied by a symmetric update formula

$$I_{k\pm 1} = I_k + \frac{\Delta s}{2N+1}\,(f_{k\pm N\pm 1} - f_{k\mp N\mp 1}). \tag{4.47}$$

In fact, this type of discretization is commonly used for implementing convolutions with symmetric filter kernels. In our introductory paper about the fast LIC algorithm [81] we used this kind of discretization, too.

Triangle Filter. The next simple filter kernel after a box is a triangle or hat. The triangle filter is of second order and has three knots, i.e., $m = 2$ and $l = 3$. Since the filter is continuous, the first-order contributions vanish and we have $I \equiv I^{\langle 2\rangle}$. From Table 4.1

we find that the second order coefficients are $c_{12} = c_{32} = 1/N^2$ and $c_{22} = -2/N^2$. In addition to the intensity itself we need to maintain an additional intensity difference $\Delta I \equiv \Delta I^{\langle 2 \rangle}$. Initially, these values are given by

$$I_0 = \frac{1}{N^2} \left(F_N^{(2)} - 2F_0^{(2)} + F_{-N}^{(2)} \right) \tag{4.48}$$

$$\Delta I_0 = \frac{\Delta s}{N^2} \left(F_N^{(1)} - 2F_0^{(1)} + F_{-N}^{(1)} \right). \tag{4.49}$$

Using left-handed Riemann sums, additional samples in forward direction can be computed via

$$I_{k+1} = I_k + \Delta I_k \tag{4.50}$$

$$\Delta I_{k+1} = \Delta I_k + \frac{\Delta s}{N^2} \left(f_{k+N} - 2f_k + f_{k-N} \right). \tag{4.51}$$

The corresponding update formulae in backward direction are

$$I_{k-1} = I_k - \Delta I_{k-1} \tag{4.52}$$

$$\Delta I_{k-1} = \Delta I_k - \frac{\Delta s}{N^2} \left(f_{k+N-1} - 2f_{k-1} + f_{k-N-1} \right). \tag{4.53}$$

Higher-order Filter Kernels. Higher-order filter kernels can be evaluated in a similar way. Once again we point out, that kernels of B-spline type are particularly nice because they involve terms of fixed order only. This means, that only a single row of the general update scheme in Eq. (4.41) need to be considered. For example, the cubic B-spline filter listed in Table 4.1 requires to update three values $I \equiv I^{\langle 3 \rangle}$, $\Delta I \equiv \Delta I^{\langle 3 \rangle}$, and $\Delta^2 I \equiv \Delta^2 I^{\langle 3 \rangle}$. Using left-handed Riemann sums, the corresponding update formulae in forward direction are given by

$$I_{k+1} = I_k + \Delta I_k, \quad \Delta I_{k+1} = \Delta I_k + \Delta^2 I_k \tag{4.54}$$

$$\Delta^2 I_{k+1} = \Delta^2 I_k + \frac{27\, \Delta s}{8\, N^3} \left(f_{k+N} - 3f_{k+N/3} + f_{k-N/3} - f_{k+N} \right) \tag{4.55}$$

The backward formulae can be derived in an analogous way. Note, that in this case the total number N of samples used to approximate the filter need to be a multiple of three. Otherwise, the samples $f_{k \pm N/3}$ would not be available. This is a general constraint of the technique, getting more dominant if more complex filter kernels are used. However, usually we want to apply simple filters. In addition, quite large numbers of N will be used. Therefore, in practice it is always possible to choose filter lengths which are compatible with the breakpoints of the piecewise polynomial filter kernel.

4.5.4 Hit Count and Accumulation Buffer

In order to produce an actual LIC image, we need to compute LIC intensities in a 2D domain Ω, and not just for individual samples on a single field line. We can solve this problem by distributing *many field lines* homegeneously in the output image Ω. In practice, Ω is subdivided into a number pixels p with non-overlapping domains Ω_p, i.e.,

$$\Omega = \bigcup_p \Omega_p. \tag{4.56}$$

For each pixel p an associated LIC intensity I_p has to be determined. Ideally, this intensity should be computed as an average over the whole pixel area,

$$I_p = \frac{1}{A} \int_{\Omega_p} I(x)\, dA. \tag{4.57}$$

A standard method in computer graphics is to approximate such integrals by a sufficiently high number of point samples in Ω_p. In the simplest case just one sample located in the center of Ω_p would be evaluated. For example, this is done in the LIC algorithm of Cabral and Leedom, c.f. Sect. 4.3.4. The fast convolution method outlined in the previous sections can be used to produce many samples located on common field lines. Although we have only limited control of how the samples are distributed in a particular pixel, a natural approach would be to average all samples falling into a common pixel.

In our implementation we maintain two variables for each pixel p, `hitcount[`p`]` and `accum[`p`]`. We refer to these variables as hitcount and accumulation buffer. Instead of computing only one intensity value from a single field line $\boldsymbol{\sigma}(s)$, N_u intensity updates are performed on that curve in forward and backward direction, yielding additional samples I_k with $|k| = 1, ..., N_s$. These additional samples will be added to `accum[`p_k`]`, where p_k denotes the pixel of the output image the sample falls into. At the same time `hitcount[`p_k`]` is increased by one. Traversing all pixels once, we compute a new field line $\boldsymbol{\sigma}(s)$ whenever the number of hits for a pixel is less than a user-defined minimum `minHit`. At the end, a second pass is necessary to normalize the accumulated intensities according to the total number of hits per pixel. In summary, the fast LIC algorithm can be described by the following pseudocode:

```
for all pixels p
    set hitcount[p]=0 and accum[p]=0
for all pixels p
    if (hitcount[p] < minHit) then
        compute field line starting from center of p
        compute initial intensity I_0 and differences Δ^p I_0^⟨j⟩
        set accum[p] = accum[p] + I_0
        set hitcount[p] = hitcount[p] + 1
        for |k| = 1,...,N_u do
            compute I_k and Δ^p I_k^⟨j⟩ by applying update scheme
            determine pixel p_k containing σ(s_k)
            set accum[p_k] = accum[p_k] + I_k
```

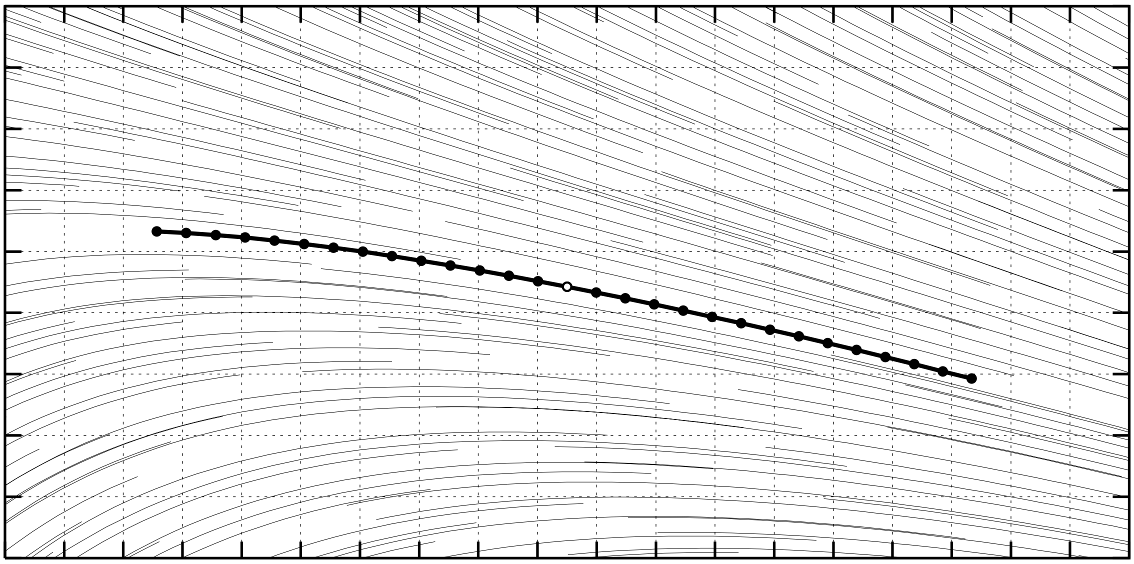


Figure 4.8: The fast LIC algorithm relies on sampling the input texture at evenly spaced locations along a field line $\boldsymbol{\sigma}$. Intensity values I_k are added to the pixels containing the sample location $\boldsymbol{x}_k$. New field lines are computed only for pixels which are not yet covered by a sufficiently high number of samples. Field lines and sample points should be distributed more or less evenly across the whole image.

set $\texttt{hitcount}[p_k] = \texttt{hitcount}[p_k] + 1$
for all pixels p
set $I_p = \texttt{accum}[p]/\texttt{hitcount}[p]$

A typical pattern of field lines and sample points generated by the algorithm is shown in Figure 4.8. In order to minimize the computational costs, field lines should be distributed as evenly as possible throughout the image. In this way the average number of hits per pixel is minimized. There are a number of algorithmic parameters which influence the performance of the algorithm as well as the quality of the resulting images. We shall discuss these issues in detail later on. For the moment we just make the following remarks:

- In contrast to the LIC algorithm of Cabral and Leedom, the computation of field lines is performed without referencing input image or output image. This allows us to utilize any of the numerical integration methods discussed in Chapter 3. An integrator with adaptive step size control and error monitoring ensures accuracy and accelerates field line computation significantly in homogeneous regions.

- As already mentioned in Section 4.3.1, special treatment is necessary when field lines leave the computational domain Ω, or when singularities are encountered. In any case, we may simply continue the path in the current direction in order to provide a full-sized convolution interval. Of course, no intensity updates should be performed for extended path segments.

- In the above pseudocode the number of additional samples N_u on a field line computed by means of intensity updates appears as a user-defined parameter. However, this parameter can be adjusted automatically. In fact, in Section 4.6.2 we show, that the optimal value of this parameter gradually changes as more and more field lines are computed.

- The number of samples within a pixel as well as their relative distribution depends on the order in which the pixels of the output image are processed. Therefore, the traversal scheme can be used to optimize the algorithm. The effect of different traversal schemes will also be discussed in Section 4.6.3.
- The algorithm involves two sampling processes, one for approximating the one-dimensional convolution integral, the other for averaging LIC intensities within a pixel. The choice of parameters like sampling distance, number of samples per pixel, or spatial distribution of samples depends on the kind of input image as well as the desired quality of the output image. Improper choice of these parameters will result in aliasing effects, cf. Section 4.6.4.
- Finally, the resolution of input image and output image, i.e., the sizes of the corresponding pixels, are not required to be equal. The ability to choose different resolutions for both images facilitates applications like continuous zooming and magnification of details. This will be discussed in Secton 4.6.5.

Depending on the vector field being visualized, the type of filter kernel being used, as well as the length of the filter kernel, the new method turns out to be about an order of magnitude faster than the algorithm proposed by Cabral and Leedom, cf. [81]. Performance gains increase as larger filter lengths are chosen. For example, many LIC images shown in this work were computed using (one-sided) filter lengths of 40, 80, or even 120 and more pixels. Computing such images essentially is not feasible using the original method while it takes only a few seconds using the new approach.

4.6 Algorithmic Details

In the previous section we described how to evaluate LIC integrals for piecewise polynomial filter kernels efficiently for many points on a common field line. We outlined how this idea could be used to generate 2D LIC images. In this section we want to work out the proposed fast LIC algorithm and discuss how the performance of the algorithm can be optimized.

4.6.1 Sampling Distance and Input Image

The fast LIC algorithm approximates the continuous one-dimensional convolution integral Eq. (4.9) by a weighted sum of evenly spaced samples $T(s_k)$. In this section we want to investigate how to choose the sampling distance Δs in an optimal way. It should be noted that irrespectively of the actual value of Δs the convolution sum is well-defined and might be evaluated at any point within a pixel. However, if the sampling distance is too large the continuous LIC integral will not be approximated well and artifacts may occur. On the other hand, if the sampling distance is too small unnecessary work will be done and the performance of the algorithms drops off.

In order to simplify the notation we rewrite the continuous LIC integral as

$$I(s) = \int_{-\infty}^{\infty} T(s-u)\, g(u)\, du. \tag{4.58}$$

Here, T represents the input image along a field line and g is a continuously defined filter kernel. We can express the discrete approximation of this integral as computed in the fast LIC algorithm by

$$\tilde{I}(s) = \int_{-\infty}^{\infty} T(s-u)\, g(u)\, \mathit{III}(u)\, du, \quad \text{with} \quad \mathit{III}(s) = \sum_i \delta(s - i\Delta s). \tag{4.59}$$

Notice, that the continuous filter kernel g has been modulated by a comb or shah function III, i.e., by a set of evenly spaced δ-impulses. For $\tilde{I}$ to be a close to I the distance Δs should not exceed the characteristic feature size of the input image T. For random noise images this is about the width of one pixel, which we will denote by Δ. If the sampling distance Δs doesn't match this criterium, high frequency components will be observed. Intuitively, it is clear that intensities $\tilde{I}(s)$ and $\tilde{I}(s+\Delta s)$ are strongly correlated because they are computed from almost the same samples. However, the value of $\tilde{I}(s+\Delta s/2)$ may be completely different if T changes significantly over half a sampling width. Therefore, intensity correlation along a field line will start to oscillate if the sampling distance is too large.

Fourier analysis provides an excellent tool to understand the occurence of high frequencies components in detail. First of all, let us recall that the analogue of convolution in spatial domain is modulation (i.e., multiplication) in Fourier space, and vice versa. Therefore, from Eq. (4.59) we obtain the following Fourier transform pair,

$$\tilde{I}(x) = T(x) * (g(x)\, \mathit{III}(x)) \quad \Longleftrightarrow \quad \tilde{I}(k) = T(k)\, (G(k) * \mathit{III}(k)). \tag{4.60}$$

The Fourier transform of the comb function $\mathit{III}(x)$ is another comb function $\mathit{III}(k)$ with reciprocal spacing $2\pi/\Delta s$. Therefore, discrete sampling causes the function $G(k)$ to be replicated at higher frequencies in Fourier space. Corresponding frequency components of the input image are copied directly into the output image.

At this point we note that the Fourier transform of a box is a sinc function. Therefore, we have $|G| = |\sin(k)/k|$. The graph of this function is depicted in Figure 4.9 a). Note, that $|G|$ vanishes exactly at integer multiples of $2\pi/\Delta$, where Δ denotes the width of the box. The bigger the spatial extent of a function g the more localized it is in Fourier space. For example, the triangle filter in Figure 4.9 b) has an extent of 20Δ. Its power spectrum is almost a delta function.

Now, let us consider particular choices of input images. For LIC we are especially interested in noise images. In the most simple case for each pixel a random value is generated, yielding a piecewise constant input image. Extracting vertical or horizontal lines

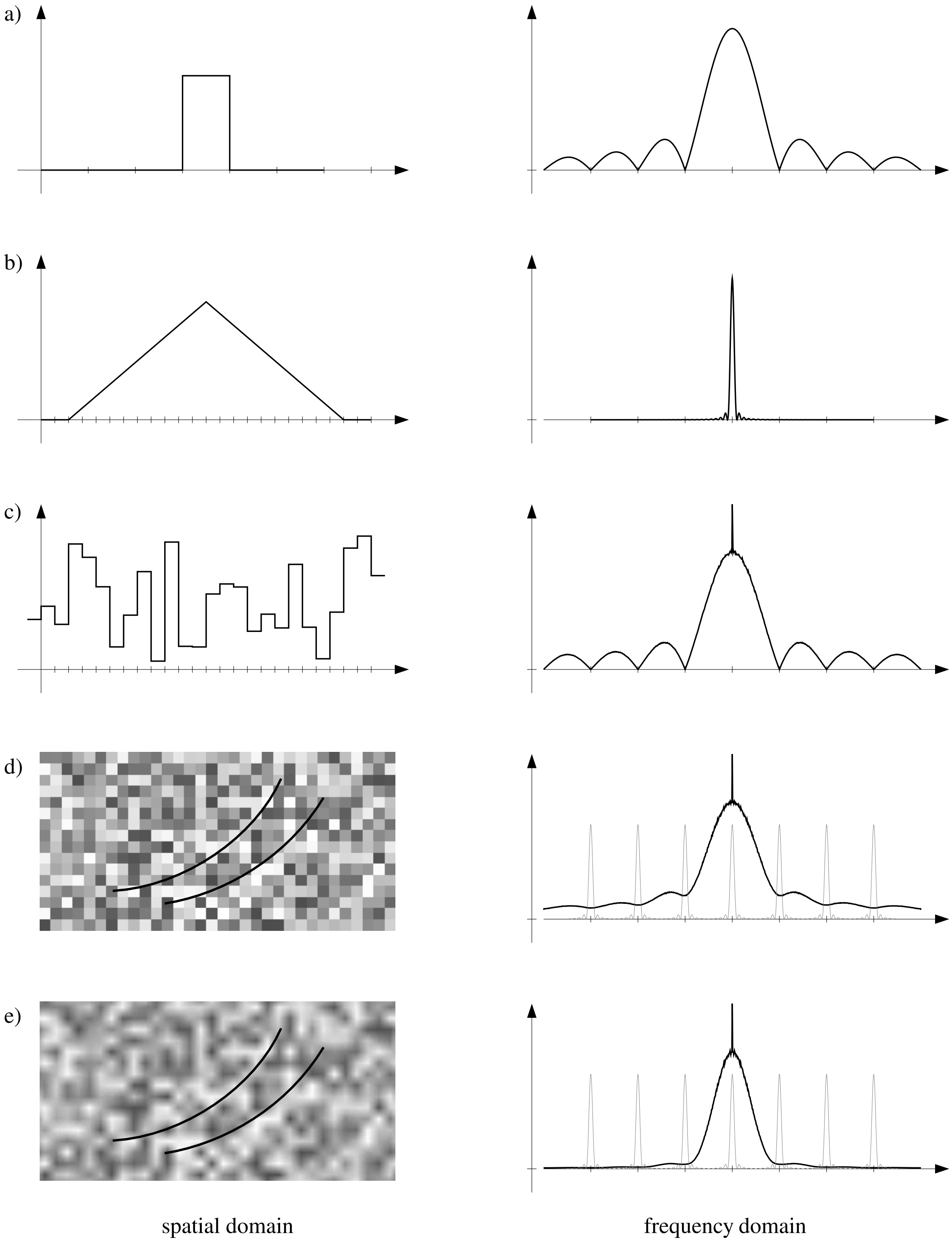


Figure 4.9: The plots on the right are the power spectra of the functions on the left. The Fourier transform of a box is a sinc function (a). The wider a function in physical space, the more localized it is in Fourier space (b). The average power spectrum of a regular noise function again is a sinc (c). In contrast, the piecewise constant regions of a 2D noise image covered by a field line are of variable width, thus producing a spectrum as shown in (d). If the input image is interpolated bilinearly high-frequency components of the power spectrum are suppressed (e).

of pixels we obtain a linear function $T(x)$ exactly made up of evenly spaced and randomly weighted boxes. The average power spectrum of such a function is the same as for a single box, except for an additional delta peak at the origin, c.f. Figure 4.9 c). This peak is caused by a non-vanishing mean value of $T(x)$. It is clear that such a regular input function should be sampled using an increment of exactly Δ. However, in general field lines of an arbitrary vector field will cross pixels of the input image at arbitrary angles. Thus, the regular structure of the 1D noise function $T(x)$ is destroyed. The power spectrum $T(k)$ doesn't vanish at multiples of $2\pi/\Delta$ anymore. Instead, a spectrum like in Figure 4.9 d) is obtained. In the same figure we superimposed the spectrum of $G(k) * III(k)$, with G being the triangle filter shown in Figure 4.9 b). As stated in Eq. (4.60) the Fourier transform of the convolved signal is obtained by modulating $T(k)$ and $G(k) * III(k)$. In order to get the same result as in the continuous case there should be no contribution from higher-order aliases of $G(k)$, i.e., the input function should be band-limited. However, in case of piecewise constant noise the best we can do is to chose a sampling distance matchings the minima of the power spectrum. From 4.9 d) we find that this is still Δ, the width of a single pixel. Note also, that choosing a sampling distance of $\Delta/2$ doesn't improve things much, since high-frequency components of the power spectrum are stil quite large.

A simple way to eliminate high-frequency components of the power spectrum is to interpolate the input image bilinearly. In this case we obtain an average spectrum like in Figure 4.9 e). Again, we have a minima at integer multiples of $2\pi/\Delta$. Therefore, a sampling distance of Δ again is favourable. Note however, that bilinear interpolation also implies that the final LIC image has less contrast. Therefore, we can't directly compare both types of input noise. This will be done in Section 4.6.4. From our discussion here it merely follows that a sampling distance $\Delta s = \Delta$ provides a good choice for approximating the one-dimensional convolution integral.

4.6.2 Adjusting the Number of Intensity Updates

Suppose, the extent of the LIC filter kernel is $N\Delta s$ in both directions. Then, in order to evaluate the LIC intensity for the first point on a field line $2N$ sample points on that line have to be computed. After this start-up phase the update scheme Eq. (4.41) can be applied. Then ideally every additional point on a field line yields a new intensity value. Therefore, the number N_u of intensity updates should be taken as large as possible. However, there are some limiting factors. First of all, field lines may leave the computational domain Ω, or they may end up in a singularity. In this case no more intensity values can be computed. Another option is, that field lines run into regions where all pixels already have been touched. For example, in vector fields containing vortices, a field line may turn around forever without hitting a single new pixel. In this case, it also doesn't make sense to perform more intensity updates on that line. In consequence an optimal value for the number of intensity updates N_u exists. This value depends on the vector field characteristics *and* the actual pixel coverage.

In the following we will outline a simple adaptive strategy to determine an estimate for the optimal value of N_u. In our discussion let us assume that every pixel must be touched at least once. Furthermore, we assume that the computational costs per field line are directly proportional to the total number of samples on that line. In particular, cache effects and non-linear costs for the numerical integrator are ignored. The total number of samples per field line consists of two terms, the number of samples required to compute the initial convolution integral and the number of intensity updates. Therefore the costs per field line are

$$C_{\text{field line}}(N_s) = \alpha\,(N + N_u)\,, \tag{4.61}$$

with a constant factor α. Computing n additional samples will causes a certain number $P(n)$ of previously untouched pixels to be hit. It is obvious that the function $P(n)$ is monotonously increasing. However, in general it will not be proportional to n. Instead, the graph of $P(n)$ will become flatter as more samples are generated. Since field lines are started always on untouched pixels, at least one pixel is hit, i.e., $P(0) = 1$. For a field line with N_u additional samples the *effective costs per pixel* depend on the number of previously untouched pixels being hit. Considering every additional hit on a pixel as being wasted, these costs can be expressed by

$$C_{\text{pixel}}(N_u) = \frac{\text{costs per field line}}{\text{number of previously untouched pixels}} = \frac{\alpha\,(N + N_u)}{P(N_u)}\,. \tag{4.62}$$

In general, this function will have a minimum indicating the optimal value of N_u. In order to locate this minimum we measure $P(n)$ for a certain number of field lines – often enough to get reliable statistics. Then Eq. (4.62) is evaluated and a new value N_u is determined. $P(n)$ is measured by keeping track of the number of new hits obtained per intensity update at a particular distance. Normalization against the number of attempted updates is necessary, because field lines may be terminated prematurely. As already stated, this happens whenever the domain Ω is left or a singularity is encountered.

