University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Multidimensional Data Visualization

State of the Art and Concept of Doctoral Thesis

Tomáš Jirka

Technical Report No. DCSE/TR-2003-03
March, 2003

Distribution: public

Technical Report No. DCSE/TR-2003-03
March, 2003

Multidimensional Data Visualization

Tomáš Jirka

Abstract

Multidimensional volumetric data are often used for storing information in various fields of science such as physics, astronomy etc. because they best match the character of the underlying phenomena. The information contained in such data is, however, usually very dense and thus difficult to understand without subsidiary tools. Proper visualization is definitely one of the most effective and crucial ones.

Multidimensional data can be sorted according to various criteria. First, it is the domain, over which the data are defined, and which is usually two or three dimensional. Second, it is the dimension of the data values themselves, which is theoretically unlimited and depends on the application. Two or three dimensional vector fields can be encountered most frequently, but fields of quadratic tensors are also quite common. It is, however, necessary to realize, that the character of the data must be taken into account as well. Three dimensional vectors need to be treated in a different way than a set of three scalar values. The third important criterion is, whether the data vary in time. If so, they are usually called time dependent. Otherwise, we speak of time independent data.

Such a variety of kinds of data implies even larger variety of visualization techniques. These may be again divided into categories according to various criteria. Obviously, the type of data to apply the particular technique to is the primary one. Among the secondary criteria e.g. the following aspects may belong; whether the approach focuses on the whole data set or just certain region, whether it visualizes the actual data or some derived quantities (e.g. velocity magnitudes, gradients and other), whether it aims to be "photo-realistic" or not et cetera.

This report aims to bring a summary of existing approaches that deal with multidimensional data visualization and to describe selected methods in detail. It should also introduce our previous work, which focused especially on isosurface extraction and gradient estimation, and present the goals of our future research.

This work was supported by identification of grant or project.

Copies of this report are available on
`http://www.kiv.zcu.cz/publications/`
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Copyright © 2003 University of West Bohemia in Pilsen, Czech Republic

Table of Contents

1 INTRODUCTION

Numerous scientific and industrial branches need to work with multidimensional data in general. Among these, one can find physics, astronomy, medical science, engineering, aerodynamics, archeology, seismology, metallurgy, meteorology, food industry etc. Conversely, one can hardly think of a scientific field, where such need is odd. The term *multidimensional* covers several types of data, the most frequently used being vector fields, tensor fields and multi-scalar datasets. Moreover, data from these subgroups may combine together in one dataset, which is sometimes the case in scientific simulations.

Multidimensional data can be sorted according to various criteria. As the first criterion, one may see the domain, over which the data are defined, and which is usually two or three dimensional. As the second one, we may regard the dimension of the data values themselves, which is theoretically unlimited and depends on the particular application. Two or three dimensional vector fields can be encountered most frequently, but fields of quadratic tensors also appear quite commonly. It is, however, necessary to realize, that the character of the data must be taken into account as well. Three dimensional vectors need to be treated in a different way than a set of three scalar values. The third important criterion is, whether the data vary in time. If so, they are usually called time dependent. Otherwise, we speak of time independent data. A more detailed taxonomy of multidimensional data will be described later in chapter 2.

Such a variety of kinds of data implies even larger variety of visualization techniques. These may be again divided into categories according to various criteria. Since multidimensional datasets usually contain fairly big amount of information, which could hardly be mentally integrated and analyzed without proper visualization, the primary criterion for evaluating individual methods should probably be the quality of the visualization in terms of simplicity, accuracy, speed of response et cetera. Each user, however, compares the outcomes of these methods from a different point of view, prefers different features and accentuates different details. A method, which may be of no use to some users, may be crucial for others. Thus, less subjective classification must be adopted.

Obviously, the type of data, to which a particular technique applies, may represent the primary criterion. Among the secondary ones, the following aspects belong: whether the approach focuses on the whole data set or just certain region, whether it visualizes the actual data or some derived quantities (e.g. velocity magnitudes, gradients and other), whether it aims to be "photo-realistic" or not, et cetera. A brief overviews of selected multidimensional data visualization techniques can be found for example in [52] and [64]. More detailed studies, usually devoted to one specific type of visualization methods, can be found in [38] as well as in [45], for instance.