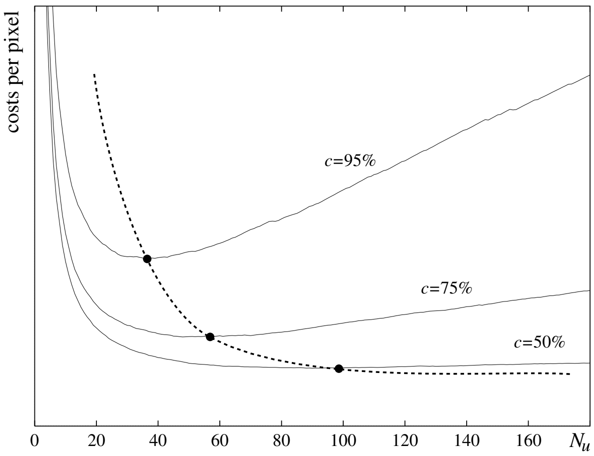


Figure 4.10: Effective costs per pixel versus number of intensity updates N_u on a field line. The three curves correspond to different degrees of pixel coverage.

Figure 4.10 depicts the effective costs per pixel for three different degrees of pixel coverage on a sample vector field. As expected, $C_{\text{pixel}}(N_u)$ increases as more pixels are covered. In other words, the probability to find a previously untouched pixel gets smaller. At the same time the optimal field line length becomes shorter. Comparing the fast LIC algorithm with and without adaptive sampling control gave the following results for a 512×512 image:

N_u	Number of field lines	Number of hits N_{hit}	Total costs
150 (fixed)	5200	1,300,000 (100%)	100%
200…10 (adaptive)	6670	901,000 (69.3%)	80.7%

The value N_{hit} listed in the third column of the table denotes the total number of intensity values computed by the algorithm. Taking into account the size of the image, these numbers correspond to 5.0 and 3.5 hits per pixel, respectively. Note, that for the adaptive strategy the total number of field lines has increased due to their reduced average length. Nevertheless, the total costs estimated by $C_{\text{total}} \sim N_{\text{hit}} + N \times N_{\text{field lines}}$ are only about 80% of the costs of the fixed length strategy. This is, because the number of hits has been reduced drastically.

4.6.3 Seed Point Selection Strategies

In this section we want to discuss different ways of traversing the output image. Obviously, the order in which the pixels of the output image are processed determines what seed points are chosen for field line computation. Our objective is to achieve a very uniform distribution of hits per pixel. Given a user-defined minimal number of hits, this helps to improve the performance of the algorithm, since superfluous hits are avoided. Moreover, a uniform distribution of hits as well as a gradual increase of pixel coverage will be of advantage for the adaptive sampling strategy discussed in the previous section.

Let us first discuss pixel traversal in scanline order. Obviously, if two field lines are started from neighbouring locations, then they are likely to hit the same pixels. In general, this is not desireable. Instead, field lines should be distributed more homogenously throughout the image. For maximal performance, all pixels of the output image should be covered with a minimal number of lines. Unfortunately, the optimal distribution of field lines for a particular vector field is very difficult to compute.

Instead, compared to scanline order, better results can be obtained by a simple heuristic approach. The image is subdivided into small rectangular blocks of $n \times m$ pixels. In a first pass a first pixel out of each block is tested, in a second pass a second one, and so on. This approach has the advantage that the algorithm quickly proceeds into regions which have not yet been covered with field lines at all. Later on, only the gaps between the pixels have to be filled. As a consequence, pixel coverage increases more gradually.

The idea of the blocking approach can be exploited even further with the help of so-called Sobol quasi-random sequences [70, 79]. On the first sight such a sequence is just a permutation $\mathcal{P} : i \leftrightarrow x_i$ with $i, x_i \in \{1, ..., n\}$, so all integers up to a given size appear exactly once. However, the order in which these numbers are arranged is quasi-random like. In contrast to a true random set the discrepancy of the samples is reduced, i.e. clustering is avoided. Sobol quasi-random sequences can be generated by number theoretic methods. In its original form the length of a Sobol sequence is a power of two. However, sequences of arbitrary length can be obtained simply by discarding numbers. Given two such sequences $\mathcal{P}_x$ and $\mathcal{P}_y$ of equal length $n = m$, all pixels of a square image

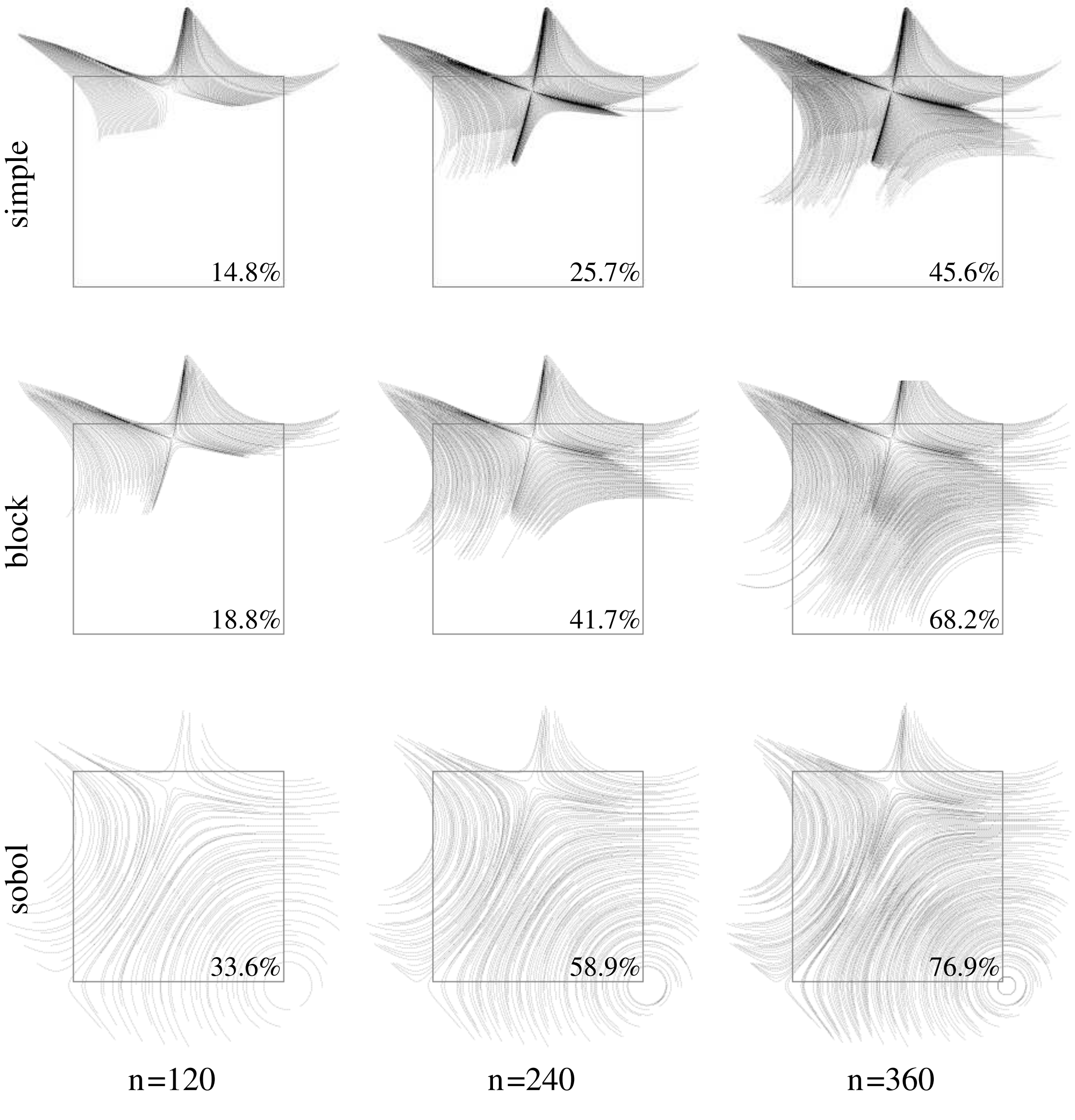


Figure 4.11: Pixel coverage for different seed point selection strategies. Grey levels encode number of hits per pixel. For given number of field lines, the Sobol-based strategy yields highest pixel coverage. Likewise, it produces the smallest number of hits per pixel for a complete LIC image.

can be accessed in quasi-random order using the following index scheme

$$\begin{array}{llll} (x_1, y_1)\,, & (x_2, y_2)\,, & \dots, & (x_n, y_m), \\ (x_1, y_2)\,, & (x_2, y_3)\,, & \dots, & (x_n, y_1), \\ \dots, & & & \\ (x_1, y_m)\,, & (x_2, y_1)\,, & \dots, & (x_n, y_{m-1}) \end{array} \tag{4.63}$$

For non-square images with $n < m$ a similar numbering can be obtained by utilizing an extended sequence $\mathcal{P}'_x$ with $x'_i = x_{i \bmod n}$, $i \in \{1, ..., m\}$. If $n > m$ then $\mathcal{P}_y$ is extended instead of $\mathcal{P}_x$. For better performance both sequences can be tabulated.

In Figure 4.11 pixel coverages are shown that are obtained after 120, 240 and 360 field lines have been computed according to one of the three strategies. Grey levels encode the number of hits per pixel. The Sobol technique obviously achieves higher pixel coverage for a given number of field lines. The total number of samples is usually about 10-20% less compared to the block iteration method. On different computer architectures we obtained the following results for computing a test image of size 512^2 pixels:

	SGI Indigo^2 R4400 150 MHz		SGI Indy R4600PC 100 MHz		Cray T3D (1 processor)	
	work	computing time	work	computing time	work	computing time
block	100%	4.83 sec	100%	7.30 sec	100%	11.42 sec
sobol	88%	5.59 sec (115%)	88%	6.95 sec (95%)	88%	10.87 sec (95%)

This shows that the Sobol scheme only pays on machines without second level cache (here: Cray T3D, SGI Indy R4600PC). Accessing large blocks of memory in quasi-random order on second level cache machines apparently causes too many cache misses, which may completely outweigh the benefits of the selection strategy.

4.6.4 Anti-Aliasing

The one-dimensional nature of line integral convolution causes high-frequency components of the input image in directions perpendicular to the flow to be retained without attenuation in the output image. On the one hand, this is the reason for the high spatial resolution of LIC. In fact, this resolution is only limited by the size of a single pixel. On the other hand, severe aliasing problems may occur if high-frequency input is processed and sampling parameters are not chosen correctly. In this section we want to discuss why these problems occur and how they can be avoided.

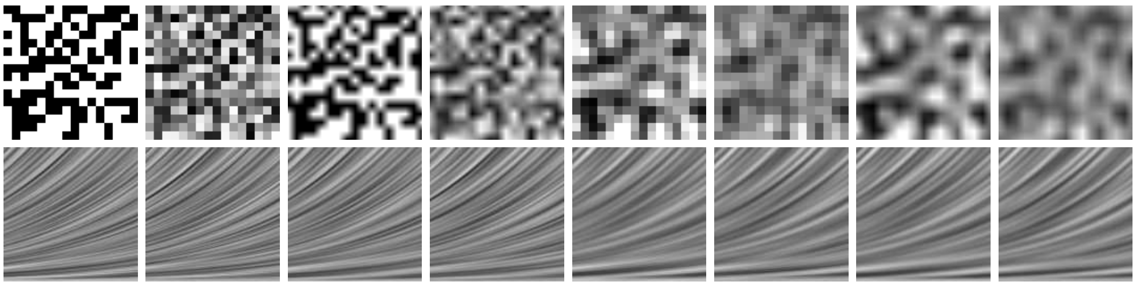
It has already been pointed out that fast LIC actually involves two sampling processes. First, the continuous convolution integral Eq. (4.9) is approximated by sampling the input image T at a number of evenly-spaced locations. This sampling process has already been discussed in Section 4.6.1. Second, the final LIC intensity of a pixel is determined by computing an area average, i.e.,

$$I_p = \frac{1}{A} \int_{\Omega_p} I(\boldsymbol{x})\, dA \approx \frac{1}{n} \sum_{\boldsymbol{x}_i \in \Omega_p} I(\boldsymbol{x}_i). \tag{4.64}$$

Although the function $I(\boldsymbol{x})$ is constructed from a finite set of point samples, it can be evaluated at any point inside a pixel. Thus it represents a continuously defined input signal. Often, this input signal contains frequency components larger than the Nyquist rate $\nu = 1/2\Delta$ defined by the resolution of the output image. According to sampling theory these high-frequency components are transformed into the low-frequency range, a phenomenon called aliasing [65]. An observer will not be able to visually reconstruct the true LIC signal from a sampled output image. Instead, jaggies and other artifacts occur. Often, the resulting LIC images tend to look somewhat noisy. To avoid aliasing the LIC signal has to be low-pass filtered. Computing pixel averages like in Eq. (4.64) is a particularly simple way of low-pass filtering. This approach, also called super-sampling, is a common strategy in computer graphics.

An interesting point of fast LIC is that the samples produced by the algorithm are more or less irregularly distributed in a pixel. Often a certain kind of randomness or jittering of the sampling positions is quite favourable. This so-called stochastic sampling breaks up aliasing by replacing distinct low-frequency aliases by multi-frequency noise, which turns out to be much less disturbing to our visual system [14].

Now the question is, how many samples per pixel are necessary in order to approximate Eq. (4.64) accurately? To answer this question we measured average standard deviation of I_p *per pixel* for various kinds of input noise. In particular, we compared binary and grey noise, both with constant and bilinear interpolation. We also investigated bandlimited noise with a cutoff frequency of $\nu = 1/4\Delta$, compare Figure 4.12. Of course, to get a reliable estimate of intensity deviation quite a large number of homogeneously distributed samples per pixel is necessary. Therefore we modified the Sobol-based seed point selection strategy outlined in Section 4.6.3 and applied it to a set of 3×3 subpixels. In this way we obtained average numbers of well over 20 samples per pixel. The estimated intensity deviations σ_p obtained from this experiment are listed in column two of Table 4.2. Of course, the values σ_p depend on the variance of the input images. The biggest deviation occurs for binary noise with constant interpolation, while the smallest value belongs to bandlimited grey noise with bilinear interpolation. In order to compare these numbers we normalized them such that contrast of the final LIC image was the same. Note, that after



(1) (2) (3) (4) (5) (6) (7) (8)

Figure 4.12: Input noise for LIC. Pixels may either be black and white or may take values between 0 and 255 (grey noise). Moreover, point sampling may be used (1,2,5,6) or images might be interpolated bilinearly (3,4,6,8). Images 5-8 are bandlimited with a cutoff frequency of $1/4\Delta$. In 5 and 7 the original signal was black and white, while in 6 and 8 grey noise has been used.

Type of input image	contrast of LIC image σ_0	unnormalized pixel average σ_p	normalized pixel average $15\,\sigma_p/\sigma_0$	required number of samples ($\gamma=0.9$) $d=\pm1$	$d=\pm5$
1) constant binary	16.76	11.47	10.26	45.7	12.9
2) constant grey	9.59	6.62	10.35	46.4	13.1
3) bilinear binary	14.99	5.91	5.91	16.5	5.6
4) bilinear grey	8.59	3.41	5.95	16.7	5.7
5) constant binary bandlimited	12.00	3.40	4.25	9.5	3.9
6) constant grey bandlimited	6.98	1.99	4.27	9.5	3.9
7) bilinear binary bandlimited	11.58	2.53	3.28	6.5	3.2
8) bilinear grey bandlimited	6.73	1.53	3.40	6.8	3.3

Table 4.2: The first column contains measured intensity deviations σ_0 in a LIC image for different types of input noise (triangle filter, $L = 30$). In the second column the average deviations per pixel are listed. After normalization with respect to σ_0 these values determine how many samples per pixel are required in order to estimate the true pixel intensity with a tolerance of d at a confidence level of γ.

normalization binary and grey noise essentially have equal properties.

The estimates of normalized standard intensity deviation allow us to determine the number of samples n required to determine pixel intensities I_p with certain accuracy. Assuming that pixel intensities are Gaussian distributed, we have to consider Student's t-distribution $F_{n-1}(c)$ [62]. More precisely, for given tolerance d and confidence value γ, the number of samples n has to be determined from

$$d = c\,\frac{\sigma}{\sqrt{n}}, \quad \text{with } c, \text{ so that } F_{n-1}(c) = \frac{1+\gamma}{2}. \tag{4.65}$$

This equation cannot be solved for n in a simple way. Therefore we computed solutions numerically by means of a bracketing approach. The last two columns of Table 4.2 contain the number of samples required to determine I_p with tolerances $d = \pm1$ and $d = \pm5$ at a confidence level of $\gamma = 0.9$, respectively. Here we chose I_p to be in the range 0...255. Note, that for $d = \pm1$ the required number of samples is much higher than the average number of hits per pixel obtained by applying the seed point selection strategies presented in Section 4.6.3. This means that LIC images computed using these strategies exhibit relatively large intensity errors. Fortunately, intensity errors of neighbouring pixels usually are correlated. In this case we may still obtain visually acceptable results. However, in order to compute high-quality LIC images without any aliasing effects at all, either much more samples have to be computed or bandlimited input images have to be used. Of course, the use of bandlimited noise impairs spatial resolution of the LIC texture. It depends on the application if this is acceptable or not. In practice we usually didn't take bandlimited noise but merely used bilinear interpolation. This seemed to be a good compromise between computational costs, image quality, and spatial resolution.

4.6.5 Resolution Independence and Continuous Zooms

A key feature of the proposed LIC algorithm is that field line integration is completely separated from the actual convolution operation. In contrast to the algorithm of Cabral and Leedom, this allows us to process arbitrary vector fields, for example fields which are defined in a procedural way. In particular, if the vector field is defined on a regular uniform grid the resolution of this grid may be chosen completely independent from the resolution of input image and output image.

Up to now, we silently assumed that input image and output image had equal sizes. However, this need not necessarily be the case because the input image is interpolated and sampled at non-integer locations anyway. In the following we want to discuss the effects of changing size and resolution of both images relative to each other. We point out possible applications and consider, how the sampling strategies must be modified in the particular cases.

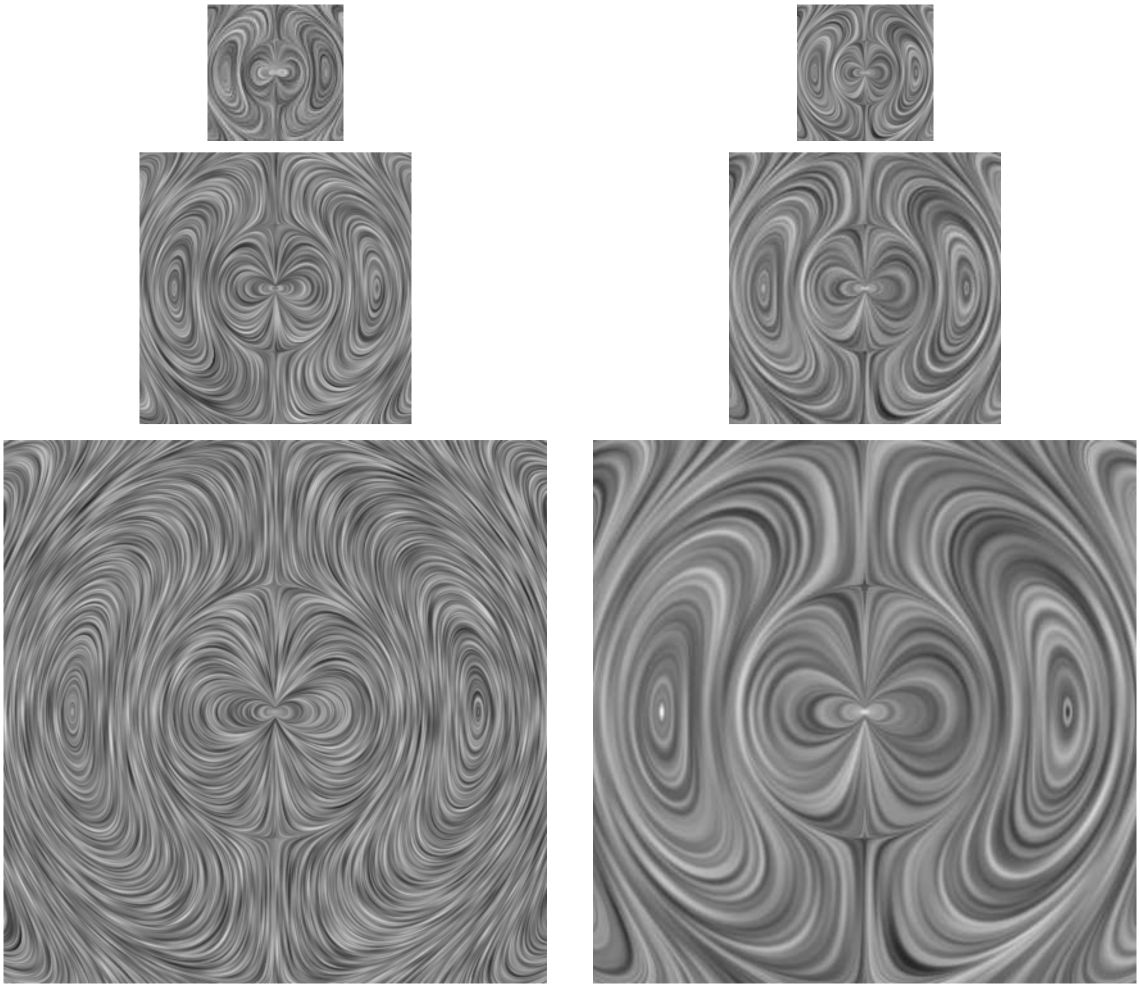
Large LIC Images, Fine Details. As already mentioned, the standard case in line integral convolution is that input image and output image are of equal size. Taking a larger input image will produce a correspondingly larger output image. Since the feature size of a random noise image is related to the size of a single pixel, more details will be contained in larger LIC images. This is illustrated in Figure 4.13 a), where the same vector field was visualized using LIC images of increasing size. Filter length remained constant in terms of pixel width, but gets smaller compared to the total size of the images.

Large LIC Images, Constant Detail. In order to produce exactly the same LIC image at higher resolutions, first of all LIC filter length has to be increased relative to the size of a pixel of the output image. One might be tempted to zoom up the input image in order to reflect the desired size of the output image. However, explicite resampling is not necessary, since input images of arbitrary resolution can be processed in fast LIC. This is possible, since the input image is never accessed on a per pixel basis. Instead, a suitable interpolation method (e.g. constant or bilinear interpolation) is used to evaluate the input image at arbitrary inter-pixel locations. The effect of enlarging the size of the output image relative to the size of the input image is illustrated in Figure 4.13 b).

Low-resolution input images essentially provide low-frequency signals. Therefore, such images might be used in favour of explicitely blurred or filtered high-resolution images. The use of low-frequency input images is important, if the resulting LIC textures are to be processed by some lower bandwidth device like a video recorder or certain image compression algorithms.

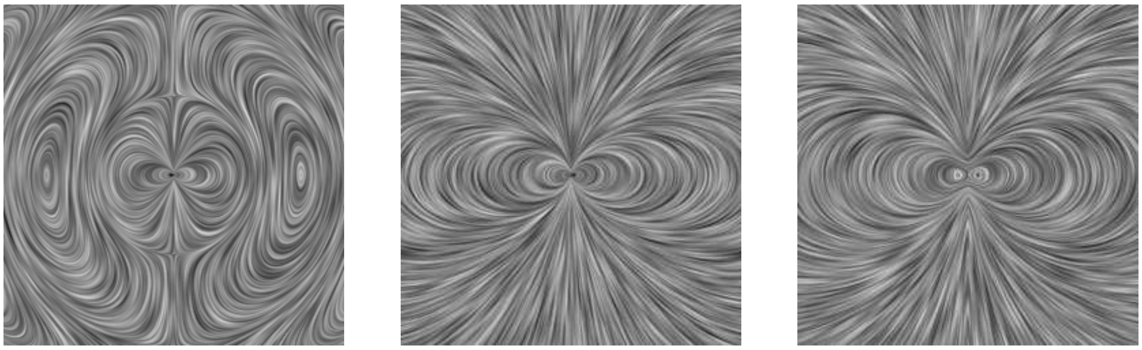
For optimal performance different sampling widths should be used when computing LIC intensities along a single field line. Initially, when computing intensity of the seed point of a field line, we may take large steps, just sufficient to detect all relevant structures of the input image. According to the discussion in Section 4.6.1 this is usually about the size of a pixel of the input image. Afterwards, when producing additional samples on a field line by means of intensity updates we choose a smaller sampling distance equal to the size of a pixel of the output image. This ensures that many output image pixels are touched and that no intensity information is discarded.

Magnification of Detail Structures. Details of a vector field can be easily visualized by applying the LIC algorithm to subregions of the field only. Obviously, if the same filter length and the same input image is used, then the image of the magnified part of the field has the same characteristics as the image of the orginal field. In particular, feature size

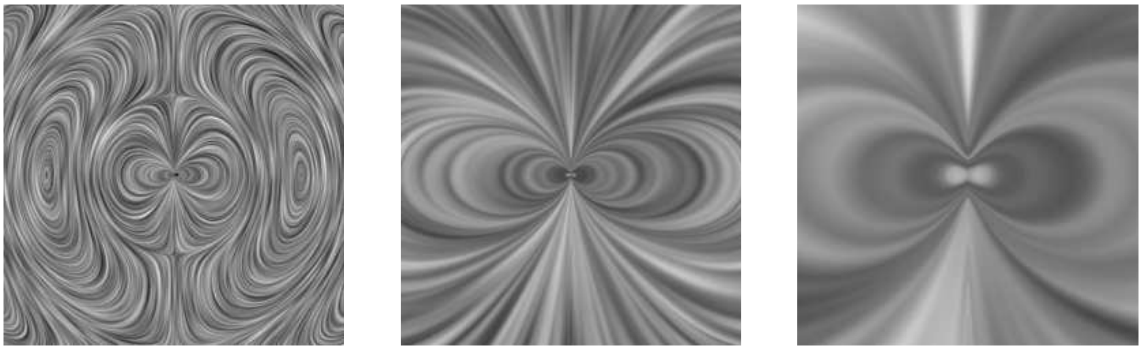


a) b)

Figure 4.13: LIC images of different size. In a) sizes of input and output image were changed simultaneously (120^2, 240^2, and 480^2 pixels). Filter length remained constant in terms of pixel widths. In b) only the output image was enlarged while the input image remained the same. In this way high-resolution copies of exactly the same LIC image are obtained. Filter length increases relative to the size of an output pixel.



a) Detail views built from similar input images.



b) Detail views with scaled filter length and input image.

Figure 4.14: Visualization of a vector field at different scales. In a) only the vector field itself was zoomed, while input image and filter length remained constant. In b) input image and filter length were scaled together with the vector field. The different feature sizes of the LIC texture clearly reveal the different magnification factors.

and granularity of the LIC texture remain the same, as is illustrated in Figure 4.14 a). In this example details of a vector field were visualized at three different resolutions. A better way to reflect the relative scale of the view is to adjust filter length and input image according to the current magnification factor. Looking at Figure 4.14 b) we immediately understand that the images encode the vector field at different scales. In this particular example the zoom factors in a linear dimension were 1:5:25.

4.7 Evaluation of the Algorithm

After the technical aspects of computing LIC images have been discussed we now want to analyse some properties of line integral convolution in general and of the fast LIC algorithm in particular. We will first look at some statistical properties of LIC images, like distribution of intensity values or correlation of these values along field lines. We will make use of these results to compare the effect of using different filter kernels qualitatively.

4.7.1 Global Intensity Distribution

In Section 4.3.3 we already mentioned that the contrast of a LIC image decreases as larger filter kernels are used. To describe this phenomenon more precisely we note that the LIC intensity of a pixel is computed as a weighted sum of essentially *independent* samples of the input image. In particular, the assumption of statistical independence is well fulfilled for random noise input images. Assuming further that all samples of the input image have constant mean value μ_0 and constant variance σ_0^2, mean value and variance of the resulting LIC image are given by

$$\mu = \sum_i g_i \mu_0, \quad \text{and} \quad \sigma^2 = \sum_i g_i^2 \sigma_0. \tag{4.66}$$

The weights g_i are obtained by evaluating the LIC filter kernel $g(x)$ at discrete locations. For our further considerations it is convenient to rewrite these equations in continuous form. Then the sums have to replaced by integrals and we have

$$\mu = \mu_0 \int_{-\infty}^{\infty} g(x)\,dx \quad \text{and} \quad \sigma^2 = \sigma_0^2 \int_{-\infty}^{\infty} g^2(x)\,dx. \tag{4.67}$$

Since the filter kernel g is normalized to unity the average intensity of a LIC image is equal to the mean value μ_0 of the input image. However, variance σ^2 will depend on the particular filter being used. For the special case of a scaled filter $a^{-1}g(ax)$, note that variance changes as follows:

$$\sigma_a^2 = \frac{\sigma_0^2}{a^2} \int_{-\infty}^{\infty} g^2(ax)\,dx = \frac{1}{a}\sigma^2. \tag{4.68}$$

This equation means that the variance of a LIC image decreases as the extent of the filter kernel is enlarged, i.e., as more samples are being averaged. Since variance just controls the global contrast of an image it is clear that longer filter kernels produce flatter LIC images. We already mentioned this phenomenon in Section 4.3.3. Moreover, the statistical independence of the input samples also allows us to apply the central limit theorem of statistics, stating that regardless of the exact distribution the input samples the distribution of the final LIC intensities approaches a Gaussian distribution with mean value μ and variance σ^2, i.e.,

$$p(I) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(I-\mu)^2}{2\sigma^2}}. \tag{4.69}$$

Often LIC images are generated using different filter kernels or different input noise. In these cases, in order to produce images of equal contrast intensities simply have to be rescaled, so that the corresponding intensity histograms have equal width.

4.7.2 Intensity Correlation Along a Field Line

Beside the global intensity distribution of a LIC image another interesting function is the correlation of intensity values along a field line. The correlation of two functions f and h

is closely related to convolution itself. It is defined by

$$corr(f, g) = \int_{-\infty}^{\infty} f(x + y)\, g(y)\, dy. \tag{4.70}$$

Notice, that if either f or g is symmetric, i.e., $f(-x) = f(x)$, then correlation and convolution are identical. The auto-correlation of a function is defined as the correlation with itself. In a sense, auto-correlation measures how much function values separated by given distance are related to each other. For statistical data analysis the functions f and g are commonly normalized to zero mean value. The corresponding function is then called covariance or auto-covariance, if f and g are identical, i.e.,

$$cov(f, g) = \int_{-\infty}^{\infty} (f(x + y) - \langle f \rangle)\, (g(y) - \langle y \rangle)\, dy. \tag{4.71}$$

For zero shift x auto-covariance is equal to variance σ^2. If function values separated by a distance x are completely uncorrelated auto-covariance drops to zero.

The Wiener-Khinchin theorem states that the Fourier transform of the auto-covariance of a signal is given by the power-spectrum of the signal itself,

$$cov(f, f) \Longleftrightarrow |F(k)|^2. \tag{4.72}$$

In case of LIC the signal is the convolution of an input texture T and a filter kernel g, i.e., $I = T * g$. By applying the convolution theorem, we therefore obtain the following Fourier transform pair:

$$cov(I, I) \Longleftrightarrow |T(k)|^2\, |G(k)|^2. \tag{4.73}$$

In the special case of truely random noise input we have $|T(k)|^2 = 1$, and intensity auto-variance is just equal to the *auto-correlation of the filter kernel*. For symmetric filter kernels, correlation can be replaced by convolution, yielding

$$cov(I, I) = g * g. \tag{4.74}$$

Remember, that filter kernels of B-spline type are given by repeated convolution of a box filter with itself. Therefore, in this case auto-covariance will also be of B-spline type. In particular, for a box filter we expect $cov(I, I)$ to be of triangle shape, for a triangle filter auto-covariance will be a third-order B-spline, and so on. We will take a closer look at auto-covariance of different filter kernels when discussing how to animate LIC textures, see Figure 4.17.

4.7.3 Effective Filter Length

For a meaningful comparison between different filter kernels some care has to be taken about the filter lengths. We showed that larger filter lengths yield larger feature sizes and less contrast in the resulting LIC images. Similarly, when comparing e.g. a box filter and a triangle filter of equal length, it is clear that the triangle filter produces a smaller feature size and a higher contrast. The reason is, that pixels at the boundaries get less and less weighted. Therefore the *effective* filter length is smaller for the triangle filter.

In a comparative study we want to choose equal effective filter lengths. We determine these filter lengths such that the variance of the resulting intensity distribution is equal to the one of a corresponding box filter. To actually compute theses variances we again make use of the simplifying assumption that all intensities being averaged in the convolution integral are statistically independent. Then we may apply Eq. (4.67) which states that variance using a filter kernel g essentially depends on the value of the integral $\int g^2(x)\,dx$. By taking the ratio of the variance of a higher-order filter and a box filter we get the following results for the filter kernels listed in Table 4.1:

$$\text{Triangle filter:} \qquad L_{\text{eff}} = \tfrac{4}{3} L_{\text{box}}$$
$$\text{3th-order filter:} \qquad L_{\text{eff}} = \tfrac{33}{20} L_{\text{box}}$$

Figure 4.15 ilustrates the effect of different filters kernels on LIC images. All images have been computed using the fast LIC technique described above. The differences between the box filter and higher-order filter kernels are subtile, but well noticeable (at least on a computer monitor). In order to make theses differences visible in a printed hardcopy, the images are shown using a relatively large magnification. It turns out that a triangle filter produces visually more pleasant result than a simple box filter. On the other hand, there is almost no difference between a triangle filter and a third-order B-spline. This means that it usually doesn't pay to use third- or even higher-order filter kernels. In fact, most of the examples shown in this work have been produced using a triangle filter (this includes all surface LIC textures, cf. Chapter 5). Recall from Section 4.5.3 that a triangle filter implies only minor additional costs compared to a box. Since field line integration, interpolation, and sampling of the input texture are the most expensive parts of the algorithm we found that overall costs only increase by about 10-15of a box.

In principle, convolution by a triangle filter can also be achieved by convolving twice with a box filter. The reason is that the order multiple convolutions can be exchanged. However, in practice some differences occur since the result of the first box filter convolution is stored in an intermediate image. This implies that pixel averages are computed. Thus, the effective filter kernel is not truely one-dimensional anymore. The results appear to be somewhat more blurry. At the same time spatial resolution of the LIC image is decreased a little bit. An example obtained by applying a box filter twice is shown in the lower part of Figure 4.15.

4.8 Animating LIC Textures

Although static LIC images clearly reveal the directional structure of a vector field, the actual *sign* of the field still remains ambiguous. Without further hints we cannot say whether vectors are pointing in forward or backward direction along a field line. An interesting approach to resolve this ambiguity is based on the animation of LIC textures.

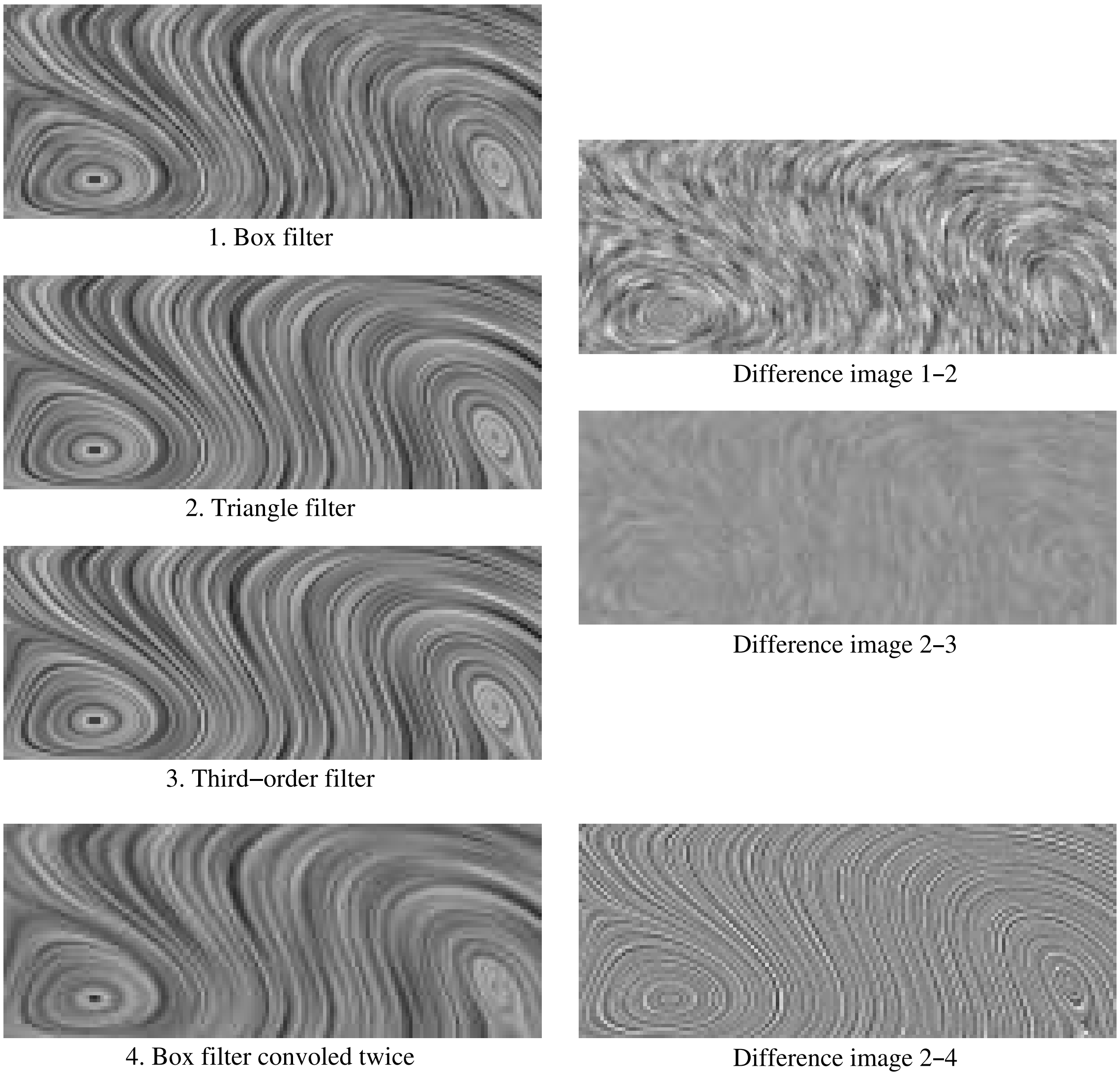


Figure 4.15: LIC images obtained with different filter kernels. The contrast of the three difference images has been increased for better visibility. A triangle filter is visually superior to a box filter, while there is not much difference between a traingle filter and a third-order B-spline, provided equal effective filter lengths have been chosen. In principal, convolving the result of a box-filter convolution with a box filter again should yield the same result as a single convolution using a triangle filter. However, storing the intermediate result already involve spatial averaging. Therefore a more blurry result is obatined.

The technique relies on a well-known perceptual phenomenon, namely that local phase changes in convolution filters may cause the impression of global motion. This effect can be exploited in computer graphics [32] and scientific visualization [34]. Its application to LIC has already been discussed in [9]. Although the speed of the motion effect can be controlled locally, one must keep in mind that the animation does not reflect the true physical motion of particles in an instationary flow field. Nevertheless, it provides an intuitive way for decoding sign and magnitude of a vector field.

4.8.1 Frame Blending

In order to generate an animated sequence of LIC images a periodically varying filter kernel need to be designed. In order to facilitate practical computations the overall support of the filter should be limited. The most simple solution to this problem is to cycle an arbitrary filter of length $2L$ periodically through some bigger interval of length $2L_a$. In Fig. 4.16 a) and b) this is illustrated for a box filter and for a triangle filter, respectively. However, the straight-forward cycling approach has a number of drawbacks. First of all, although initially a smooth filter kernel might have been chosen, discontinuities inevitably occur when the filter is shifted across the boundaries of the animation interval. Usually, the corresponding changes in the filter characteristics are well noticeable, disturbing the visual impression of the motion effect. Moreover, the discontinuities also complicate the use of the fast LIC algorithm. Although cycled filter kernels remain piecewise-polynomial functions, in general additional breakpoints and coefficients are introduced when expressing the kernel in terms of truncated power functions. This increases the number of arithmetic operations needed.

In order to avoid discontinuities at the boundaries of the animation interval, Cabral and Leedom [9] suggested to multiply the shifted kernel with a window function. In particular, they proposed to use a so-called Hann filter for both data windowing as well as for the shifted part itself. The resulting periodic LIC filter without normalization is

$$g(x,t) = (1 + cos(\kappa x))\,(1 + cos(n\kappa x + \omega t)), \quad t \in [0,1]. \tag{4.75}$$

with $\kappa = \pi/L$. The parameter n controls the number of ripples of the filter. For $n = 2$ the corresponding function is depicted in Figure 4.16 (c). At any point of a period the filter remains a smooth function. It always vanishes at the boundaries of the animation interval. Unfortunately, Eq. 4.75 is not a piecewise-polynomial function and therefore cannot be used together with the fast LIC algorithm at all. What we need is a simple periodic filter made up of piecewise-polynomial functions, which should not change their functional form within the whole period.

It turns out that such a filter can be constructed by smoothly blending two shifted non-periodic filter kernels. The individual filters are never moved across the boundary of the animation interval. Instead, they get less and less weighted as they approach the boundary. As soon as they receive zero weight they are instantaneously set back to the beginning of the interval. Figure 4.16 d), e), and f) illustrate how in this way a periodic function can be constructed from different non-periodic input filters. Denoting the original filter $g(x)$, the resulting periodic filter $\tilde{g}(x,t)$ is given by

$$\tilde{g}(x,t) = (1-t)\, g(x-(L_a-L)\, t) + t\, g(x+(L_a-L)(1-t)), \quad t \in [0,1]. \tag{4.76}$$

Here we assumed the extent of $g(x)$ to be $2L$, while the length of the animation interval is $2L_a$. The blending technique can easily be combined with the fast LIC algorithm. Since convolution is a linear operation it is possible to perform the blending operation in a post-processing step. In this case first a non-periodic motion sequence would be generated using a standard piecewise polynomial filter kernel g moving from $-L_a + L$ to $L_a - L$. The images of this sequence would then be combined similar to Eq. (4.76). Note, that in order to generate a periodic sequence of N frames, the original non-periodic sequence must contain $2N$ images.

4.8.2 Exploiting Temporal Coherence

Of course, it is not very efficient to compute $2N$ LIC images with slightly shifted filter kernels separately. A much better approach would be to modify the fast LIC algorithm so that not only spatial coherence is exploited but also temporal one. Note, that in case of a shifted filter kernel $\bar{g} = g(x-(L_a-L)\, t)$ we have

$$I(\boldsymbol{x},t) = I(\boldsymbol{x}-(L_a-L)\, t, 0). \tag{4.77}$$

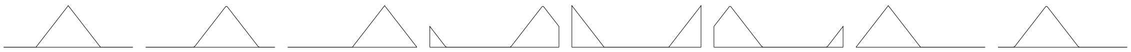
Therefore, pixel intensity at a given location in one frame is equal to pixel intensity at another location in another frame. In principal intensities for all frames can be computed in one pass, thus avoiding repeated computation of field lines and convolution sums.

For practical purposes we proceed as follows. In addition to the usual number of intensity updates N_u performed on a field line $|\, L_a - L\, |$ additional samples are computed on that line. These samples provide just enough information to determine LIC intensities for all standard sample locations for all time steps t. For every time step a separate accumulation buffer need to be maintained. All LIC intensities for a particular sample location are added to the associated buffers. Afterwards, for each pixel accumulated LIC intensities have to be normalized and frames have to be blended similar to Eq. (4.76).

An obvious drawback of this method is, that a large amount of memory is needed to hold a separate accumulation buffer for each frame. For example when using an accumulation buffer of 4-byte integers, a sequence of 25 video frames, 768×576 pixels each, would need about 85 MB of memory (recall that due to the mandatory blending 50 frames are needed to create a periodic sequence of 25 frames). The situation can be improved somewhat by performing the blending operation for each sample individually just before intensity values are added to a buffer. In this case only N instead of $2N$ buffers need to be stored.

One might think that there is a performance penalty associated with this approach because every sample has to be blended individually instead of just the final LIC intensities. However, note that usually the average number of samples per pixel is quite small. Also

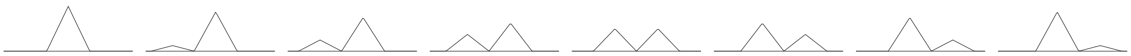
a) Cycled box filter.



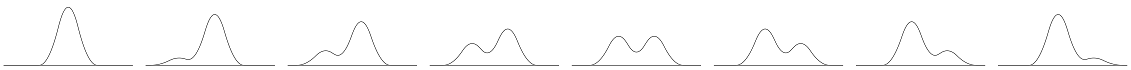
b) Cycled triangle filter.

c) Windowed Hann filter.

d) Blended box filter.



e) Blended triangle filter.



f) Blended third-order B-spline with overlap.

Figure 4.16: Animated LIC filter kernels.

note, that a final pass over all buffers which otherwise would be needed to perform frame blending is not neccessary anymore. In practice it turns out that due to cache effects on many computer architectures this more than compensates the costs for the additional blending operations.

For very large images memory requirements for maintaining N full-sized accumulation buffers may still be prohibitive. In this case LIC animation sequences can be computed on a parallel computer with distributed memory. We described a parallel extension of the fast LIC algorithm in [96].

4.8.3 Contrast Adjustment

In this section we want to point out a subtle peculiarity of LIC animation sequences. It turns out that auto-covariance of all periodic LIC filter kernels presented in the previous section doesn't stay constant during the animation but osciallates. Plots of auto-covariance for different filter kernels are depicted in Figure 4.17. For the cycled box filter (a) as well as for the cycled triangle filter (b) auto-covariance exhibits distinctive peaks in the middle of the animation interval. These peaks are caused by filter kernels reentering the animation interval at the beginning. The peaks are much less dominant in case of a windowed Hann filter (c) as well as for our favourite blended filter kernels (d,e,f).

However, for these filters variance σ^2 of the final LIC image changes. Remember, that variance is equal to auto-covariance for $x = 0$. Thus, these changes can be directly observed in Figure 4.17. Since variance directly controls contrast of a LIC image, images appear to get flatter during the animation.