This work intends to bring an overview of the major approaches and directions in multidimensional data visualization. The main problems and techniques should be discussed in more detail. On the other hand, the description of all the types of multidimensional data and the corresponding visualization methods exceeds the scope of this work, if possible at all.

An introduction to isosurface extraction, on which we have worked, will also be described here. Although, extracting isosurfaces is known as related rather to scalar fields, in combination with the *dimension contraction*, isosurfaces are an important tool

for depicting multidimensional data. Work with isosurfaces has also led us to studying and comparing techniques for computing normal vectors in the vertices of triangle meshes, which are usually used for representing isosurfaces (and surfaces in general). Furthermore, we have moved to gradient estimation, where we concentrated especially on accuracy.

The document structure is as follows. In Chapter 2 we present a classification of the kinds of multidimensional data. Chapters 3, 4 and 5 discuss the major approaches to visualization of vector or flow fields, tensor fields and multi-scalar data respectively. Chapter 6 summarizes our previous work related to this topic and in chapter 7 our future goals are stated.

2 TYPES OF MULTIDIMENSIONAL DATA

As mentioned above, there are numerous types of multidimensional data and even more methods for their visualization. Although we will only discuss the selected ones in this work, we will briefly introduce all the types in a classification based on the dimensionality of the domain, over which they are defined, as well as on the dimensionality of the data values themselves. This classification is borrowed from [5] and it will be divided into two sections. The first being devoted to multi-scalar data and the second to vector and tensor data. In the following text, the term tensor will stand for tensors of second and higher order, although scalars and vectors are tensors (of zeroth and first order respectively) as well. In the tables, the cells with the data types to be discussed later in this work will have links to the corresponding chapters.

2.1 Multi-Scalar Data

The first part of the classification, which describes various kinds of multi-scalar data, is shown in Table 1, where **E**ntities defined over **d**-dimensional domain containing (**n**-) **S**calar data are denoted by $\mathbf{E_d^{(n)S}}$. Furthermore, if the entity state evolves in time, lower index **t** is added.

Label	Application	
$\mathbf{E_1^S}$	math	
$\mathbf{E_2^S}$	meteorology aerodynamics physics and astronomy	
$\mathbf{E_2^{nS}}$	geography physics and astronomy medical science	see chapter 5
$\mathbf{E_3^S}$	physics and astronomy remote scanning chemistry	
$\mathbf{E_3^{nS}}$	physics physical chemistry and biochemistry medical science archeology	see chapter 5
$\mathbf{E_{2;t}^{nS}}$	astrophysics meteorology CFD	see chapter 5

$\mathbf{E_{3;t}}^{\mathbf{nS}}$	astrophysics meteorology CFD oceanography	see chapter 5
$\mathbf{E_m}^{\mathbf{nS}}$	physics - dynamical systems computer science – algorithm illustration	see chapter 5

Table 1: Classification of multi-scalar data

2.2 Vector and Tensor Data

Table 2 is organized the same way as Table 1 with the difference that it lists vector and tensor data instead. The entities thus contain (**n**-) **V**ector or (**i**, **j**, **k**...-) **T**ensor data. Thus, the notation is either $\mathbf{E_d}^{\mathbf{V}}{}_{\mathbf{n}}$ or $\mathbf{E_d}^{\mathbf{T}}{}_{\mathbf{i;j;k;...}}$. Index **t** has the same meaning as in the previous case.