In order to eliminate this visually disturbing artifact intensities have to be rescaled so that contrast remains constant. In every frame the resulting variance can be computed via Eq. (4.67). In case of a blended filter kernel this equation has quite a simple solution, provided the individual filters being blended don't overlap. This means, that the samples being averaged are statistically independent. In particular, we have

$$\sigma^2(t) = \sigma_0^2 \int_{-\infty}^{\infty} \tilde{g}^2(x,t)\,dx = \big((1-t)^2 + t^2\big)\ \sigma_0^2 \int_{-\infty}^{\infty} g^2(x)\,dx, \tag{4.78}$$

where σ_0^2 denotes variance of the input image. This expression states that variance is a parabolic function of t with a minimum at $t = 0.5$. In order to achieve a constant intensity distribution with mean value μ and varaince $\sigma^2 = \sigma^2(0)$ the blended LIC images have to be rescaled as follows:

$$I \leftarrow \frac{I - \mu}{\sqrt{(1-t)^2 + t^2}} + \mu \tag{4.79}$$

The situation gets more involved if both parts of the blended filter $\tilde{g}(x,t)$ overlap. Then the integral $\int \tilde{g}^2(x,t)\,dx$ is more difficult to compute. Analysing Eq. (4.76) or inspecting Figure 4.16 reveals that an overlap occurs if $L_a < 3L$. Let us consider the special case of a blended box filter $\tilde{b}(x,t)$. Assume, that both filter boxes b being blended overlap by a factor u, with $u \in [0,1]$. The contribution due to the overlap equals the integral over a squared box filter with length uL. Thus we obtain

$$\sigma^2(t,u) = \big((1-t)^2 + t^2 + 2\,(1-t)\,t\,u\big)\ \sigma_0^2 \int_{-\infty}^{\infty} b^2(x)\,dx. \tag{4.80}$$

If there is no overlap, i.e, $u = 0$, this result is equivalent to Eq. (4.78). On the other hand, if both filters fully overlap, i.e., $u = 1$, the preceding factor equals one and we obtain the same variance as for a single box filter b. In case of higher-order filters $\sigma^2(t,u)$ is quite tedious to compute analytically. Thus, in practice we determine the required scaling factors for contrast adjustment numerically.

4.8.4 Variable Speed Animation

Finally, we would like to mention that it is of course also possible to adapt the velocity of LIC filter animation locally. Such an approach can be used to emphasize vector magnitude in addition to directional sign.

Note, that if speed of filter animation is varied but length L_a of the animation interval is kept constant the resulting LIC sequence will not be periodic anymore. Forsell [30] described a simple technique for generating non-periodic variable speed LIC animations.

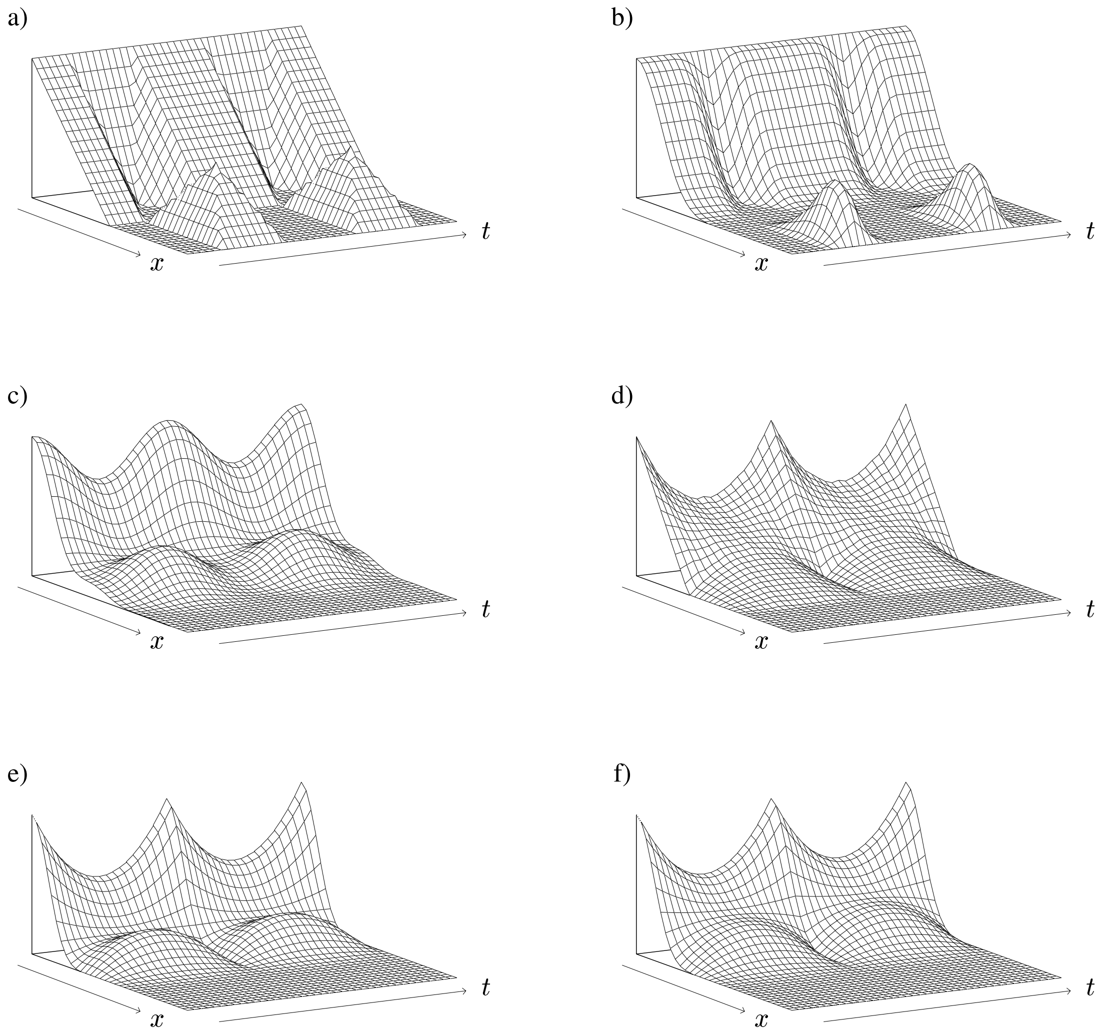


Figure 4.17: Auto-covariance for different animated LIC filter kernels: a) cycled box filter, b) cycled triangle filter, c) windowed Hann filter, d) blended box filter, e) blended triangle filter, f) blended third-order B-Spline with overlap. Auto-covariance at $x = 0$ turns out to be equal to variance of the resulting LIC image. The cycled filters are characterized by constant variance, but auto-covariance exhibits sharp peaks in the middle of the animation interval. The other filters produce less significant changes, but require the overall contrast to be adjusted.

She first computed a periodic constant speed animation sequence. In order to create variable speed animations intensity of a pixel was interpolated from those two frames of the constant speed sequence, which most closely resemble the actual phase shift of that pixel. However, there still remains a major problem. With ongoing time, filter kernel phases of neighbouring pixels will lose any correlation. Severe spatio-temporal aliasing effects are introduced. For example the texture may appear to move in the opposite direction in some areas.

A better alternative is to use the same phase shift for all pixels but to change the length of the animation interval accordingly. A filter moving through an interval just slightly bigger than the length of the filter itself appears to move very slow. On the other hand, a filter moving through a large interval in the same time appears to move very fast. Using the frame blending technique described above the length of the animation interval can be locally adapted in a natural way. However, note that in this case the amount of overlap of the two filter kernels being blended varies locally. In regions of very low velocity there is almost complete overlap, while in regions of large velocity there is no overlap at all. Thus, in order to adjust contrast of the resulting LIC images an equation like (4.79) cannot be applied globally. Instead, intensities have to be rescaled locally using a factor determined by the actual amount of overlap at that location.

Beside contrast adjustment generation of variable speed animation sequences also has some minor impact on the fast LIC algorithm itself. Like for constant speed sequences LIC intensities on a field line can be computed for all frames once the intensity differences $\Delta^p I_k^{\langle j \rangle}$ have been determined for sufficiently many samples. Of course, these samples can be obtained using fast LIC intensity updates. However, in contrast to constant speed animations the total number of samples required on a field line is not known in advance. It depends on velocity how much filter kernels are shifted and how many samples on a field line are required. Likewise, more than one additional update might be necessary in order to compute LIC intensity for a neigbouring location on a field line. This is the case if velocity at the neighbouring location is bigger and thus filter kernels need to be shifted by some larger distance. In practice it is most convenient to step a long a field line, evaluate the input texture at as many locations as necessary, then compute intensity differences, and finally combine these differences to generate LIC intensities for N_u points on the field line.

Chapter 5

Application to 3D Vector Fields

In the previous chapter we discussed the visualization of two-dimensional vector fields by means of line integral convolution (LIC). We now want to explore methods for visualizing three-dimensional fields. Obviously, this is much more difficult. Volumetric information cannot be depicted directly. Instead, some sort of projection has to be performed in order to obtain a 2D image. Depending on the particular point of view some parts of the volume will always be occluded by others. Therefore, an essential requirement for 3D visualization comprises the identification of the relevant structures in a volume. These structures have to be emphasized while less interesting information has to be suppressed.

LIC has proven to be a powerful method in 2D because it is able to depict a planar vector field at whole without suppressing any information at all. In principle, the method can also be used to synthesize 3D directional textures instead of 2D ones. However, it is not clear how to project these 3D textures into a 2D image. One possibility would be to perform some kind of direct volume rendering. In fact, some researchers investigated this approach [78, 45]. Unfortunately, direct volume rendering is quite slow, and – even worse – it is not clear how to highlight essential information in a volumetric field. In addition, often unsatisfactory results are obtained because the human eye is used to perceive surfaces instead of diffuse cloudy objects as generated by many volume rendering algorithms. Therefore, the extraction of *surfaces* from 3D vector fields seems to be more promising. In order to visualize directional structures *on* a surface, of course line integral convolution again is an excellent tool. For this reason, we want to consider the implementation of LIC on surfaces in the following sections.

However, before we do so we want to investigate how to compute suitable surfaces on which LIC textures can be mapped on afterwards. A very natural tool for analysing 3D vector fields are so-called *stream surfaces*. Built from a whole bunch of individual stream lines, these surfaces are well suited to reveal essential information in a vector field. Therefore, we first look at stream surfaces in more detail and dicuss how well-shaped triangular approximations can be constructed. In Sections 5.2 and 5.3 we then investigate algorithms for computing LIC textures on parametric as well as on more general non-parametric surfaces. Both approaches have specific pros and cons which will be pointed out during the presentation.

5.1 Stream Surfaces

As many other vector field visualization techniques stream surfaces have been utilized in fluid mechanics first. In fact, they are quite a popular tool supported by many standard flow visualization packages. Of course, stream surfaces can be applied to general vector fields, too. However, in this case care has to be taken because general vector fields may bear strongly divergent terms or even may contain sinks and sources. These features may cause problems for standard stream surface algorithms. After reviewing stream surfaces in fluid mechanics we therefore present an improved algorithm for constructing accurate and well-shaped triangular stream surfaces in arbitrary 3D vector fields. We also mention ways of defining proper initial seed lines for stream surfaces automatically.

5.1.1 Background

Recall from Section 2.1.3 that a stream surface $\boldsymbol{x}(t,u)$ is mathematically defined as a set of infinitely many stream lines released from a common seed line $\boldsymbol{m}(u)$,

$$\frac{d}{dt}\boldsymbol{x}(t,u) = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0,u) = \boldsymbol{m}(u). \tag{5.1}$$

A stream surface is naturally parametrized by the two variables t and u. Lines of constant u are just ordinary stream lines while lines of constant t are commonly called time lines.

In case of incompressible flow fields, i.e., vector fields with vanishing divergence, a stream surface can be represented by a so-called *stream function* S. A stream function is a scalar field the gradient of which is always perpendicular to the direction of the associated flow field,

$$\boldsymbol{f} \cdot \nabla S = 0. \tag{5.2}$$

For a single vector field there exist infinitely many stream functions. It can be shown [94] that an incompressible vector field can be expressed as the cross product of the gradients of two dual stream functions S_1 and S_2,

$$\boldsymbol{f} = \nabla S_1 \times \nabla S_2. \tag{5.3}$$

Any linear combination of two stream functions will again be a stream function. On a stream surface the value of the corresponding stream function is constant:

$$S = \alpha_1 S_1 + \alpha_2 S_2 = \text{const.} \tag{5.4}$$

The use of scalar-valued stream functions is quite popular in fluid mechanics, since mass flux and many flow fields are assumed to be incompressible.

The representation of a stream surface as an isosurface of a stream function can be exploited algorithmically. Van Wijk [89] described methods for computing a pair of dual stream functions. With these functions he was able to construct stream surfaces by means of a standard isosurface algorithm. However, this method cannot be applied to vector fields with non-zero divergence. Therefore, in the following we want to investigate alternative methods for creating stream surfaces.

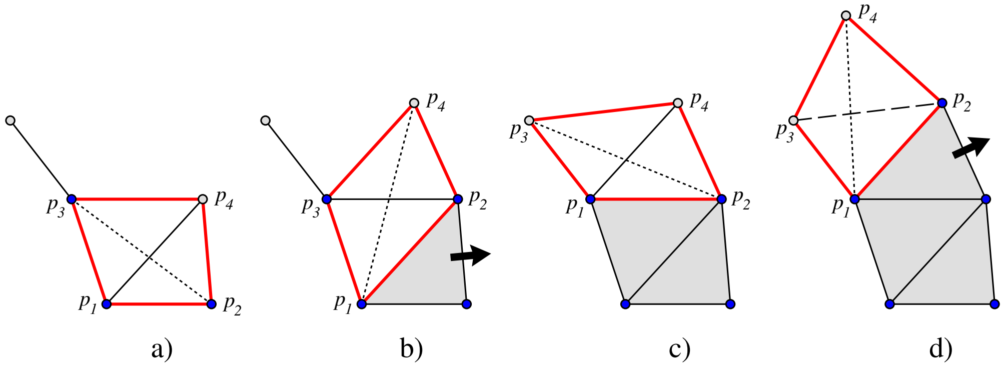


Figure 5.1: In Hultquist's algorithm triangles between neighbouring stream lines are created recursively. The method *advance_ribbon()* analyzes quadrilaterals as shown above. It creates left- or right-aligned triangles depending on which of the two diagonals of a quadrilateral is shorter. The black arrows in b) and d) indicate recursive invocation of *advance_ribbon()*.

5.1.2 Hultquist's Algorithm

The standard approach for constructing stream surfaces relies on tracing a set of n individual stream lines released from a predefined seed line. Neighbouring stream lines are connected by triangles in order to obtain a polygonal approximation of the surface. In the most simple case a regular triangulation scheme is applied, i.e., each stream line is approximated by m points and the resulting mesh of $(n-1) \times (m-1)$ quadrilaterals is triangulated regularly by pairs of triangles.

Obviously, this naive approach produces ill-shaped triangles if neighbouring stream lines move away from each other or if the flow exhibits strong shear. Better results can be obtained using an *advancing front* algorithm proposed by Hultquist in 1992 [44]. The general idea of this algorithm is to propagate the initial seed line more or less perpendicular to the flow by means of a recursive triangulation method. Along a single stream line equi-distant steps are taken. Whenever the distance between two neighbouring stream lines gets too large, a new line is inserted. Likewise, a stream line is removed whenever the distance between its two neighbouring lines gets too small.

In Hultquist's algorithm every stream line is represented by a so-called *tracer*. A tracer is able to compute the next point on its associated curve. It also temporarily stores the last three points of the curve. Two of these points are already part of the triangulation, while the third and front-most point does not yet belong to any triangle. The heart of the algorithm is a recursive method *advance_ribbon()*. This method produces triangles right from a given tracer and propagtes neigbouring stream lines as necessary. It is assumed that the initial seed line is not closed. At the beginning the left-most tracer of the stream surface is propagated by one step and *advance_ribbon()* is called for this tracer. The steps performed by *advance_ribbon()* are illustrated in Figure 5.1. Let us look at these steps in more detail, ignoring insertion and deletion of tracers for the moment.

- The method tries to create triangles right from the edge $\boldsymbol{p}_1\boldsymbol{p}_3$ shown in Figure 5.1 a). The quadrilateral formed by the points $\boldsymbol{p}_1\boldsymbol{p}_2\boldsymbol{p}_4\boldsymbol{p}_3$ is considered. Depending on

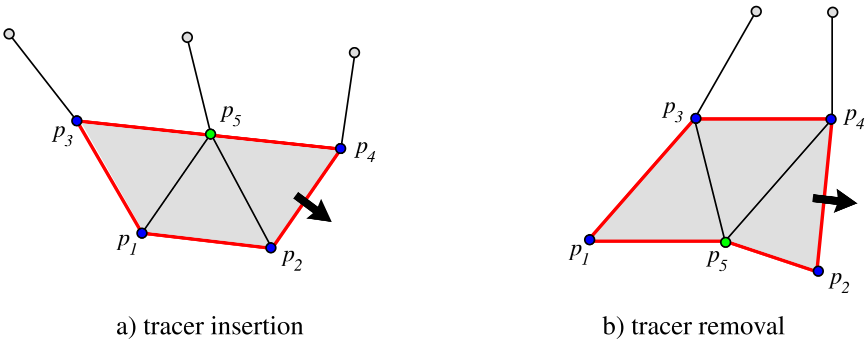


Figure 5.2: If the distance $|\boldsymbol{p}_3-\boldsymbol{p}_4|$ between two neighbouring stream lines get too large a new tracer need to be inserted. In Hultquist's algorithm the resulting patch of five points is triangulated as shown in a). If a tracer is to be deleted, the triangulation scheme shown in b) is applied.

whether the distance $|\boldsymbol{p}_1-\boldsymbol{p}_4|$ or $|\boldsymbol{p}_2-\boldsymbol{p}_3|$ is shorter either a triangle aligned to the left stream line or to the right one is created.

- If the triangle $\boldsymbol{p}_1\boldsymbol{p}_2\boldsymbol{p}_4$ has been selected the right tracer is propagated by one step and *advance_ribbon()* is called recursively in order to generate triangles right from the newly created face. This is indicated by the black arrow in Figure 5.1 b).

- Next, a propagated quadrilateral is considered. Again, the lengths of the diagonals are compared and either a left- or right-aligned triangle is created. If a left-aligned triangle has been chosen the method doesn't terminate immediately but tries to create more triangles aligned to the right. However, a right-aligned triangle is only accepted if the distance $|\boldsymbol{p}_1-\boldsymbol{p}_4|$ is shorter than the length of the *previous* diagonal, i.e., shorter than $|\boldsymbol{p}_1-\boldsymbol{p}_2|$ in Figure 5.1 c).

- The method terminates after exactly one left-aligned triangle has been created and no more right-aligned triangles are accepted. Stream surface generation is continued by propagating the very first tracer one more step and calling *advance_ribbon()* for this tracer again.

Note, that it might happen that no right-aligned triangle at all will be created. This is the case if initially $|\boldsymbol{p}_3-\boldsymbol{p}_4| > |\boldsymbol{p}_2-\boldsymbol{p}_3|$. On the other hand, two or more right-aligned triangles are also possible. In this way the algorithm tries to keep the current front always perpendicular to the flow. Consequently, the total number of steps taken by a tracer need not to be constant.

It has already been mentioned that it might be necessary to insert new tracers into the current front if the distance between two neighbouring stream lines gets too large. Also, if stream lines converge it is favourable to remove tracers from the front. Hultquist proposed to triangulate patches of five points each as shown in Figure 5.2 a) and b), respectively. Right from these patches, stream surface generation is continued using the ordinary *advance_ribbon()* method described above.

5.1.3 A 2D Case Study

In order to evaluate the outlined stream surface algorithm and to identify possible problems we applied it to compute a stream surface for the 2D vector field shown in Figure 5.3. The field contains two critical points, a sink and a saddle. Stream lines were started along a vertical line on the left side of the field. The resulting triangulation is shown in Figure 5.4 a). Green patches indicate that a new tracer has been inserted, while red patches indicate that a tracer has been removed. Other triangles are shown either in white or in light grey, illustrating the outer loop of the advancing front method. Along the top-most stream line white and grey triangles alternate after every step. This is because *advance_ribbon()* was always called for this line first.

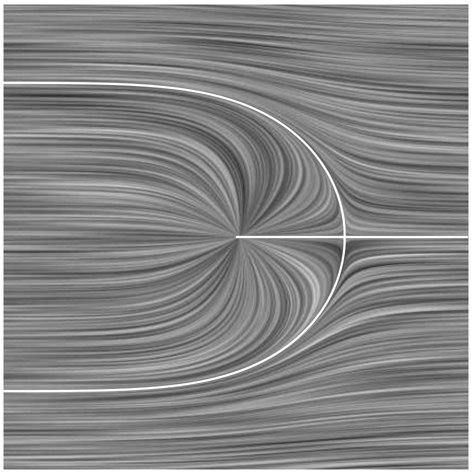


Figure 5.3: Vector field containing a sink and a saddle. This field was used to evaluate stream surface algorithms.

Originally, Hultquist's stream surface algorithm was designed specifically to visualize flow fields from fluid mechanics. In such fields node-type singularities like sinks don't occur. Consequently, the impact of these singularities on the algorithm wasn't dicussed in the original paper. In our implementation we simply didn't generate any triangles whenever a tracer terminated in a sink. Notice, that in the vicinity of the sink thin peaked triangles are produced.

In contrast to nodes, saddle-type singularities occur quite frequently in flow fields. Saddles represent a severe problem for tracing algorithms because stream lines started in different sectors of a saddle diverge more and more. At the beginning well-shaped triangles can still be obtained by adding additional tracers. However, in the vicinity of the saddle inevitably very thin, almost degenerated triangles occur. Hultquist detected a saddle by computing the average length and height of a qudrilateral. Whenever this ratio exceeded a certain threshold the front of tracers was split and both parts were propagated separately in opposite directions. Of course, this meant that the saddle point itself as well as the two divergent separatrices weren't located exactly. Therefore, the triangulation in Figure 5.4 a) doesn't fill the area around the saddle point completely.

Another problem of the outlined algorithm concerns propagation of the front in case a tracer has been inserted. In order to keep the front perpendicular to the flow, the number of right-aligned triangles produced by *advance_ribbon()* isn't fixed but is adapted locally. This ensures that more or less equi-lateral triangles are generated. However, a similar adapative startetgy isn't applied when a tracer is inserted or deleted. As illustrated in Figure 5.2, in these cases exactly one right-aligned triangle is generated and triangulation is always continued at the corresponding edge. Therefore in Figure 5.4 a) a number of thin triangles occur in the area between the sink and the saddle.

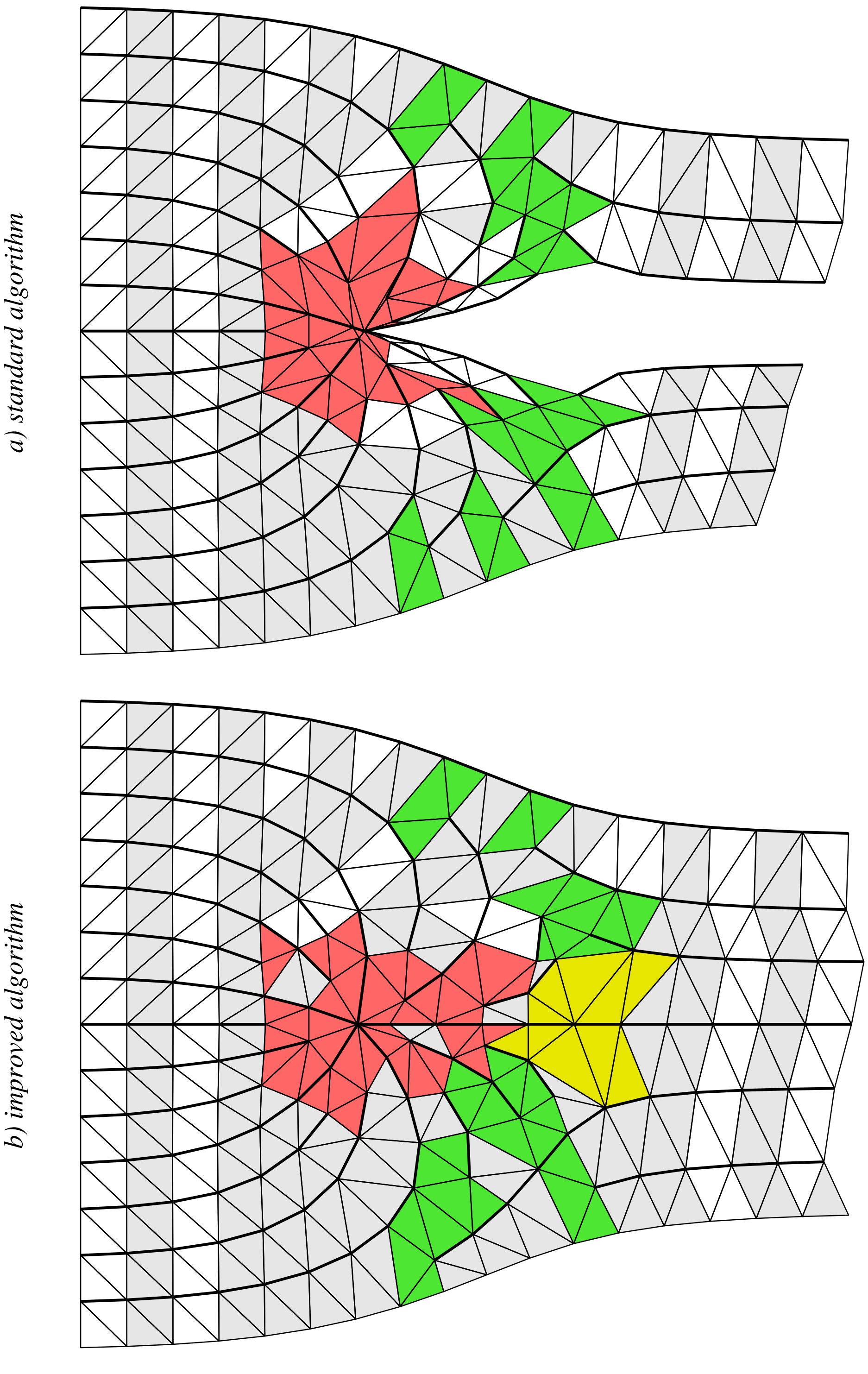


Figure 5.4: Stream surfaces for the 2D vector field shown in Figure 5.3. Hultquist's original algorithm (upper part) produces bad triangles in the vicinity of the singularity. The saddle point isn't contained in the triangulation at all. Better results are obtained using an improved method (lower part). In this case critical points are located explicitely and insertion and deletion of tracers is handled slightly different.

5.1.4 Determining Critical Points

To improve the quality of stream surface triangulation for general vector fields first of all critical points of the field have to be identified correctly and the triangulation strategy has to be modified in the neighbourhood of these points. Two types of critical points have to be distinguished, sinks and saddles. For both cases we developed different strategies. We will discuss these strategies separately below. The result of the modified stream surface algorithm applied to our test case is shown in Figure 5.4 b).

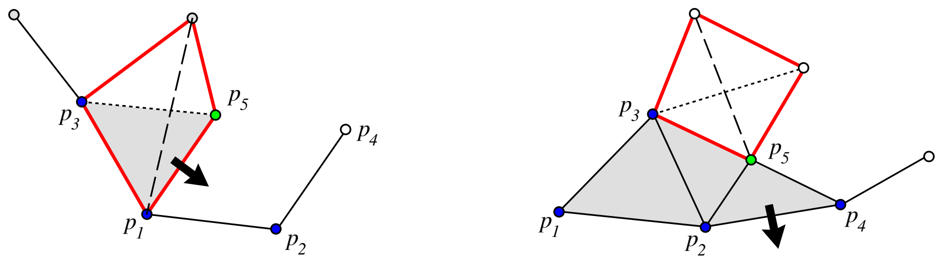
Sinks. Stream surfaces are constructed by taking equi-distant steps along individual stream lines. However, often the last step before a sink will to be shorter since stream lines terminate. Consequently, a number of small ill-shaped triangles are generated near a sink. Better result can be obtained by taking a longer step prior to a sink instead of a shorter one. To achieve this not only the next but the next *two* points on a stream line have to be precomputed and stored by a tracer. Whenever computation of the front-most point fails because a singularity has been detected, the next point (which is not yet part of the triangulation) can be shifted ahead exactly into the sink. Usually, the location of a sink cannot be determined exactly by a numerical integrator. Therefore, different tracers will terminate at slightly different locations. To obtain a consistent triangulation in spite of that, all singularities are stored in a global list. When a tracer detected a sink, it is first tested whether the sink is only a small distance apart from a previously encountered singularity. If this is the case, the coordinates of previous singularity are used. The resulting triangulation then will not contain cracks or wholes.

Moreover, for stream lines ending in a common sink the last patch between both lines will be degenerated. In this situation the patch is filled by just a single triangle, instead of trying to generate multiple faces by means of the standard *advance_ribbon()* method. A similar strategy is applied if a tracer has to be deleted and the front-most points of a patch coincide.

Saddles. Instead of comparing width and height of a quadrilateral, we detect saddles by computing the angle between the flow vectors at point $\boldsymbol{p}_3$ and $\boldsymbol{p}_4$ of a quadrilateral. If this angle exceeds a certain threshold, e.g. 150 degrees, a saddle is assumed. A saddle is a critical point of the vector field $\boldsymbol{f}$, i.e., vector magnitude will be zero at this point. Therefore, we can locate the saddle by minimizing $\boldsymbol{f}^2$. Our visualization software is able to compute the derivative $\boldsymbol{f}'$ of an arbitrary vector field. We therefore decided to apply a conjugate gradient method for minimization. In particular, we make use of the standard library routine *frprmn()* described in [70]. The gradient of $\boldsymbol{f}^2$ can be expressed by means of $\boldsymbol{f}'$ as follows:

$$\nabla \boldsymbol{f}^2 = 2\boldsymbol{f}'\boldsymbol{f} = 2\left[\sum_i \frac{\partial f_i}{\partial x_j} f_i\right] \tag{5.5}$$

After the coordinates of the saddle have been determined, the next step comprises the computation of the eigenvectors of $\boldsymbol{f}'$ at the critical point. These vectors determine the direction of the separatrices of the saddle. Again, for computing the eigenvectors of



a) $|\boldsymbol{p}_1 - \boldsymbol{p}_5| \leq |\boldsymbol{p}_2 - \boldsymbol{p}_3|$ b) $|\boldsymbol{p}_1 - \boldsymbol{p}_5| > |\boldsymbol{p}_2 - \boldsymbol{p}_3|$

Figure 5.5: Improved triangulation scheme for tracer insertion. Like in the ordinary *advance_ribbon()* method distances between certain points of a patch are considered in order to determine what triangles are to be generated. The black arrows indicate that triangulation is continued recursiveley right from these edges.

the general real matrix $\boldsymbol{f}'$ we apply standard library routines (LAPACK). From the three different eigenvectors of $\boldsymbol{f}'$ we chose the one most parallel to the quadrilateral which indicated the saddle. Using the direction of this vector we insert two new tracers a small distance above and below the critical point. With the help of these tracers a triangulation can be obtained which exactly includes the saddle point. The newly inserted tracers represent the divergent separatrices of the saddle. If the distance between a newly inserted tracer and its right or left neighbours is too small, the neighbouring tracers are deleted. In this way, in total either 3, 4, or 5 triangles are produced. The triangles produced in the neighbourhood of a saddle are drawn in yellow in Figure 5.4 b). Since in this example the surface approaches the saddle from two sides, actually four separatrices were found. To ensure that exactly the same coordinates are used on these curves, again saddles can be stored in a global list.

5.1.5 Further Improvements

Beside the modifications required to handle sinks and saddles correctly, we employed some other strategies which also helped to improve the quality of stream surface triangulation. In this section we want to summarize these changes briefly.

Tracer Insertion. When a tracer is inserted into the front triangulation should be adapted to the shape of the patch. Similarly to the ordinary *advance_ribbon()* method, this can be achieved by considering distances between different points of the patch. In particular, we first compare the distances $|\boldsymbol{p}_1 - \boldsymbol{p}_5|$ and $|\boldsymbol{p}_2 - \boldsymbol{p}_3|$ as shown in Figure 5.5. If $|\boldsymbol{p}_1 - \boldsymbol{p}_5| \leq |\boldsymbol{p}_2 - \boldsymbol{p}_3|$ then a single triangle $\boldsymbol{p}_1\boldsymbol{p}_5\boldsymbol{p}_3$ is created. Triangles right from the edge $\boldsymbol{p}_1\boldsymbol{p}_5$ are created by recursive invocation of *advance_ribbon()*. In addition, it is checked if more right-aligned triangles attached to $\boldsymbol{p}_3$ should be created. Again, this is done using exactly the same strategy as in the regular case. On the other hand, if $|\boldsymbol{p}_1 - \boldsymbol{p}_5| < |\boldsymbol{p}_2 - \boldsymbol{p}_3|$ at least three triangles are created. As indicated by the black arrow in Figure 5.5 triangles right from $\boldsymbol{p}_2\boldsymbol{p}_4$ are generated by recursive invocation of *advance_ribbon()*. If necessary, more triangles attached to $\boldsymbol{p}_3$ are created in the usual way.

Note, that if the tracer front is more or less perpendicular to the flow just the standard triangulation shown in Figure 5.5 a) is obtained. However, in case of more distorted patches better results are obtained. Propagation of the front might be terminated or additional rows of triangles might be started. As a result, the tendency of the front to be aligned perpendicular to the flow is improved.

It should be pointed out that the new insertion scheme requires some care near a saddle point or near an obstacle. In this case quite degenerate patches may occur. These may cause ill-shaped triangles to be produced if the location $\boldsymbol{p}_5$ of the new tracer isn't chosen properly. In particular, problems occur if the distance between the next point $\boldsymbol{p}_6$ computed by a newly inserted tracer and $\boldsymbol{p}_3$ or $\boldsymbol{p}_4$ gets too small. In order to prevent this situation we chose the location of a new tracer such that these distances are roughly the same, compare Figure 5.6. This is done via a bracketing approach, i.e., by iteratively adjusting the tracer's initial position. Instead of chosing this position along the line $\boldsymbol{p}_3\boldsymbol{p}_4$ it is favourable to place the tracer along $\boldsymbol{p}_1\boldsymbol{p}_2$ and to perform an additional step afterwards. The new tracer will then stay closer to the true stream surface.

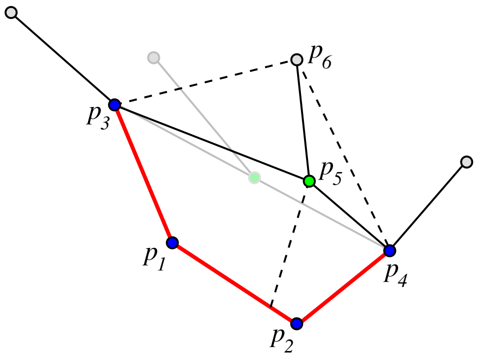
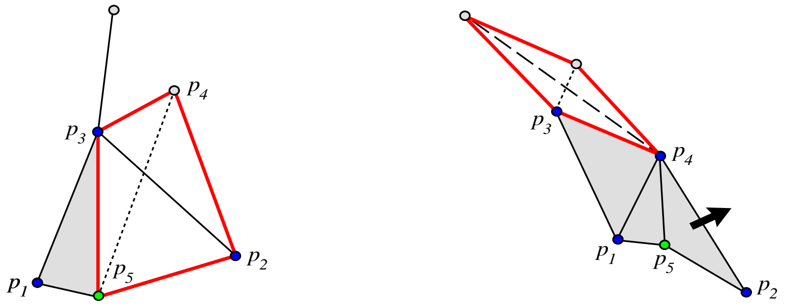


Figure 5.6: New tracers are inserted such that the distances $\boldsymbol{p}_3\boldsymbol{p}_6$ and $\boldsymbol{p}_4\boldsymbol{p}_6$ are roughly the same.

Tracer Removal. Let us now discuss the removal of a tracer. Usually, the distance $|\boldsymbol{p}_3 – \boldsymbol{p}_5|$ shown in Figure 5.7 will be smaller than $|\boldsymbol{p}_1 – \boldsymbol{p}_4|$. Then the triangle $\boldsymbol{p}_1\boldsymbol{p}_5\boldsymbol{p}_3$ will be created like in the standard triangulation scheme. However, the remaining quadrilateral $\boldsymbol{p}_5\boldsymbol{p}_2\boldsymbol{p}_4\boldsymbol{p}_3$ will be processed by means of the ordinary *advance_ribbon()* method. Therefore, it is possible that the second triangle gets attached to $\boldsymbol{p}_3$ instead of $\boldsymbol{p}_5$. Also, a third triangle will not necessarily be created. On the other hand, if $|\boldsymbol{p}_3 – \boldsymbol{p}_5| > |\boldsymbol{p}_1 – \boldsymbol{p}_4|$ then always three triangles will be created, c.f. Figure 5.7 b). Afterwards, triangulation will be continued right from $\boldsymbol{p}_2\boldsymbol{p}_4$. In addtion, it is checked whether more triangles attached to $\boldsymbol{p}_3$ should be created or not.

While the insertion of a tracer can be detected "on the fly" deletion should be handled separately. We would like to remove the most narrow ribbon of the current front first. This can be achieved by deleting either the tracer to the left or to the right of that ribbon. We usually select the one with the smallest distance to the opposite stream line. Instead of simply computing point distances only contributions perpendicular to the flow are considered. This ensures that tracers get also removed in distorted situations. After the optimal set of obsolete stream lines has been determined all selected tracers are marked and will be deleted as soon as they are encountered by *advance_ribbon()*. The detection of the tracers to be deleted is performed just before a new row of triangles is started, i.e., before the invocation of *advance_ribbon()* for the very first tracer and before recursive invocation of this method for all right-aligned triangles except for the first one on each level.



a) $|\boldsymbol{p}_3 - \boldsymbol{p}_5| \leq |\boldsymbol{p}_1 - \boldsymbol{p}_4|$ b) $|\boldsymbol{p}_3 - \boldsymbol{p}_5| > |\boldsymbol{p}_1 - \boldsymbol{p}_4|$

Figure 5.7: A stream line is removed from the current front if the distance between its two neighbours gets too small. Instead of always using a standard triangulation scheme triangulation better should be adapted to the shape of the patch. Triangles right from $\boldsymbol{p}3\boldsymbol{p}5$, respectively $\boldsymbol{p}2\boldsymbol{p}4$ are created by means of *advance_ribbon()*.

The modified stream surface algorithm performed very well in practice. Examples of stream surfaces computed using the new method are for example shown in Figure 1.1 on page 6, Figure 1.3 on page 8 and Figure 5.12 on page 124. The last example contains a 3D version of the vector field we used in our 2D case study. Like in the 2D case sinks and saddles were precisely located by the algorithm.

5.1.6 Seed Line Selection

An important issue for stream surface generation comprises the choice of the initial seed line $\boldsymbol{m}(u)$ along which tracers are started. For case of simplicity this line is often chosen parallel to one of the major coordinate axes or it is even defined completely by hand. While the first approach often results in unsatisfactory surfaces the latter method obviuously is very time-consuming.

A good automatic seed line selector should assure that $\boldsymbol{m}(u)$ is oriented perpendicular or nearly perpendicular to the flow. However, at each point in a vector field there are infinitely many such directions. From these directions two of them are prominent, namely the curvature vector $\boldsymbol{n}$ as well as the binormal vector $\boldsymbol{b}$ at each point. Together with the tangent vector $\boldsymbol{t}$ these vectors form the so-called *Frenet frame* of a field line [7]. Cai and Heng [11] pointed out that especially meaningful stream surfaces can be constructed by choosing the initial seed line along the binormal vectors of the local Frenet frame. If this is done, surface normals will coincide with the curvature vectors of the stream lines, at least initially. In order to determine a local Frenet frame the first and second derivatives of a stream line $\boldsymbol{x}(s)$ with respect to arc-length s have to be computed:

$$\boldsymbol{t} = \frac{d\boldsymbol{x}}{ds} = \frac{\boldsymbol{f}}{|\boldsymbol{f}|}, \qquad \boldsymbol{n} = \frac{d^2\boldsymbol{x}}{ds^2} = \frac{\boldsymbol{f}'\boldsymbol{f}}{|\boldsymbol{f}|^2} - \frac{\boldsymbol{f}(\boldsymbol{f}'\boldsymbol{f})}{|\boldsymbol{f}|^4}\,\boldsymbol{f} \tag{5.6}$$

The binormal vector is given by $\boldsymbol{b} = \boldsymbol{t} \times \boldsymbol{n}$. Suitable seed lines can be obtained by integrating along the direction of $\boldsymbol{b}$ starting from a predefined center point. The location of this point can be conveniently changed interactively, for example by means of a 3D cross-hair dragger. All stream surfaces in this work have been constructed in this way. Straight initial seed lines are obtained by merely considering the binormal direction at the center point instead of updating this vector locally.

5.2 Surface LIC in Parameter Space

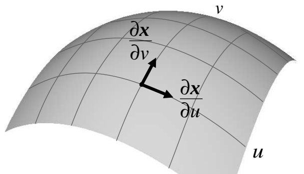
Stream surfaces provide an excellent tool to gain insight into the structure of a 3D vector field. However, while we know that the flow of a vector field is constrained to its stream surfaces the exact structure of the flow within such a surface often remains unclear. One possible solution to this problem is to map a directional LIC texture onto the surface. Topologically, a stream surface is not different from an ordinary two-dimensional Euclidean plane. Therefore, the generalization of 2D visualization techniques such as LIC seems to be feasible.

In this section we first want to investigate the generation of LIC textures for parametric surfaces. In this case, line integral convolution can be performed completely in 2D parameter space. The resulting LIC texture can be mapped onto the surface afterwards in a straight-forward way. In order to precisely understand the transformation between 3D physical space and 2D parameter space we start our discussion introducing parametric surfaces formally. Later on, we consider algorithmic issues and outline how to cope with the problem of texture distortion.

5.2.1 Parametric Surfaces

A parametric surface or *manifold* M is defined by

$$\boldsymbol{x}(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} u \\ v \end{pmatrix} \in \mathbb{R}^2. \qquad (5.7)$$



The cartesian coordinates x, y, z of a surface point $\boldsymbol{x}$ are assumed to be differentiable functions of the parameters u and v. The derivatives $\partial\boldsymbol{x}/\partial u$ and $\partial\boldsymbol{x}/\partial v$ are always tangential to the surface. A parametrization is said to be *regular* if both vectors are linearly independent, i.e., if they form a basis of the *tangent space* $TM_{\boldsymbol{x}}$ of M at $\boldsymbol{x}$. Note, that stream surfaces are parametric surfaces by definition. In this case lines of constant u and v correspond to stream lines and time lines, respectively. While parametric surfaces can be parametrized globally using 2D Euclidean coordinates, general manifolds usually can only be parametrized locally.

Suppose, $\boldsymbol{u}(t)$ is an integral curve of a 2D vector field $\boldsymbol{g}$ defined in parameter space. By means of Eq. (5.7) we obtain a 3D curve $\boldsymbol{x}(\boldsymbol{u}(t))$ on M. Differentiating this curve with respect to t yields the tangential vector

$$\frac{d\boldsymbol{x}}{dt} = \frac{\partial \boldsymbol{x}}{\partial u}\frac{du}{dt} + \frac{\partial \boldsymbol{x}}{\partial v}\frac{dv}{dt} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{u}}\boldsymbol{g}. \tag{5.8}$$

By mapping the whole 2D vector field $\boldsymbol{g}$ onto the surface we obtain a tangential vector field $\boldsymbol{f}$ on M:

$$\boldsymbol{f} : \boldsymbol{x} \ni M \mapsto TM_{\boldsymbol{x}}, \quad \boldsymbol{f} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{u}}\boldsymbol{g} \tag{5.9}$$

In order to compute a 2D vector field $\boldsymbol{g}$ from its corresponding tangential field $\boldsymbol{f}$ we need to invert this equation. In particular, we have to solve a linear system consisting of three equations, one for each component of $\boldsymbol{f}$, but only two unknowns, du/dt and dv/dt. In order to obtain a 3×3 system we consider a third linearly independent parameter w, introduce a new component dw/dt and define

$$\frac{\partial \boldsymbol{x}}{\partial w} \equiv \frac{\partial \boldsymbol{x}}{\partial u} \times \frac{\partial \boldsymbol{x}}{\partial v}. \tag{5.10}$$

The resulting regular system may be solved for example using Cramer's rule. The components of the 2D vector field $\boldsymbol{g}$ are then given by

$$\frac{du}{dt} = \frac{\det\left[\frac{d\boldsymbol{x}}{dt}, \frac{\partial \boldsymbol{x}}{\partial v}, \frac{\partial \boldsymbol{x}}{\partial w}\right]}{\det\left[\frac{\partial \boldsymbol{x}}{\partial u}, \frac{\partial \boldsymbol{x}}{\partial v}, \frac{\partial \boldsymbol{x}}{\partial w}\right]}, \qquad \frac{dv}{dt} = \frac{\det\left[\frac{\partial \boldsymbol{x}}{\partial u}, \frac{d\boldsymbol{x}}{dt}, \frac{\partial \boldsymbol{x}}{\partial w}\right]}{\det\left[\frac{\partial \boldsymbol{x}}{\partial u}, \frac{\partial \boldsymbol{x}}{\partial v}, \frac{\partial \boldsymbol{x}}{\partial w}\right]}. \tag{5.11}$$

If the 3D vector $d\boldsymbol{x}/dt$ is not tangential to M there will also be a non-vanishing component dw/dt. The 2D parameter space vector computed by Eq. (5.11) then corresponds to the projection of the 3D vector onto the surface.

Parametric surfaces are easily extracted from 3D curvilinear grids by keeping one grid index fixed. Such grids are often used in CFD simulations, conforming to shape of a body in the flow. The 2D vector field corresponding to the projection of the 3D flow field onto a grid plane can be obtained from Eq. (5.11). However, in this case a non-tangential vector $\partial\boldsymbol{x}/dw$ need not to be computed via Eq. (5.10). Instead it can be taken directly from the curvilinear grid itself. The matrix

$$J = \left[\frac{\partial \boldsymbol{x}}{\partial u}, \frac{\partial \boldsymbol{x}}{\partial v}, \frac{\partial \boldsymbol{x}}{\partial w}\right] \tag{5.12}$$

is just the Jacobian matrix which transforms between physical space and parameter space.

It should be mentioned, that the mapping between physical space and parameter space may introduce numerical errors. First, in parameter space usually the vectors are interpolated linearly, although the mapping itself is non-linear. Second, often the Jacobian matrix has to be approximated by taking finite differences. Both types of errors can be controlled by choosing the resolution of the surface grid to be sufficiently fine.

5.2.2 The Surface LIC Algorithm

In the previous section we sketched how to transform 3D vectors defined on a parametric surface into 2D parameter space. Discrete approximations of parametric surfaces may either exhibit regular structure, e.g., if surfaces are extracted from a 3D curvilinear grid, or may be given by an unstructured triangular mesh, e.g., if Hultquist's stream surface algorithm is applied. Usually, 3D vectors are stored at the vertices of such a surface mesh. After transforming these vectors into parameter space we obtain a 2D vector field defined at the nodes of a 2D uniform or triangular grid.

For the 2D vector field a conventional LIC texture can be computed by means of an arbitrary LIC algorithm. Existing 2D code can be used without any modification. Afterwards the resulting LIC texture has to be mapped back onto the surface again. This task is commonly called *texture mapping* in computer graphics. Today, it is supported by many 3D graphics libraries such as OpenGL [63] and, in fact, is implemented in hardware on modern graphics workstations. The final textured surface can be displayed at truely interactive frame rates on such computers. This is an important feature for analyzing the structure of 3D surface flows.