Label	Application	
$\mathbf{E_2}^{\mathbf{V}}{}_{\mathbf{2}}$ $\mathbf{E_{2;t}}^{\mathbf{V}}{}_{\mathbf{2}}$	physics oceanography CFD	see chapter 3
$\mathbf{E_2}^{\mathbf{V}}{}_{\mathbf{3}}$ $\mathbf{E_{2;t}}^{\mathbf{V}}{}_{\mathbf{3}}$	physics meteorology	see chapter 3
$\mathbf{E_3}^{\mathbf{V}}{}_{\mathbf{3}}$ $\mathbf{E_{3;t}}^{\mathbf{V}}{}_{\mathbf{3}}$	physics meteorology aerodynamics CFD	see chapter 3
$\mathbf{E_3}^{\mathbf{T}}{}_{\mathbf{3;3}}$ $\mathbf{E_{3;t}}^{\mathbf{T}}{}_{\mathbf{3;3}}$	fluid dynamics material science (stress etc.)	see chapter 4
$\mathbf{E_3}^{\mathbf{T}}{}_{\mathbf{n;n}}$ $\mathbf{E_{3;t}}^{\mathbf{T}}{}_{\mathbf{n;n}}$	material science (stress etc.)	see chapter 4

Table 2: Classification of vector and tensor data

The visualization methods are, however, not limited by using either scalar or vector data. On the contrary, they can be combined.

3 VECTOR FIELD VISUALIZATION

Vector field allow the scientists to describe a wide range of phenomena. Therefore, this kind of data is often used to store information obtained via scientific simulations, measurements etc. Easy to understand visualization of vector fields thus belongs among important tasks of computer graphics. In general, vector fields characterize the dynamic evolution of arbitrary systems as, for instance, electromagnetic field or some other quantity. Most often, however, vector fields are used to represent fluid flow data. This is also probably the most intuitive notion one can think of, when trying to imagine a vector field. For this reason, the terms *vector field* and *flow field* are sometimes interchanged. In the following paragraphs, the term flow field is also sometimes used, yet meaning vector field in general.

3.1 General Concepts

Easily understandable and yet informatively rich visualization of flow fields is required in a vast variety of application fields as the aerodynamics, turbomachinery, astronomy, automotive industry, design, weather simulation and meteorology, climate modeling, ground water flow, etc. Among the users from all these scientific and industrial fields the demands on the output images as well as on the aspects of the visualization process like accuracy, speed of response or storage requirements, differ significantly. Consequently, the spectrum of flow visualization approaches is very rich and covers various features, e.g., 2D vs. 3D solutions, techniques for steady and time-dependent data et cetera.

Since the visual outcomes must be as easily understandable and interpretable for human eyes as possible, many of the computer graphics flow visualization techniques build upon simulating real world experiments. Visualization of, for instance, path lines, which will be explained later in this text, simulates inserting a light-emitting particle into the flow and recording its movement on a photographic plate. Streak lines, on the other hand, resemble continuous injection of dye into the flow from a constant place, thus leaving a colorful trace in the liquid. Finally, time lines correspond to emitting a set of e.g. hydrogen bubbles to the flow at one instant of time but from different locations, usually from a line perpendicular to the flow [39].

3.1.1 Dimensionality – Spatial and Temporal

In flow visualization, available solutions significantly differ with respect to the dimensionality of the given flow data. Techniques, which are intuitive and useful for 2D data, like color coding or arrow plots, sometimes lack similar advantages in 3D.

In addition to the spatial dimensions addressed above, also dimensionality with respect to time is of great importance in flow visualization. First of all, flow data itself incorporates a notion of time – flows are often interpreted as differential data with respect to time, i.e., when integrating the data, paths of moving entities are obtained. Additionally, the flow itself can change over time (like in turbulent flows, for example), resulting in time-dependent or unsteady data. In this case, two dimensions of time are present and the visualization must carefully distinguish between both in order to prevent the user from being confused. This is especially true, when animation should be used

for flow visualization. Then, even a third temporal dimension can show up in visualization, requiring special care to avoid confusion along with interpretation of the animations.