The outlined algorithm has been proposed for the first time by Lisa Forssell in 1994 [30]. She computed LIC images in parameter space to visualize the flow over the surface of a space shuttle. Actually, Forssell even generated animated LIC sequences in parameter space. These animation sequences were again mapped onto the surface and could be played back in real-time on a fast graphics computer.

5.2.3 Compensating Texture Distortions

The main problem behind the surface LIC approach described above are distortions introduced by the non-isometric mapping between parameter space and physical space. The cell sizes of the parametric surface might differ significantly. Therefore, in some areas of the surface the mapped LIC texture is compressed while in others it is stretched. For visualization purposes it is often desireable that the texture has equal characteristics across the whole surface. For the same reason we prefered using arc-length parametrization when computing 2D LIC images. This ensures that the features in a LIC texture are of equal size throughout the whole image, compare Section 4.3.2. Another point is that varying resolution of surface texture might interfere with the depth perception of an object. Usually, parts of an object with high frequency texture appear to be further away then parts with low frequency texture.

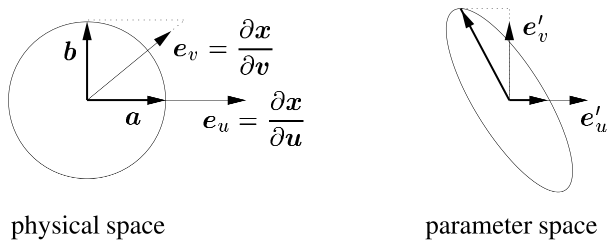
As a first remedy Forssell suggested to adjust the length of the LIC filter kernel so that it is constant in physical space [30, 31]. Recall that filter length determines the characteristic size of features along a stream line in LIC images. Therefore, if filter length is scaled properly in parameter space the distortions of the mapping into physical space can be compensated – at least partially. The method also helps to compensate errors in the perceived texture speed in surface LIC animations. The filter length formula suggested by Forssell was

$$L(\boldsymbol{x}) = a + b\, r(\boldsymbol{x}), \quad \text{with} \quad r(\boldsymbol{x}) = \frac{|d\boldsymbol{u}/dt|}{|d\boldsymbol{x}/dt|}. \tag{5.13}$$

The quantity r just denotes the ratio between vector magnitude in parameter space and vector magnitude in physical space. This ratio is low in sparse areas of the grid and high in dense areas. The constant contribution a is a non-vanishing minimum filter length.

Of course, better results can be obtained if not only filter lengths are scaled, but if also the characteristics of the input texture were adapted locally according to the resolution of the surface mesh. The goal is to create input noise which has equal properties everywhere after being mapped onto the surface. Consequently, features of the noise image should be larger in areas which are mapped into a small region in physical space and vice versa. Since distortions in u and v direction might be different such a noise image need not to be isotropic anymore. Nevertheless, some attempts have been made to reduce texture distortion by means of isotropically scaled input noise. For example, Thomas Loser [55] applied a technique originally developed by Kiu and Banks [50] to generate multi-frequency input noise for LIC. In order to determine noise frequency at a particular location Loser computed how much a unit vector perpendicular to the flow would be scaled by the mapping.

It seems that more accurate techniques for compensating texture distortions have not yet been applied for surface LIC. However, such techniques are not too hard to design. In the following we describe a simple algorithm for generating input noise with the desired properties. The basic idea is to create a noise image by drawing a set of irregularly distributed ellipses on top of a black background. The axes of the ellipses are chosen such that circles are obtained if the image is mapped into physical space. This is illustrated in the following figure:



In physical space the radius vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ have unit length. While the first vector $\boldsymbol{a}$ is aligned to one of the coordinate axes, the second orthonormal vector $\boldsymbol{b}$ can be computed by taking the cross product of the surface normal and $\boldsymbol{a}$. Of course, the parameter space components of $\boldsymbol{a}$ and $\boldsymbol{b}$ could be computed via Eq. (5.11). However, instead of evaluating matrix determinants it is easier to compute the required dot products directly:

$$\begin{aligned} a_u &= |\boldsymbol{e}_u|^{-1} \qquad & b_u &= -\frac{\boldsymbol{e}_u \boldsymbol{e}_v}{|\boldsymbol{e}_u|^2} L^{-1} \\ a_v &= 0 & b_v &= L^{-1} \end{aligned} \qquad \text{with } L = |\boldsymbol{e}_v - \frac{\boldsymbol{e}_u \boldsymbol{e}_v}{|\boldsymbol{e}_u|^2} \boldsymbol{e}_u| \tag{5.14}$$

a)

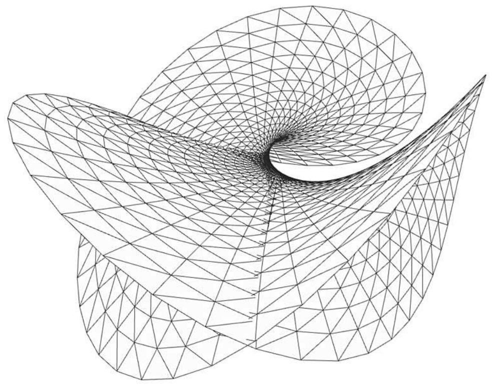
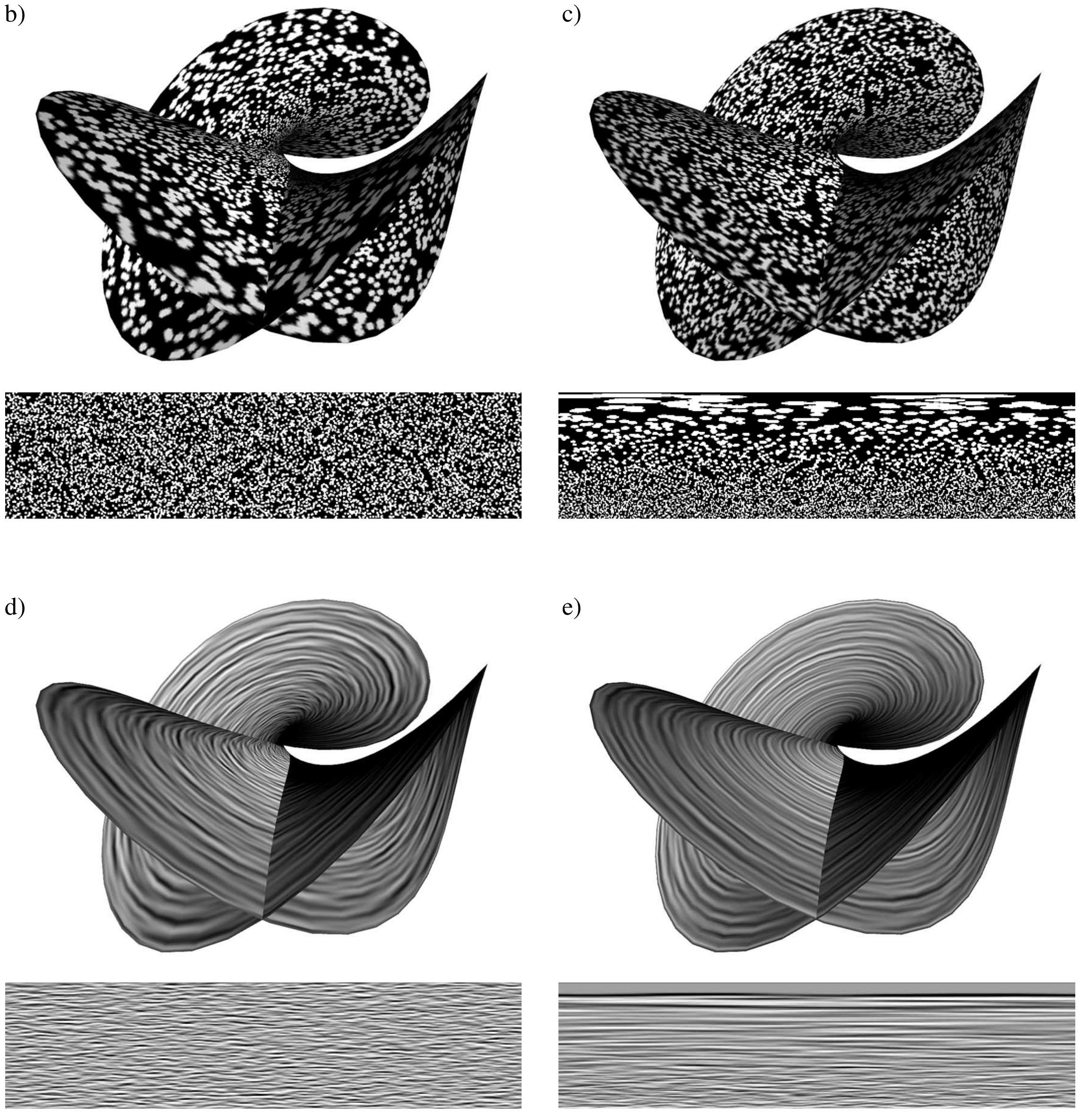


Figure 5.8: LIC images computed in parameter space appear to be distorted when mapped onto a surface (d). These distortions are due to the non-isometric transformation between parameter space and physical space. However, they can be compensated by using suitably chosen input noise and by locally adapting LIC filter lengths (e). The noise image in (c) consists of irregularly distributed spots, which are of equal size when projected onto the surface.



Once the radius vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ have been determined, the corresponding ellipses can be drawn in parameter space. In practice, the ellipses are approximated by simple polygons and a conventional scan conversion algorithm such as [38] is applied. The density of the ellipses should be adapted to the size of a grid cell in physical space, $A = \frac{1}{2} \mid \boldsymbol{e}_u \times \boldsymbol{e}_v \mid$. This can be achieved by generating a pseudo random number of each cell. If this number is smaller than cA an ellipsis is drawn for that cell. Here c is a scaling factor determined by the overall area of the surface and the desired size of the spots in physical space.

Results of the outlined algorithm are shown in Figure 5.8. LIC textures were computed from a constant vector field and are mapped onto a periodic parametric surface (Enneper's surface). The surface mesh is gets highly squeezed arround the center of the model. Nevertheless, a uniform surface LIC texture can be obtained if input noise and filter length are chosen correctly. Variations in texture size are now solely due to the perspective viewing transformation.

5.3 LIC in Physical Space

Computing surface LIC images in parameter space is a simple and elegant approach. Unfortunately, many surfaces are not or at least not easily parametrizable in a global sense. Examples include isosurfaces or other implicite surfaces. For these surfaces usually triangular approximations are computed by means of a marching-cubes-style algorithm [54]. The topology of such triangulations may become arbitrarily complex. Surfaces occuring in CAGD (Computer Aided Geometric Design) often consist of smoothly joined parametric patches, but the combined surface may as well exhibit complex topology and therefore in general can't be parametrized globally. In all these examples the surface LIC technique described in the previous section cannot be applied. Moreover, even in case of parametric surfaces (e.g. stream surfaces) the computation of LIC textures in parameter space might be unfavourable. To maintain a high image quality the resolution of the parameter space LIC texture should be correlated to the biggest cell of the surface mesh in physical space. Therefore, often large parts of a LIC texture get mapped into small regions of the surface. This limits the efficiency of the algorithm.

Alternatively, LIC textures might be computed on a triangulated surface *directly in physical space*. Such an approach can be applied to surfaces of arbitrary topology. In addition, texture distortions are avoided since no non-isometric texture mapping is involved. In the following we first give an overview of physical space surface LIC algorithms. We then consider field line integration on a triangular domain, discuss the definition of input noise, compare the layout of local triangle textures, and finally discuss how the use of texture memory can be optimized.

5.3.1 Overview

Performing surface LIC in physical space implies that field lines have to be traced on a triangular domain in $\mathbb{R}^3$. First of all this means that the integrator need to detect when a field line is leaving the current triangle. If this is the case, the exit point need to be determined and field line integration need to be continued in the opposite triangle. While this task is common to all physical space surface LIC algorithms, methods may differ in respect to the following points, namely

- definition of the tangential vector field $\boldsymbol{f}$,
- choice of points for which LIC intensities are computed,
- method used to render of the final textured surface,
- application of the fast LIC technique, and
- type of input noise being used.

Let us look at these points in more detail. We already pointed out that surface LIC can only be used to visualize the direction of tangential vectors on a 2D manifold instead of arbitrary 3D vectors. Nevertheless, these tangential vectors usually are defined in 3D physical coordinates. However, two situations must be distinguished. Either the vector field may be defined on the surface mesh itself, preferably at the vertices of the triangles. Vectors inside the triangles can then be obtained by interpolation. Alternatively, an independent 3D vector field might be evaluated at every point in a triangle. Sampling a 3D vector field at the vertices of the surface mesh and interpolating the vertex values afterwards usually is inadequate because it impairs the accuracy of visualization. It turns out that the kind of input data influences the choice of local coordinates which should be used for field line integration.

Next, we must decide for which points of a surface LIC intensities should be computed. This question is closely related to the way how the final LIC surface is rendered. For example, Mao et. al. [59] proposed a method based on a ray-casting approach. Given a fixed viewpoint, for each pixel of the output image a ray is shot onto the surface. At the first ray-surface intersection the LIC integral is evaluated. Combined with some additional surface shading, the LIC intensity determines the color of the pixel. Obviously, the main drawback of this method is that only a still image is produced and not a 3D model which can be viewed from arbitrary directions.

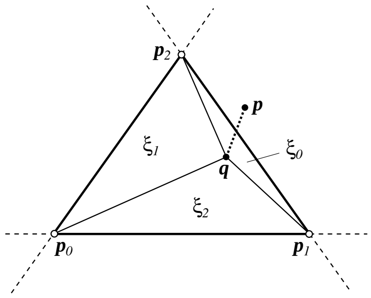
Teitzel et. al. [85] used local textures for all surface triangles. For all pixels of a triangle texture LIC intensities were computed. During rendering the local textures were mapped onto their corresponding triangles. This allows to exploit modern graphics hardware like in case of parameter space methods. In fact, the algorithm proposed in [85] is quite similar to our own one described below and published in a first version in [3]. However, it does not adapt the resolution of local textures to the actual size of a triangle. In addition, it doesn't employ the fast LIC technique developed in Chapter 4.

Finally, different types of input noise can be used when performing LIC in physical space. The noise may either be defined on the surface itself, and, for example, stored in local 2D textures similar to the textures used to hold the final LIC values. Alternatively, a procedural 3D noise function might be used. As will be pointed out below, the latter approach has a number of advantages.

5.3.2 Field Line Integration on Triangular Surfaces

Let us now consider integration of field lines on a triangular domain. We restrict our treatment to piecewise linear C^0 continuous triangular surfaces. This means that all vectors are locally projected into planar triangles. As was pointed out in [85] this results in a discontinuous tangential vector field on M. However, in practice discontinuities rarely cause problems since field line integration is restarted at triangle boundaries anyway. Only if 3D vectors oriented almost perpendicular to the surface are projected into neighbouring triangles, it might happen that projected vectors point into opposite directions at triangle edges. If this is the case, a field line terminates. We don't care about this situation since surface LIC should only be applied to visualize truely tangential vector fields, e.g. flow patterns on stream surfaces. Nevertheless, in principle it is possible to strictly avoid discontinuities. This requires to construct a C^1 continuous surface, e.g. a surface consisting of cubic Bezier triangles [29].

It has already been mentioned that tangential vectors usually are either defined at the vertices of the surface mesh or are obtained by evaluating an independent 3D vector field. In the first case we apply linear interpolation inside the triangles. Although field lines then in principle can be found analytically (c.f. Section 2.2), use of a numerical integrator is faster, more robust, and much simpler. Like in the 2D case in principle any numerical integration scheme can be applied. However, usually field lines quickly run out of the current triangle. Therefore, higher-order integration methods taking large steps usually are not as efficient as in the flat case.



In order to check whether a point on a field line is still contained in the current triangle or not it is convenient to make use of barycentric coordinates. These coordinates have already been introduced in Section 3.2.5. The barycentric coordinates $\boldsymbol{\xi}(\boldsymbol{p})$ of a 2D point $\boldsymbol{p}$ sum up to 1, i.e., $\xi_0 + \xi_1 + \xi_2 = 1$. Whenever at least one of the ξ_i is negative, we know that $\boldsymbol{p}$ is outside the triangle. If this is the case, we need to determine the precise exit position of the field line. Suppose, $\boldsymbol{q}$ is the last point contained in the triangle. Then we are looking for a number s such that $\boldsymbol{q} + s\,(\boldsymbol{p}-\boldsymbol{q})$ lies exactly on a triangle edge. If the i-th component of $\boldsymbol{\xi}(\boldsymbol{p})$ was negative, then the desired value of s can be found by evaluating

$$\xi_i(\boldsymbol{p}) + s\,(\xi_i(\boldsymbol{q}) - \xi_i(\boldsymbol{p})) = 0, \quad \text{or} \quad s = \xi_i(\boldsymbol{p})/(\xi_i(\boldsymbol{p}) - \xi_i(\boldsymbol{q})). \tag{5.15}$$

It may happen that two components of $\boldsymbol{\xi}(\boldsymbol{p})$ are negative. In this case Eq. (5.15) has to be evaluated for both components, yielding two candidate values of s. The smaller of the two corresponds to the actual location of the exit point.

Although the point-in-triangle test can be performed efficiently by means of barycentric coordinates, it is not clear which coordinates should be used for numerical field line integration. There are at least three possible choices, namely 3D Euclidean coordinates $\boldsymbol{x}$, barycentric coordinates $\boldsymbol{\xi}$, as well as 2D Euclidean coordinates $\boldsymbol{u}$. Let us consider these choices in detail.

3D Euclidean coordinates. Although field line integration in a triangle inherently is a 2D problem, note that the 3D Euclidean coordinates $\boldsymbol{x}$ of a point on a field line usually need to be computed anyway. These coordinates are required to evaluate a procedural noise function as well as an independent 3D vector field. In addition, vector values $\boldsymbol{f}$ are commonly specified in 3D Euclidean coordinates, too. If we want to integrate this vector we first have to project it onto the triangle. This can be achieved by $\boldsymbol{f} \leftarrow \boldsymbol{f} - \boldsymbol{n}\boldsymbol{f}$, where $\boldsymbol{n}$ denotes the normal vector of the triangle. The main problem is to determine the barycentric coordinates $\boldsymbol{\xi}$ of a 3D point $\boldsymbol{x}$. This requires to solve the linear system

$$\boldsymbol{x} = \boldsymbol{p}_0 + (\boldsymbol{p}_1 - \boldsymbol{p}_0)\,\xi_1 + (\boldsymbol{p}_2 - \boldsymbol{p}_0)\,\xi_2. \tag{5.16}$$

Since we are dealing with 3D vectors, we have 3 equations and 2 unknowns. Similar to Eq. (5.11) we may compute a solution by means of Cramer's rule, yielding

$$\xi_1 = \frac{\det\left[(\boldsymbol{x} - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}, \tag{5.17}$$

$$\xi_2 = \frac{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{x} - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}. \tag{5.17$'$}$$

The third coordinate is $\xi_0 = 1 - \xi_1 - \xi_2$. Although Eq. (5.17) is not the most efficient way to solve the linear system, it is clear that several arithmetic operations are required in order to obtain a solution. Moreover, updating 3D vectors when integrating in a 2D domain necessarily implies redundant work. Therefore let us investigate alternative approaches.

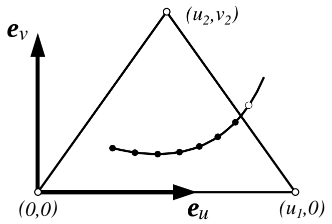
Barycentric coordinates. Since we need to compute barycentric coordinates anyway to perform the point-in-triangle test, it might make sense to use them for field line integration too. Like 3D Euclidean coordinates the vector $\boldsymbol{\xi}$ has three components. However, only two of them are linearly independent. In order to integrate field lines we need to express the vector field $\boldsymbol{f}$ in barycentric coordinates. While barycentric coordinates ξ_i of a position vector $\boldsymbol{p}$ sum up to 1, barycentric coordinates η_i of a directional vector $\boldsymbol{f}$ sum up to 0. This can be proven by writing $\boldsymbol{\eta}(\boldsymbol{f}) = \boldsymbol{\xi}(\boldsymbol{f} + \boldsymbol{p}_0) - \boldsymbol{\xi}(\boldsymbol{p}_0)$. Similar to Eq. (5.17) we now have

$$\eta_1 = \frac{\det\left[\boldsymbol{f}, (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}, \tag{5.18}$$

$$\eta_2 = \frac{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), \boldsymbol{f}, (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}{\det\left[(\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0), (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)\right]}, \tag{5.18$'$}$$

as well as $\eta_0 = -\eta_1 - \eta_2$. Therefore, if a 3D vector field has to be evaluated at every point in a triangle, use of barycentric coordinates is not much cheaper compared to 3D Euclidean coordinates. The integrator merely need to update two components instead of three. On the other hand, if the vector field can be *linearly interpolated* within a triangle, then Eq. (5.18) need to be evaluated only at the vertices of the triangle, but not in every integration step. Instead, the required components η_1 and η_2 can be interpolated from their corresponding vertex values by means of ξ_0, ξ_1, and ξ_2. In fact, in this case barycentric coordinates provide a very efficient choice. For example, a simple Euler integrator just requires the following operations:

determine initial value of $\boldsymbol{\xi}$
while point is in triangle, i.e., all $\xi_i >= 0$
    interpolate vector field, $\boldsymbol{\eta} = \xi_0 \boldsymbol{\eta}(\boldsymbol{p}_0) + \xi_1 \boldsymbol{\eta}(\boldsymbol{p}_1) + \xi_2 \boldsymbol{\eta}(\boldsymbol{p}_2)$
    perform Euler step, $\boldsymbol{\xi} \leftarrow \boldsymbol{\xi} + h\,\boldsymbol{\eta}$
end



2D Euclidean coordinates. Barycentric coordinates aren't orthonormal coordinates. This is the reason why the projection of an arbitrary 3D vector into a triangle was difficult to compute. However, if we use orthonormal coordinates $\boldsymbol{u}$ within a triangle, then the projected vector components are simply given by the dot products $\boldsymbol{f}\boldsymbol{e}_u$ and $\boldsymbol{f}\boldsymbol{e}_v$, where $\boldsymbol{e}_u$ and $\boldsymbol{e}_v$ denote the axes of the 2D Euclidean coordinate system. What remains is to compute the barycentric coordinates $\boldsymbol{\xi}$ of a 2D point $\boldsymbol{u} = (u, v)^T$. However, this can be done via Eq. (3.54). Let u_i and v_i be the 2D Euclidean coordinates of the triangle vertices. By choosing the 2D system so that $u_0 = v_0 = v_1 = 0$ we simply obtain

$$\xi_0 = 1 - \xi_1 - \xi_2, \quad \xi_1 = (u\,v_2 - v\,u_2)/u_1 v_2, \quad \xi_2 = v/v_2. \tag{5.19}$$

The use of 2D Eulidean coordinates requires to store two 3D vectors $\boldsymbol{e}_u$ and $\boldsymbol{e}_v$ for each triangle, as well as the three vertex components u_1, u_2, and v_2. Compared to the overall amount of memory needed by a surface LIC algorithm (which is mainly due to acumulation and hitcount buffer) this is usually only a small fraction. To illustrate the use of 2D Euclidean coordinates again we list the pseudo-code of a simple Euler integrator:

determine initial value of $\boldsymbol{\xi}$ and $\boldsymbol{u}$
while point is in triangle, i.e., all $\xi_i >= 0$
    compute 3D coordinates, $\boldsymbol{x} = \xi_0 \boldsymbol{p}_0 + \xi_1 \boldsymbol{p}_1 + \xi_2 \boldsymbol{p}_2$
    evaluate 3D vector field, $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{x})$
    project vector into triangle, $\boldsymbol{g} = (\boldsymbol{f}\boldsymbol{e}_u, \boldsymbol{f}\boldsymbol{e}_v)^T$
    perform Euler step, $\boldsymbol{u} \leftarrow \boldsymbol{u} + h\,\boldsymbol{g}$
    update barycentric coordinates $\boldsymbol{\xi}$ via Eq. (5.19)
end

5.3.3 Solid Noise

In order to perform surface LIC in physical space at every point on the surface an input noise function need to be defined. Of course it would be possible to define a 2D noise image for every triangle similar to the texture being used to store the final LIC values. At every point inside a triangle the corresponding 2D noise image could be sampled. In fact, this approach was chosen in [85]. Alternatively, a 3D noise function could be used. One advantage of such an approach is that no artifacts due to partially covered pixels can occur at triangle edges. Of course, it is not really feasible to sample a 3D noise function on a discrete mesh like we did in the 2D case. This would require an immense amount

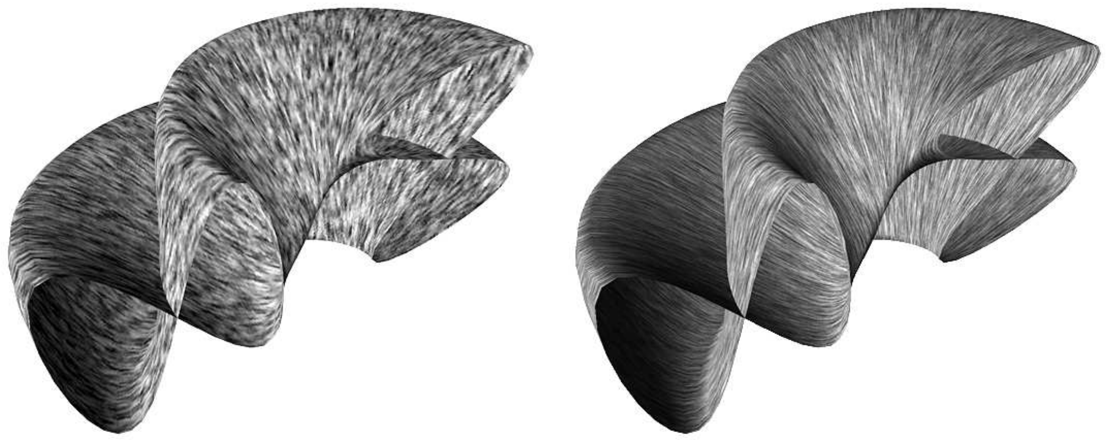


Figure 5.9: The resolution of the input noise determines how much detail is contained in the final surface texture. In both images the LIC filter kernels had the same length in physical space units.

of memory. Instead, a procedural noise function such as the one proposed by Perlin [67] should be used.