3.1.2 Flow Data Definition

An inherent characteristic of flow data is that derivative information is given with respect to time, which is provided across an n-dimensional domain $W \subset R^n$, for example, for representing 3D fluid flow (n = 3). In the case of multidimensional flow data (n> 1), temporal derivatives **v** of nD locations **p** within the flow domain W are n-dimensional vectors:

$$\mathbf{v}(\mathbf{p}) = d\mathbf{p} / dt, \quad \mathbf{p} \in \mathbf{O} \subseteq R^n, \mathbf{v} \in R^n, t \in R \tag{1}$$

In analytic models (like dynamical systems – see [30]), vectors **v** are often described as functions of the respective spatial locations **p**, say like **v** = **Ap** for steady linear flow data if **A** is a constant n×n-matrix. A general formulation of (possibly unsteady, i.e., time-dependent) flow data **v** would be

$$\mathbf{v}(\mathbf{p},t) : \mathbf{O} \times \mathbf{?} \rightarrow R^n, \quad \mathbf{p} \in \mathbf{O} \subset R^n, t \in \mathbf{?} \subset R \tag{2}$$

where **p** represents the location of the flow and t represents the system time. If t is considered to be constant, i.e., for steady flow data, the more simple case of $\mathbf{v}(\mathbf{p}) : \Omega \rightarrow R_n$ is given.

3.1.3 Discrete vs. Analytical

In cases of results from flow simulation, like in automotive applications or airplane design, vector data **v** are usually not given in analytic form, but need to be reconstructed from the (discrete) simulation output. As usually, numerical methods are used to actually do the flow simulation such as finite element methods, mostly producing a large-sized grid of many sample vectors $\mathbf{v}_{i;t}$, which discretely represent the solution of the simulation process at time steps t_i. The terms $\mathbf{p} \in \Omega$ and $t \in \Pi$ in (2) should be replaced by $\mathbf{p} \in [\mathbf{p}_1,...,\mathbf{p}_n]$, where n is the number of grid vertices, and $t \in [t_1,...,t_n]$, where this time n is the number of time steps, respectively. For further procedure, it is assumed that the flow simulation was based on an at least locally continuous model of the flow, thus allowing for continuous reconstruction of the flow data **v** during visualization. One option for doing such reconstruction would be to apply a reconstruction filter $h : R^n \rightarrow R$ to compute $\mathbf{v}(\mathbf{p},t) = \sum_i h(\mathbf{p} - \mathbf{p}_i) \cdot \mathbf{v}(\mathbf{p}_i, t)$. Filter h usually has only local extent, efficient procedures for finding those flow samples $\mathbf{v}_{i;t}$, which are nearest to the query point **p**, are needed to do proper reconstruction (refer to the Cell search section in chapter 3.3.1). It is necessary to realize that the reconstruction includes both, spatial as well as temporal, characters (see sections Spatial velocity interpolation and Temporal velocity interpolation in subchapter 3.3.1).

3.1.4 Grids

As already mentioned, applications producing vector data usually present them as a field of vector quantities aligned to grid vertices. There are of course many types of grids with different properties. Among the most frequently exploited types, one can find regular Cartesian and curvilinear grids, regular or irregular tetrahedral grids, multi-

zoned, moving or complex unstructured grids etc. Grid type is an important property of each vector field data set as it significantly influences the effectiveness of some algorithms. An example would be the *cell search* or *point location* algorithm (described in 3.3.1), which heavily depends on the type of the used grid. Cell search is trivial and very fast in Cartesian grids, more complicated and less effective in curvilinear grids and quite ineffective in unstructured grids. Another example could be the algorithm for *vector field reconstruction*, which is used for approximating vector quantities in locations other than the grid vertices and in temporal instants between two consecutive time steps. The theory of grids is, however, too broad to discuss it here in detail.

The approach of attaching the data values to fixed locations is sometimes called *Eulerian* [39]. The opposite is *Lagrangian* approach, which links the physical quantities to small particles moving through the area with the flow and given as a function of starting position and time. This attitude is, however, less frequently exploited.

3.1.5 Main Approaches

The large spectrum of users' needs has led to a development of numerous approaches for vector field visualization, which can be sorted into four main categories, the first being *direct* flow visualization. Methods from this category use an as direct as possible translation of the flow data into visualization cues, such as by drawing arrows. Flow visualization solutions of this kind allow immediate investigation of the vector data, without a lot of mental translation effort. For a better illustration of the long-term behavior induced by flow dynamics, *integration based* approaches first integrate the flow data and use the resulting integral objects as a basis for visualization. Displaying streamlines is a good example of integration based techniques. Another approach for visualizing flow data is the *feature based* approach, in which an abstraction step is performed first. From the original data set, interesting objects are extracted, such as important phenomena or topological information of the flow. These flow features represent an abstraction of the data, and can be visualized efficiently and without the presence of the original data, thus achieving a huge data reduction, which makes this approach very suitable for large (time-dependent) data sets, originating from computational fluid dynamics simulations. These data sets are simply too large to visualize directly, and therefore, a lot of time is required in preprocessing, for computing the features (feature extraction). But once this preprocessing has been performed, visualization can be done very quickly. The general idea behind the fourth, and in our classification the last, group of methods consists in *deriving scalar quantities* from the vector data first and then visualizing them via approaches like isosurface extraction or direct volume visualization. Each of the four categories will be described in a separate section bellow.

Now that we have outlined one of the possible classifications, which sorts the flow data visualization techniques into four main categories, it is desirable to briefly note that in practical applications, hybrid methods combining techniques from different categories are usually implemented. As mentioned above, the main task of vector data visualization is to communicate as large amount of information as possible to the user, yet keeping it intuitively and easily understandable. Combining various visualization techniques is obviously a good tool to fulfill such task. Thus, one can come across color coded line integral convolution applied on an isosurface. That is, a hybrid of the direct, integration-based and derived value visualization.

3.2 Direct Visualization

Direct, or global, flow visualization techniques attempt to present the complete data set, or a large subset of it, at a low level of abstraction. The mapping of the data to a visual representation is direct, without complex conversion or extraction steps. These techniques are perhaps the most intuitive visualization strategies as they present the data in a straight way. They can be best utilized for 2D vector fields.

Dealing with 3D flow data naturally brings additional challenges such as 3D rendering. A transition step between 2D flow visualization and the visualization of truly 3D flow data is the restriction to sub dimensional parts of the 3D domain, e.g., sectional slices or boundary surfaces. Thereby, techniques known from 2D flow visualization usually are applicable without major changes. When working with sectional slices, the treatment of flow components orthogonal to slices requires some special care.

For real 3D vector field visualization, rendering becomes the most critical issue. Occlusion and complexity make it difficult to get an immediate overview of an entire flow data set.

3.2.1 Color coding

A common direct flow visualization technique is to map flow attributes such as velocity, pressure, or temperature to color resulting in very intuitive depictions. Of course, the color scale, which is used for mapping, must be chosen carefully with respect to perceptual differentiation. Color coding for 2D flow fields extends to time-dependent data very well, resulting in moving color plots according to changes of the flow properties over time.

Color coding is very effective for visualizing boundary flows or sectional subsets of 3D flow data. In [48] color coding of scalars on 2D slices in 3D automotive simulation data was used and an interactive slicing probe was introduced, which maps the vector field data to hue. The use of scalar clipping, i.e., the transparent rendering of slice regions where the corresponding data value does not lie within a specific data range, allows to use multiple (colored) slices with reduced problems due to occlusion.

For color coding in 3D, volume rendering is necessary to deal with occlusion. This topic is, however, too broad. Therefore, we refer the user to the volume rendering status quo report [12]. In this paragraph, we will only outline some additional issues of volume rendering of vector fields when compared to the well-known volume rendering applied on medical volumetric data. These challenges are briefly addressed here [38]: (1) flow data sets are often significantly smoother than medical data – an absence of sharp and clear “object” boundaries (like organ boundaries) makes mapping to opacities more difficult and less intuitive. (2) flow data are often given on non-Cartesian grids, e.g., on curvilinear grids – the complexity of volume rendering gets significantly more tricky on those kinds of grids, starting with nontrivial solutions required for visibility sorting and blending. (3) flow data are also time-dependent in many cases, imposing additional loads on the rendering process.