An efficient implementation of the Perlin noise function is given in [92]. The method works by mapping floating point coordinates to an integer grid. From the integer indices a pseudo-random value is generated using arithmetic and binary operations. To get a continuous noise function, random values of neighbouring grid points have to be interpolated. However, input noise for LIC need not to be continuous. Therefore the interpolation step can be discarded, and the implementation becomes very simple. In particular, we used the following C-function:

```
int noise(float x, float y, float z) {
    int random;
    x = (int) noise_scale*x;
    y = (int) noise_scale*y;
    z = (int) noise_scale*z;
    float r1 = -x*67.0 + y*59.0 + z*71.0;
    float r2 =  x*73.0 + y*79.0 - z*83.0;
    float r3 =  x*89.0 - y*97.0 + z*101.0;
    random  = *((int*)(&r1))<<12^*((int*)(&r1));
    random ^= *((int*)(&r2))<<12^*((int*)(&r2));
    random ^= *((int*)(&r3))<<12^*((int*)(&r3));
    return (random&0x7fffffff)%GREY_LEVELS;
}
```

An important parameter is the granularity of the input noise, i.e., the resolution of the integer grid of pseudo-random numbers. This parameter, specified by `noise_scale` in the above code fragment, determines how much detail is contained in the final LIC texture. Figure 5.9 compares two surface LIC images obtained by convolving a solid noise of low and high granularity, respectively. In both cases, equal LIC filter lengths (in physical space units) have been used. The optimal choice of noise granularity also depends on how the surface is projected onto the screen. If the surface is to small in screen space, the directional information of the LIC texture cannot be resolved anymore and aliasing effects occur. Moreover, the resolution of the local triangle textures used to store the final LIC intensities should match the resolution of the input noise. If this resolution is to coarse, the input noise cannot be sampled properly. If it is too fine, texture memory would be wasted and computing time would be increased unnecessarily.

5.3.4 Layout of Triangle Textures

In this section we want to consider the layout of local texture maps used to store surface LIC intensities for a single triangle. Modern graphics computers are able to map these textures onto the surface by means of hardware acceleration. Thus, the final surface can be displayed at truely interactive frame rates. In particular, we investigate three different layouts, namely iso-parametric textures, uniform textures, and blended ones. All three approaches are illustrated in Figure 5.10. During rendering texture maps may either be interpolated bilinearly, or point sampling may be used (constant interpolation). Usually, bilinear interpolation yields better results. In Figure 5.10 both interpolation methods are compared.

Iso-parametric Textures. In the most simple case identical texture maps are used for all triangles of the surface. Such an approach has been used by Teitzel et. al. [85]. The size of the triangle textures need to be specified in advance. The three vertices of a triangle are associated with the texture coordinates (0,0), (1,0), and (0,1). This means, that textures appear sheared when mapped onto the surface. In addition, the resolution of a texture in physical space units is not constant. In fact, non-isotropic distortions may occur. Therefore, when applying the fast LIC algorithm it is difficult to find a suitable sampling distance. The optimal step size depends on size and shape of a particular triangle as well as on the direction of the vector field. Another problem comprises artifacts at triangle boundaries. These artifacts arise because texture maps of neighbouring triangles don't fit together seamlessly. However, in practice edge artifacts are hardly visible unless the user zooms up the surface model very much.

Uniform Textures. In order to achieve uniform texture resolution in physical space the size of a local texture map need to reflect the size of the corresponding triangle. Moreover, in order to avoid distortions the texture coordinates of the triangle vertices need to be chosen so that the texture grid remains uniform in physical space. In contrast to iso-parametric textures uniform textures can be used together with the fast LIC algorithm in a straight-forward way. Results are shown in the middle column of Figure 5.10 a, b and c). Note, that there still remain artifacts at boundaries of triangles.

Blended Textures. Although edge artifacts are hardly visible unless a triangle gets very large in screen space, in some situations it might be desirable to display a surface LIC model with no edge artifacts at all. Such a model can be obtained by computing a separate texture for every *vertex patch* of a surface, i.e., for all triangles surrounding a common point. Since every triangle belongs to exactly three vertex patches, about three times more texture memory is needed compared to the simple uniform approach. 2D texture

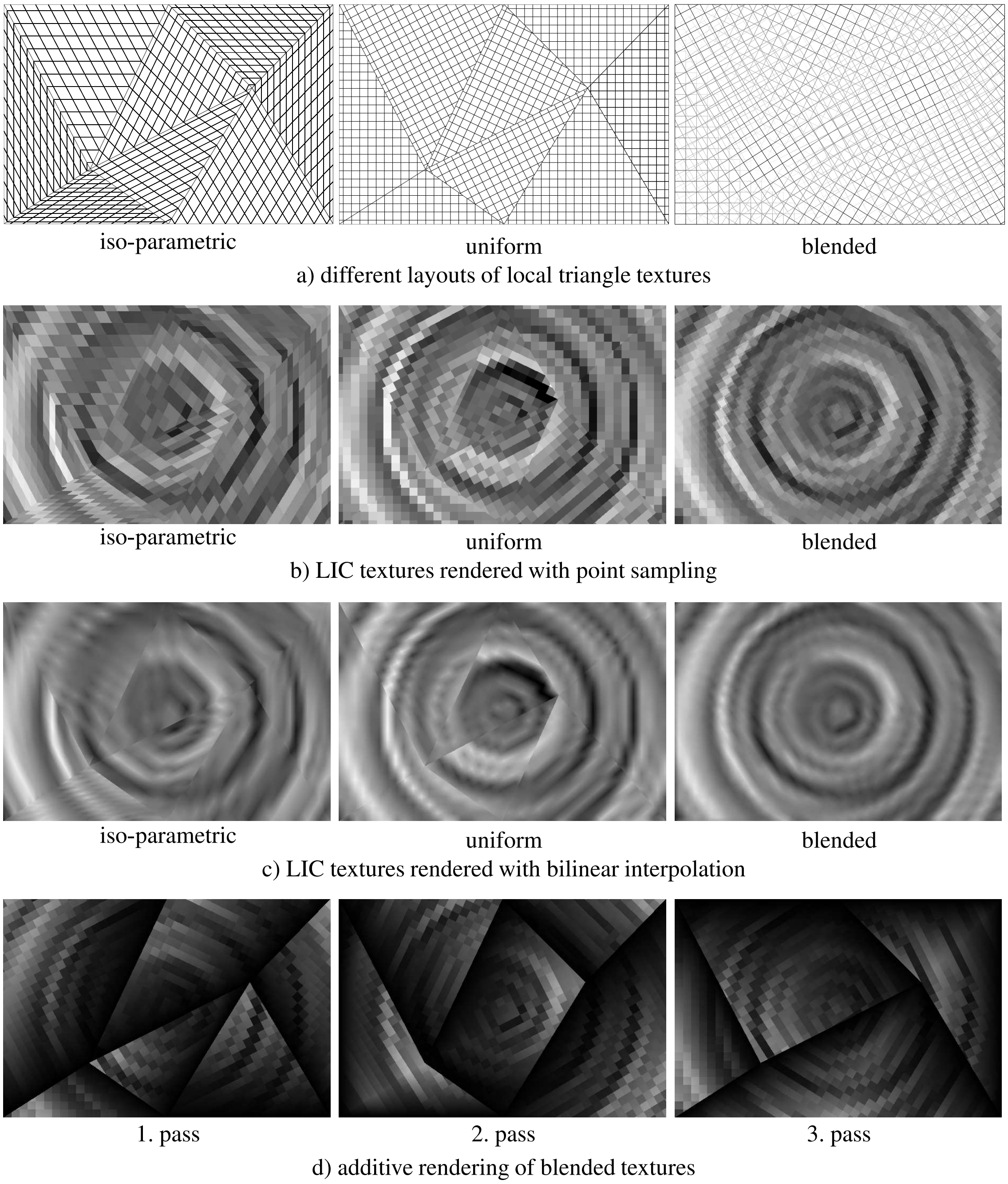


Figure 5.10: This figure illustrates different layouts of local triangle textures for surface LIC in physical space. Iso-parametric textures are most simple to use. However, texture resolution in physical space is not fixed. Piecewise uniform textures provide equal resolution, but there remain artifacts at triangle boundaries. Best results are obtained using blended uniform textures. In order to render the final surface in this case a multipass algorithm has to be applied.

coordinates are obtained by projecting the outer points of a vertex patch into the plane defined by the normal vector of the center point.

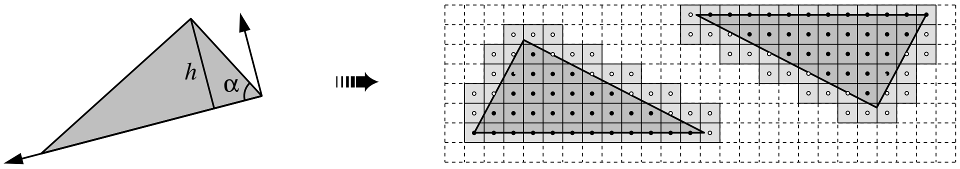
To produce an image of the final surface a three-pass rendering strategy is required. This strategy is illustrated in Figure 5.10 d). In the first pass we render each triangle with the first of its three textures being mapped onto it. However, we modulate texture color so that only the vertex the texture map belongs to gets full intensity. The other two vertices are drawn in black. In OpenGL such a modulation can be easily achieved without changing the texture itself [63]. The remaining two textures of a traingle are rendered in a similar way. However, we now turn on so-called *color blending*. More precisely, we specify a blending function which justs *adds* the color of a fragment to the color already being present in the framebuffer. In this way the final intensity of a pixel is a weighted average of the three triangle textures. Like for barycentric coordinates weights always sum up to one. At a vertex intensity is determined by the correspnding vertex patch texture alone. The other two textures receive zero weight. In the middle of a triangle the mean of all three textures is used. In this way discontinuities at triangle boundaries are avoided. If bilinear interpolation is used, a truely continuous intensity distribution is obtained, c.f. right image of Figure 5.10 c). However, due to the multi-pass algorithm rendering performance is decreased by a factor three.

5.3.5 Triangle Packing

In this section we want to consider an important practical aspect of surface LIC, namely how to organize individual triangle textures in memory. Although modern graphics computers provide fast hardware-supported texture mapping, the total size of the textures is limited. Moreover, width and height of individual textures are restricted to powers of two, i.e., textures have to be $2^i \times 2^j$ pixels large in order to be used with OpenGL.

Obviously, under these constraints it is not a good idea to allocate a separate block of texture memory for every triangle of a surface. A large amount of expensive texture memory would be wasted. In addition using many small textures instead of a few big ones can affect rendering speed significantly. Therefore we would like to place many small triangular LIC textures into few bigger square blocks. The packing should be as dense as possible. Note, that such a packing can be found very easily in case of iso-parametric triangle textures. Since all textures are of equal size, every two of them can be grouped into a rectangular shape, and these rectangles can be arranged in rows and columns within one big block. However, for the two other types of texture layouts described in the previous section a suitable packing is much harder to compute.

For simple uniform textures we developed a simplified heuristic solution strategy in [3]. The method relies on arranging similar triangles in rows. Only a limited set of triangle orientations is considered. The longest side of a triangle is either aligned to the bottom or to the top of a row. If necessary, the triangle is flipped so that the second largest angle lies either in the lower left or in the upper right. In total, there are only two possible orientations:



At the beginning the heighest triangle is put in the lower left corner of a quadratic texture region. Then the algorithm proceeds in a greedy way, i.e., it tries to find a triangle which optimally fits to the end of a row. The criterium is to minimize the wasted area between two successive triangles. The wasted area can be determined at a discrete level, i.e., by counting the number of unused pixels. Alternatively, the “most similar” triangle may be found by comparing a weighted sum of the squared angle and height differences. In this case, the next triangle is the one which minimizes

$$d^2 = c_\alpha(\alpha_0 - \alpha)^2 + c_h(h_0 - h)^2. \tag{5.20}$$

Here α_0 is the angle of the last triangle and h_0 is the height of the row. If one row of texture memory is filled, a new one is started.

This heuristic approach produces quite efficient packings. Typically 85-95% of all pixels in a quadratic texture block are covered. Examples of resulting arrangements are shown in the upper part of Figure 5.13. The tilings belong to the surface LIC models shown in the lower part of the same figure. The efficiency of the packing algorithm could be further improved by filling the empty regions at the front and back of a row. For this it would be necessary to allow $90°$ rotations of the triangles as well.

5.3.6 Outline of the Algorithm

So far we discussed field line integration on triangular domains, definition of 3D input noise, layouts of local triangle textures, as well as efficient packing of these textures. Let us now summarize how a complete fast surface LIC algorithm in physical space looks like.

Once the layout of the triangle textures has been chosen and texture coordinates of the triangle vertices have been determined memory need to be allocated in order to store number of hits and accumulated LIC intensities for every pixel of the triangle textures. Then all pixels have to be traversed once. For this purpose we make use a scan-conversion algorithm [38]. Scan-conversion allows us to quickly determine all pixels the center of which is contained in a triangle. Note, that this operation is performed in texture space. The barycentric coordinates of pixel centers along a scan line can be computed by linear interpolation. If a particular pixel has a hit count smaller than a user-defined threshold a field line is computed starting from the center of that pixel. Like in 2D fast LIC many additional samples are produced on a field line by performing intensity updates. Field lines are traced accross neighbouring triangles as necessary. The pixel a sample has to be added to can be computed by interpolating and quantizing the texture coordinates of a triangle at the sample position. Finally, accumulated LIC intensities have to be normalized according to the actual number of hits per pixel.

Some special treatment is required for pixels being covered partially by a triangle. If the center of these pixels doesn't fall inside the triangle then it might happen that no LIC intensities are computed, i.e., the pixels aren't hit at all. Moreover, if bilinear interpolation is used even LIC intensities for pixels completely outside of a triangle need to be determined. Figure 5.11 illustrates what pixels are actually accessed during bilinear interpolation. Note, that the problem is less severe for blended uniform textures, since in this case a contiguous texture for a complete vertex patch is used. Pixels being covered partially by a vertex patch only receive small weights during rendering. In pratcice we try to determine intensities for boundary pixels by evaluating the LIC textures of neighbouring triangles. If this fails, for example because the pixel center doesn't fall into a direct neighbour triangle, we simply use a constant grey value for that pixel.

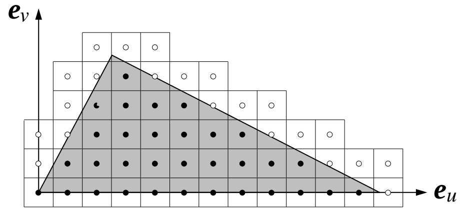


Figure 5.11: Bilinear texture interpolation requires to compute LIC intensities for some pixels outside a triangle, too.

Note, that most of the techniques described for 2D fast LIC can be applied to surface LIC as well. For examples, arbitrary piecewise-polynomial filter kernels can be used, the filter kernels can be animated, or the optimal number of intensity updates can be determined adaptively as described in Section 4.6.2. However, only advanced seed point selection strategies as described in section 4.6.3 are more difficult to implement. In order to quickly find pixels contained in a triangle we had to apply a scan-conversion algorithm. In order to change this scanline oriented traversal scheme all pixels found during scan-conversion could be stored in an intermediate array. Then this array could be traversed for example by means of a Sobol quasi-random iterator. Although such an approach would be possible we actually didn't implement it.

5.3.7 Results and Future Extensions

An example of a surface LIC model computed using the outlined algorithm is shown is Figure 5.12. Although uniform triangle textures have been used, no edge artefacts are noticeble. The surface contains 2011 triangles. Triangle textures have been packed into three square blocks of 512×512 pixels each. The final model can be displayed interatively even on a small SGI O2 workstation. Computation of the complete surface LIC texture took about 12 seconds on a MIPS R10000 195MHz processor. However, the code has not been optimized seriously. We expect that a performance gain of up to a factor of two could still be achieved. Since a 3D vector field was evaluated we used 2D Euclidean coordinates for field line integration. The LIC texture has been computed using a triangle filter with $L = 40$.

More examples of surface LIC images are shown in Figures 1.1 on page 6, Figure 1.3 on page 8, and Figure 2.4 on page 22. In all cases parameters and computing times were not very different from the ones mentioned above. In summary, it turned out that the surface LIC algorithm is very robust, computing times are acceptable, and the resulting surface models are of high quality. Results could be further improved by applying an

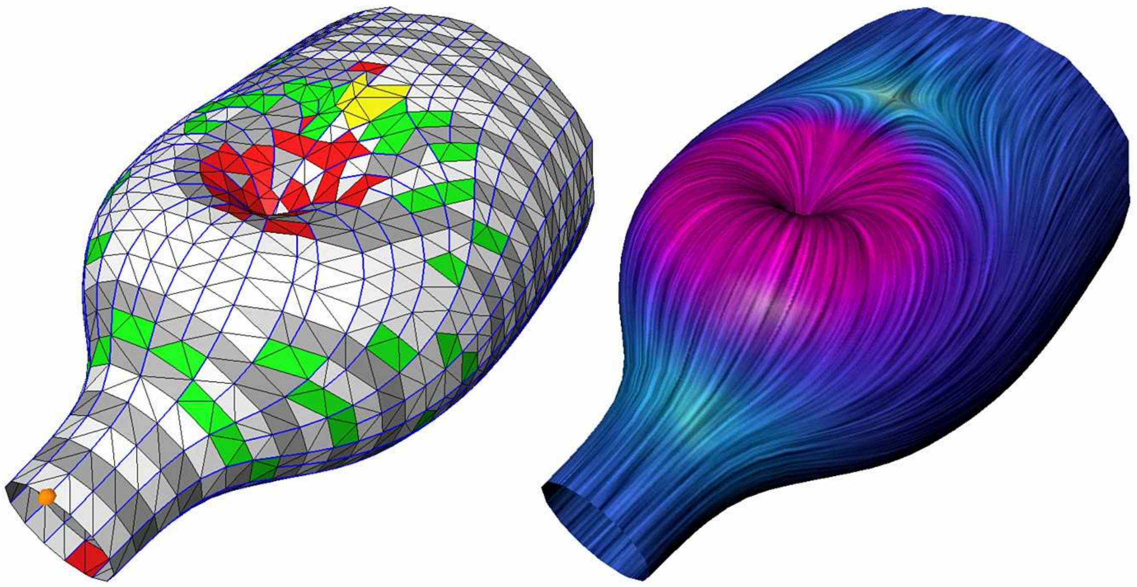


Figure 5.12: Surface LIC texture computed in physical space. The model represents a stream surface in a 3D vector field. In principle a LIC texture could have been computed in parameter space. However, choosing a physical space algorithm ensures that the LIC texture has equal resolution every where on the surface. In a) the underlying surface triangulation is shown.

emboss filter like in the 2D case, c.f. Section 4.3.3. For surfaces, such an approach is known as bump mapping. However, bump mapping depend on the current view and light directions. Therefore it can't be computed in advance together with the LIC texture. Since bump mapping is currently not supported in OpenGL the technique is not amenable for interactive applications. An example of a bump-mapped surface LIC image generated using an external software renderer (Alias/Wavefront) is shown in Figure 5.13.

All texture-based visualization methods we described in this work, the 2D algorithms as well as the ones operating on surfaces, focused on stationary vector fields. Visualization of instationary data remains a challenging problem. Probably instead of stream lines and stream surfaces better streak lines and equivalent streak surfaces should be computed. However, it is not quite clear how LIC textures should be generated on such surfaces so that interframe correlations are preserved. Nevertheless, the algorithms presented in this work provide a sound basis on which further work can be built on.

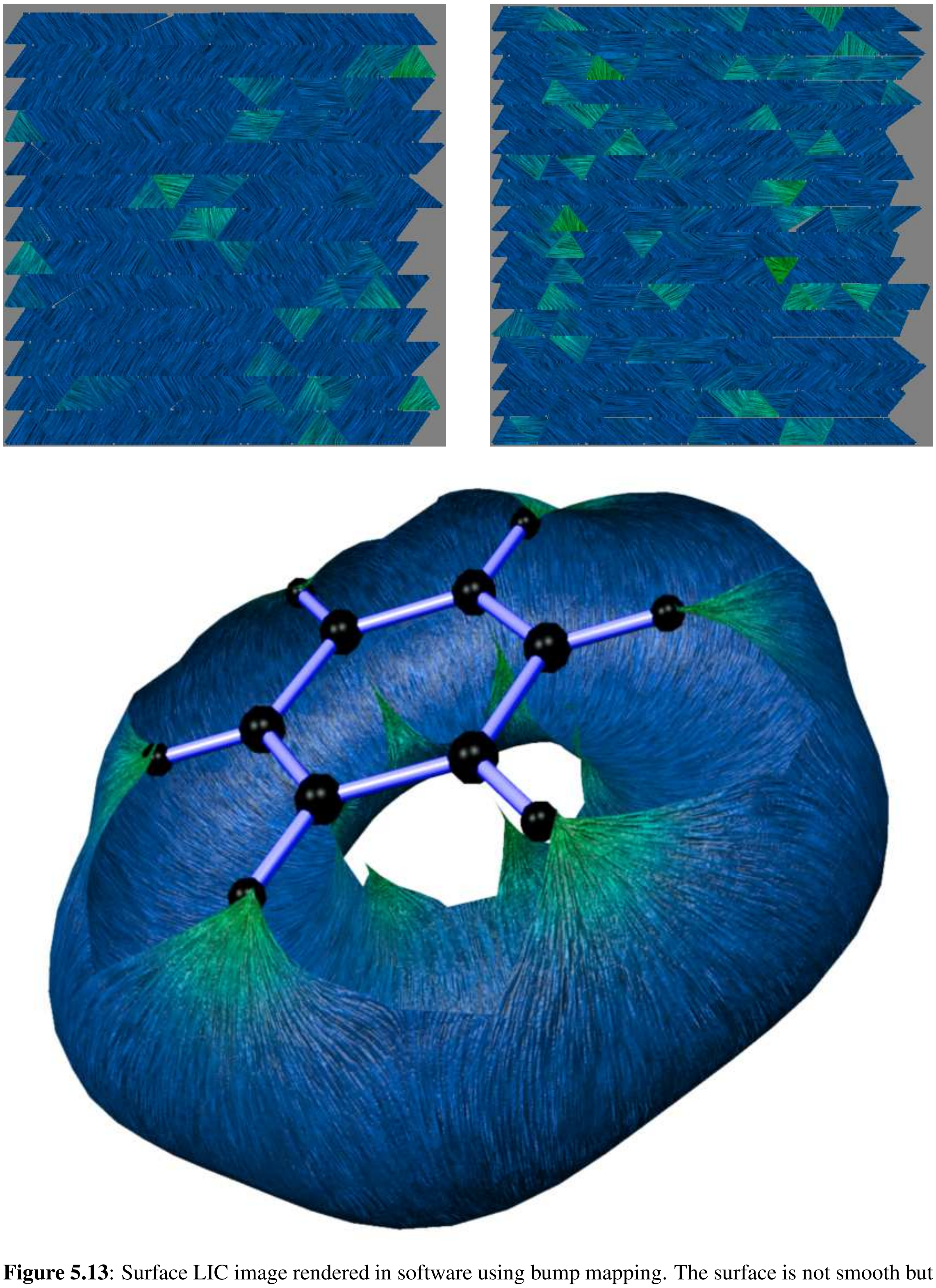


Figure 5.13: Surface LIC image rendered in software using bump mapping. The surface is not smooth but exhibits a certain roughness. This relief structure can help to emphasize directional information. In the upper part it is shown how individual triangle textures are arranged in bigger texture blocks. The surface itself shows the electrostatic field of a benzene molecule.

Chapter 6

Summary and Concluding Remarks

In this work we developed algorithms for visualizing vector fields in $\mathbb{R}^2$ and $\mathbb{R}^3$ using line integral convolution (LIC). The main principal of LIC is to compute a directional texture aligned to the integral curves of a vector field. The method produces highly intuitive images mediating a global understanding of a vector field. This was demonstrated in a number of cases including examples from fluid mechanics, electrical engineering, or applied mathematics. It was shown that LIC is related to particle streak experiments performed in flow visualization. However, LIC allows one to achieve a much larger variety of effects than a real-world experiment. Therefore the method cannot only be applied in scientific visualization but also in areas like image processing or digital art.

Line integral convolution was first conceived by Cabral and Leedom in 1993 [9]. While the original algorithm implied a significant amount of redundant work we developed a new LIC algorithm characterized by significantly improved performance, flexibility, and quality. Performance gains of an order of magnitude could be achieved by exploiting coherence of LIC intensities along the field lines of a vector field. In particular, we developed an efficient update method for computing one-dimensional convolutions using piecewise polynomial filter kernels for a large number of equi-distant samples. In order to compute 2D LIC images field lines were homogeneously distributed and multiple samples were accumulated for each pixel of the output image. We discussed how to choose optimal values for sampling distance, number of intensity updates to be performed, and selection of the field line's seed points. The flexibility of the new algorithm is evident from the ability to choose different resolutions for input texture and output image. This facilitates applications like continuous zooms. We investigated the statistical properties of the resulting LIC images and showed how anti-aliasing can be implemented by increasing the number of samples per pixel. It turned out that a triangle filter is visually superior to a box filter, but that higher-order filter kernels do not produce significantly better results. Finally, a frame blending technique for animating LIC textures was developed. Provided contrast is adjusted correctly this technique allows one to compute a high-quality periodic motion series. Such a series can be used to reveal the directional sign of a vector field as well as its magnitude.

After investigating imaging of 2D vector fields we considered visualization of 3D fields. Instead of pursueing an approach based on direct volume rendering we decided to extract stream surfaces from a 3D vector field. In order to reveal the relevant structures of the field more clearly LIC textures were mapped onto these surfaces. For this purpose we first reviewed algorithms for generating stream surfaces. A method due to Hultquist [44] was improved, so that it could be applied not only to flow fields but also to general vector fields. In particular, the method automatically determines sinks and saddles of a vector field and adapts surface triangulation accordingly. Surface LIC texture can either be computed in the surface's parameter space or in physical space. Parameter space methods require compensation of non-isometric texture-mapping. This can be achieved by means of a special noise generation method. Physical space methods provide greater flexibility and can also be applied if parameter space is highly strechted or if no global parametrization is available at all. In this case field lines need to be traced on a triangular domain. We demostrated how this can be achieved efficiently by using suitable local coordinates. To be able to interactively manipulate surface LIC models on a modern graphics computer texture mapping should be applied. We investigated different layouts of local LIC textures and proposed a three-pass rendering method which can be used to avoid artifacts which otherwise would be visible at triangle edges. We also proposed a heuristic method which can be used to combine multiple triangular textures in bigger square blocks of texture memory. The resulting physical space surface LIC algorithm turned out to be quite robust and effcient.

For all our methods we made use of accurate and well-proved numerical techniques, both for integrating field lines as well as for evaluating vector fields defined on various types of grids. In particular we applied robust Runge-Kutta integrators with error-monitoring and adaptive step size control. An important feature was the existence of smooth polynomial interpolants, which we used for dense output, for straight-line approximations, as well as for processing discontinuous vector fields. For fast evaluation of the interpolation polynomials we applied an efficient forward differencing scheme. The combination of accurate numerical methods, an intuitive visualization approach and an efficient algorithmic implementation made LIC a valueable tool for scientific visualization.

Bibliography

[1] G. V. Bancroft, F. J. Merritt, T. C. Plessel P. G. Kelaita, R. K. McCabe, and A. Globus. FAST: A multiprocessed environment for visualization of computational fluid dynamics. In *Proc. of Visualization 90*, pages 14–27, 1990.

[2] David C. Banks and Bart A. Singer. A Predictor-Corrector Technique for Visualizing Unsteady Flow. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):151–163, June 1995.

[3] Henrik Battke, Detlev Stalling, and Hans-Christian Hege. Fast line integral convolution for arbitrary surfaces in 3d. In Hans-Christian Hege and Konrad Polthier, editors, *Visualization and Mathematics*, pages 181–195. Springer-Verlag, Heidelberg, 1997.

[4] Becker, Barry G., David A. Lane, and Nelson L. Max. Unsteady flow volumes. In *IEEE Visualization '95*, 1995.

[5] P. Bogacki and L. F. Shampine. A 3(2) pair of runge-kutta formulas. *Appl. Math. Lett.*, 2(4):331–325, 1989.

[6] M. Brill, H. Hagen, H.-C. Rodrian, W. Djatschin, and S. V. Klimenko. Streamball techniques for flow vizualization. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of the Conference on Visualization*, pages 225–231, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[7] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Harri Deutsch, Thun, 20 edition, 1979.

[8] Steve Bryson and Creon Levitt. The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In *Visualization '91*, pages 17–24, 1991.

[9] Brian Cabral and Leith C. Leedom. Imaging vector fields using line integral convolution. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 263–272, August 1993.

[10] Brian Cabral and Leith C. Leedom. Highly parallel vector visualization using line integral convolution. In *Seventh SIAM Conference on Parallel Processing for Scientifc Computing*, pages 802–807, Feb 1995.

[11] Wenli Cai and Pheng-Ann Heng. Principal stream surfaces. In *IEEE Visualization '97*, October 1997.

[12] 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.

[13] A.J. Chorin and J.E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer Verlag, New York, 1979.

[14] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, Jan 1986.

[15] Madeleine Coutanceau and Jean-Rene Defaye. Circular cylinder wake configurations: A flow visualization survey. *Appl. Mech. Rev.*, 44(6):255–305, 1991.

[16] R. A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 261–267, San Jose, CA, October 1993. IEEE Computer Society Press.

[17] Dave Darmofal and Robert Haimes. Visualization of 3-d vector fields: Variations on a stream. In *AIAA Paper 92-0074*, Reno, Nevada, Jan 1992.

[18] Carl de Boor. *A Practical Guide to Splines*. Springer Verlag, New York, 1978.

[19] W. C. de Leeuw and J. J. van Wijk. A probe for local flow field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 39–45, San Jose, CA, October 1993. IEEE Computer Society Press.

[20] Willem C. de Leeuw and Jarke J. van Wijk. Enhanced spot noise for vector field visualization. In *IEEE Visualization '95*, pages 233–239, 1995.

[21] Peter Deuflhard and Folkmar Bornemann. *Numerische Mathematik II: Integration gewöhnlicher Differentialgleichungen*. Verlag de Gruyter, Berlin, New York, 1994.

[22] Peter Deuflhard and Andreas Hohmann. *Numerical Analysis. A First Course in Scientific Computation*. Verlag de Gruyter, Berlin, New York, 1995.

[23] Peter Deuflhard, Martin Seebass, Detlev Stalling, Rudolf Beck, and Hans-Christian Hege. Hyperthermia treatment planning in clinical cancer therapy: Modelling, simulation, and visualization. In Achim Sydow, editor, *Computational Physics, Chemistry and Biology*, volume 3, pages 9–17. Wissenschaft und Technik Verlag, 1997.

[24] J. R. Dormand and P. J. Prince. Higher order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7:67–75, 1981.

[25] Don Dovey. Vector plots for irregular grids. In *Visualization '95*, pages 248–253. IEEE Computer Society, 1995.

[26] Milton Van Dycke. *An Album of Fluid Motion*. The Parabolic Press, Stanford, California, 1982.

[27] W. H. Enright, K. R. Jackson, S. P. Nørsett, and P. G. Thomsen. Interpolants for runge-kutta formulas. *ACM Transactions on Mathematical Software*, 12(3):193–218, 1986.

[28] W. H. Enright, K. R. Jackson, S. P. Nørsett, and P. G. Thomsen. Effective solution of discontinuous ivps using a runge-kutta formula pair with interpolants. *Applied Mathematics and Computation*, 27:313–335, 1988.

[29] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press, New York, 1988.

[30] Lisa K. Forssell. Visualizing flow over curvilinear grid surfaces using line integral convolution. In *Visualization '94*, pages 240–247. IEEE Computer Society, 1994.

[31] Lisa K. Forssell and Scott D. Cohen. Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):133–141, June 1995.

[32] William T. Freeman, Edward H. Adelson, and David J. Heeger. Motion without movement. *Computer Graphics*, 25(4):27–30, July 1991.

[33] Thomas Frühauf. Raycasting vector fields. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[34] Allen Van Gelder and Jane Wilhelms. Interactive animated visualization of flow fields. *1992 Workshop on Volume Visualization*, pages 47–54, 1992.

[35] A. Globus, C. Levit, and T. Lasinski. A tool for visualizing the topology of three-dimensional vector fields. In *Visualization '91*, pages 33–40, 1991.

[36] Robert Haimes. pv3: A distrubuted system for large-scale unsteady cfd visualization. In *AIAA Paper 94-0321*, 1994.

[37] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1987.

[38] Paul S. Heckbert. Generic convex polygon scan conversion and clipping. In Andrew S. Glassner, editor, *Graphics Gems*, pages 84–86. Academic Press, 1990.

[39] Hans-Christian Hege, Tobias Höllerer, and Detlev Stalling. Volume rendering: Mathematical models and algorithmic aspects. Technical Report TR-93-07, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), May 1994.

[40] Hans-Christian Hege and Detlev Stalling. Lic: Acceleration, animation, zoom. In *Texture Synthesis with Line Integral Convolution (course notes)*, pages 17–49. ACM SIGGRAPH'97, August 1997.

[41] Hans-Christian Hege and Detlev Stalling. Fast lic with piecewise polynomial filter kernels. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization – Algorithms and Applications*, pages 295–314. Springer-Verlag, Heidelberg, 1998.

[42] J. Helman and L. Hesselink. Representation and display of vector field topology in fluid flow data sets. *Computer*, 22(8):27–36, August 1989.

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

[44] Jeff P. M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Visualization '92*, pages 171–178. IEEE Computer Society, October 1992.

[45] Victoria L. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 109–116. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[46] John David Jackson. *Classical Electrodynamics*. John Wiley and Sons, New York, 2nd edition, 1975.

[47] Bernd Jähne. *Digital Image Processing - Concepts, Algorithms, and Scientific Applications*. Springer Verlag, New York, 1991.

[48] Bruno Jobard and Wilfrid Lefer. The motion map: Efficient computation of steady flow animations. In Roni Yagel and Hans Hagen, editors, *Proceedings of the 8th Annual IEEE Conference on Visualization (VISU-97)*, pages 323–328, Los Alamitos, October19–24 1997. IEEE Computer Society Press.

[49] David N. Kenwright and David A. Lane. Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):120–129, June 1996.

[50] Ming-Hoe Kiu and David C. Banks. Multi-frequency noise for LIC. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[51] W. Kutta. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschr. für Math. u. Phys.*, 46:435–453, 1901.

[52] R. Lefever and G. Nicolis. Chemical instabilities and sustained oscillations. *J. Theor. Biol.*, 30:267–284, 1971.

[53] Helwig Löffelmann, Lukas Mroz, Eduard Gröller, and Werner Purgathofer. Stream arrows: enhancing the use of stream surfaces for the visualization of dynamical systems. *The Visual Computer*, 13(8):359–369, 1997. ISSN 0178-2789.

[54] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3D surface construction algorithm. In M. C. Stone, editor, *SIGGRAPH '87 Conference Proceedings (Anaheim, CA, July 27–31, 1987)*, pages 163–170. Computer Graphics, Volume 21, Number 4, July 1987.

[55] Thomas Loser. Vector and tensor field visualization using textures. Master's thesis, Institut fü Verfahrenstechnik, University of Hannover, Germany, 1997.

[56] Kwan-Liu Ma, M. F. Cohen, and J. S. Painter. Volume seeds: a volume exploration technique. *The Journal of Visualization and Computer Animation*, 2(4):135–140, October–December 1991.

[57] Kwan-Liu Ma and Philip J. Smith. Virtual smoke: An interactive 3d flow visualization technique. In *Visualization '92*, pages 46–53. IEEE Computer Society, October 1992.

[58] R. S. MacLeod, C. R. Johnson, and M. A. Matheson. Visualization tools for computational electrocardiology. In *Visualization in Biomedical Computing*, pages 433–444, 1992.

[59] X. Mao, M. Kikukawa, N. Fujita, and A. Imamiya. Line integral convolution for arbitrary 3d surfaces through solid texturing. In *Proc. Eighth Eurographics Workshop on Visualization in Scientific Computing*, pages (to be published by Springer–Verlag), 1997.

[60] N. Max, B. Becker, and R. Crawfis. Flow volumes for interactive vector field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 19–24, San Jose, CA, October 1993. IEEE Computer Society Press.

[61] N. Max, R. Crawfis, and C. Grant. Visualizing 3D velocity fields near contour surfaces. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of the Conference on Visualization*, pages 248–256, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[62] Paul L. Meyer. *Introductory Probability ans Statistical Applications*. Addison-Wesly, Reading (Massachusetts), 2nd edition, 1970.

[63] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[64] Nancy John Nersessian. Faraday's field concept. In David Gooding and Frank A. J. L. James, editors, *Faraday Rediscovered: Essays on the Live and Work of Michael Faraday*, pages 175–187. Stockton Press, New York, 1985.

[65] Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signals and Systems*. Signal Processing Series. Prentice-Hall, 1983.

[66] Lawrence Perko. *Differential Equations and Dynamical Systems*. Springer Verlag, New York, 2nd edition, 1996.

[67] K. Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[68] T. Poeschl. Detecting surface irregularities using isophotes. *Computer Aided Geometric Design*, 1(2):163–168, 1984.

[69] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[70] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[71] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 313–322, July 1985.

[72] M. Rumpf and R.-T. Happe. Characterizing gobal features of simulation data by selected local icons. In *Virtual Environments and Scientific Visualization'96*, 1996.

[73] C. Runge. Über die numerische auflösung von Differentialgleichungen. *Math. Ann.*, 46:167–178, 1895.

[74] Gerik Scheuermann, Hans Hagen, Heinz Krüger, Martin Menzel, and Alyn Rockwood. Visualization of higher order singularities in vector fields. In *IEEE Visualization '97*, October 1997.

[75] John Schlag. Fast embossing effects on raster image data. In Paul Heckbert, editor, *Graphics Gems IV*, pages 433–437. Academic Press, Boston, 1994.

[76] William Schroeder, C. R. Volpe, and W. E. Lorensen. The stream polygon: A technique for 3D vector field visualization. In *Visualization '91*, pages 126–132, 1991.

[77] Lawrence F. Shampine. Interpolation for runge-kutta methods. *SIAM J. Numer. Anal.*, 22(5):1014–1027, 1985.

[78] Han-Wei Shen, Christopher R. Johnson, and Kwan-Liu Ma. Visualizing vector fields using line integral convolution and dye advection. In *1996 Volume Visualization Symposium*, pages 63–70. IEEE, October 1996. ISBN 0-89791-741-3.

[79] I.M. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[80] Detlev Stalling. Lic on surfaces. In *Texture Synthesis with Line Integral Convolution (course notes)*, pages 51–64. ACM SIGGRAPH'97, August 1997.

[81] Detlev Stalling and Hans-Christian Hege. Fast and resolution independent line integral convolution. In *ACM SIGGRAPH Computer Graphics Proceedings*, Annual Conference Series, pages 249–256, August 1995.

[82] Detlev Stalling and Thomas Steinke. Visualization of vector fields in quantum chemistry. Preprint SC-96-01, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), December 1996.

[83] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast Display of Illuminated Field Lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, April 1997.

[84] D. Sujudi and R. Haimes. Identification of swirling flow in 3d vector fields. In *AIAA Paper 95-1715*, 1995.

[85] C. Teitzel, R. Grosso, and T. Ertl. Line integral convolution on triangulated surfaces. Technical Report 8, Universität Erlangen-Nürnberg, 1996. erscheint in: Proc. WSCG '97.

[86] Holger Theisel. *Vector Field Curvature and Applications*. PhD thesis, Universität Rostock, Fakultät der Ingineurwissenschaften, Nov 1995.

[87] Greg Turk and David Banks. Image-guided streamline placement. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 453–460. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[88] Samuel P. Uselton. Volume rendering for computational fluid dynamics: Initial results. Technical Report RNR-91-026, NAS Applied Research Branch (RNR), September 91.

[89] J. J. van Wijk. Implicit stream surfaces. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 245–252, San Jose, CA, October 1993. IEEE Computer Society Press.

[90] Jarke J. van Wijk. Spot noise-texture synthesis for data visualization. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 309–318, July 1991.

[91] Jarke J. van Wijk. Flow visualization with surface particles. *IEEE Computer Graphics and Applications*, 13(4):18–24, July 1993.

[92] G. Ward. A recursive implementation of the perlin noise function. In J. Arvo, editor, *Graphics Gems II*, pages 396–401. Academic Press, Boston, 1991.

[93] Rainer Wegenkittl and Eduard Gröller. Fast oriented line integral convolution for vector field visualization via the Internet. In Roni Yagel and Hans Hagen, editors, *Proceedings of the 8th Annual IEEE Conference on Visualization (VISU-97)*, pages 309–316, Los Alamitos, October19–24 1997. IEEE Computer Society Press.

[94] Chia-Shun Yih. *Fluid Mechanism*. McGraw-Hill, New York, 1969.

[95] O. C. Zienkiewicz. *The Finite Element Method*. McGraw-Hill, London, 1986.

[96] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Parallel line integral convolution. *Parallel Computing*, 23(7):975–989, July 1997.

Zusammenfassung

Die Linienintegral-Faltungsmethode (LIC) ist ein leistungsfähiges Verfahren zur Visualisierung von Vektorfeldern, das auf der Synthese von richtungsabhängigen Texturen beruht. Wie anhand von zahlreichen Beispielen belegt wurde, liefert die Methode intuitive, leicht interpretierbare Ergebnisse, die ein gutes Verständnis der globalen Struktur eines Vektorfeldes vermitteln. Ziel der vorliegenden Arbeit war es, schnelle Algorithmen zur Berechnung von LIC-Bildern zu entwickeln, ein besseres Verständnis für die statistischen Eigenschaften dieser Algorithmen zu gewinnen sowie schließlich Erweiterungen zur Darstellung dreidimensionaler Vektorfelder zu untersuchen.

Die Grundidee für die effiziente Berechnung von LIC-Bildern liegt in der Vermeidung redundanter Rechenschritte. Konkret wurde ein Ansatz entwickelt, mit dem eindimensionale Faltungen mit stückweise polynomialen Filterkernen schnell für viele Stützpunkte gleichzeitig berechnet werden können. Zur Synthese zweidimensionaler Bilder wurden Faltungen entlang vieler, gleichmäßig über das Bild verteilter Integralkurven eines Vektorfeldes berechnet. Das Verfahren wurde unter anderem hinsichtlich der Schrittweite, der Zahl der Stützpunkte und der Verteilung der Integralkurven optimiert. Mit Hilfe geeigneter Erweiterungen konnten animierte Texturen erzeugt werden, die das Richtungsvorzeichen sowie den Betrag eines Vektorfeldes darzustellen erlauben.

Um Einblick in die Struktur dreidimensionaler Vektorfelder zu gewinnen, wurden zunächt Algorithmen zu Berechnung von Stromflächen in solchen Feldern untersucht. Auf diesen Flächen wurden dann richtungsabhängige LIC-Texturen berechnet. Dabei kann die Textursynthese einerseits im Parameterraum der Fläche erfolgen, was jedoch spezielle Vorkehrungen zur Vermeidung von Verzerrungen erforderlich macht. Andererseits wurden Methoden zur Berechnung von LIC in Weltkoordinaten vorgestellt. Es wurde unter anderem gezeigt, wie Integralkurven effizient auf triangulierten Flächen berechnet werden können und wie LIC-Texturen für einzelne Dreiecke gespeichert und artefaktfrei auf ein Flächenmodell projeziert werden können.

Allen Verfahren liegen robuste und zuverlässige numerische Algorithmen zugrunde. Dies betrifft sowohl die Auswertung und Interpolation von Vektorfeldern auf verschiedenen Gittertypen, wie auch die Integration von Integralkurven. Für letzteres wurden Runge-Kutta-Verfahren mit geeigneten Interpolationspolynomen herangezogen. Dadurch konnten Integralkurven effizient an vielen Punkten ausgewertet werden und die Integration unstetiger Felder verbessert werden. Die Interpolationspolynome selbst wurden mit einem schnellen Differenzenverfahren ausgewertet. Die Verwendung dieser Techniken ermöglichte es, LIC zu einem schnellen und zuverlässigen Werkzeug zur Visualisierung von Vektorfeldern zu machen.