3.2.2 Arrow plots

A natural vector visualization technique is to map a line, arrow, or glyph to each sample point in the field, oriented according to the flow field. Usually a regular placement of arrows is used in 2D, for example, on an evenly spaced Cartesian grid. Two basic variants of commonly used arrow plots are:

- normalized arrows of unit length visualizing only vector directions as illustrated by Figure 1
- arrows of varying length proportional to the vector magnitudes

This technique is sometimes called hedgehog visualization. In combination with color coding, arrow plots can depict some additional value at the same time. 2D hedgehog plots can be extended to time-dependent data, although bigger time steps might result in jumping arrows, decreasing the quality of such visualization.

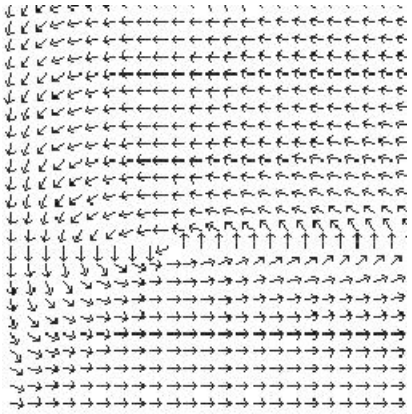


Figure 1: Normalized arrow plot (taken from [30])

Using 2D arrows on slices from 3D flow data may also be telling. When interpreting results of such visualization, however, one must keep in mind that the vector components orthogonal to the slice are usually not depicted. On the other hand, the use of arrows is quite suitable for visualizing flows over boundary surfaces, as can be seen in [56]. The problem with orthogonal vector components is suppressed as cross-boundary flows rarely appear.

Arrow plots in 3D suffer from at least two problems:

- the 2D screen projection can distort vectors‘ positions and orientation thus making the image possibly misleading – using 3D representations of arrows (like a cylinder plus a cone) decreases these problems (see Figure 2) [39].
- glyphs occluding one another – careful seeding is required

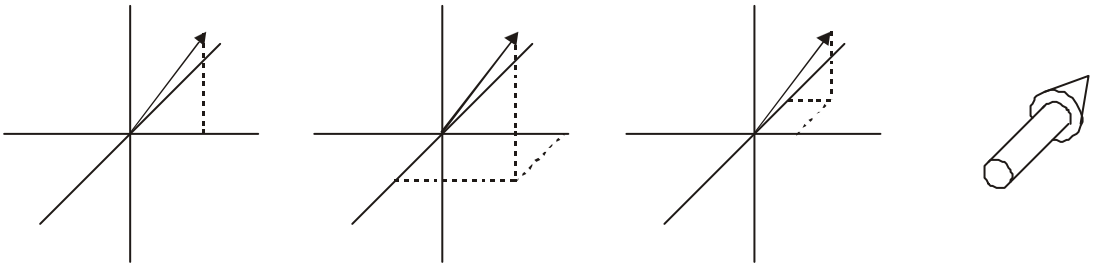


Figure 2: Ambiguity of projecting arrows on a 2D screen on the left; a 3D arrow glyph on the right

To avoid the later, arrow plots are usually based on selective seeding. For instance, one out of a few slices of the 3D field is chosen to form a region from which all the arrows start. Another approach is to highlight those parts of a 3D arrow plot, which point roughly in a user-defined direction as proposed in [2]. Dovey [11] describes a technique of seeding arrows or glyphs in curvilinear and unstructured grids. In order to achieve a uniform density on nonuniformly spaced grids, the paper presents two methods of resampling the data. While the physical space resampling assures the sample points to be well-distributed, an element-based resampling or parameter space resampling can be used to visualize vector fields at arbitrary surfaces within 3D.

3.3 Integration Based Visualization

Unlike the direct visualization methods, techniques to be described now require some sort of computation or extraction prior to the visualization itself. The final image then does not aim to display directly the vector field, it rather shows objects obtained via some computation relying on certain assumptions about the character of the vector field.

As the meaning of the vectors can usually be interpreted as a derivative with respect to some parameter (e.g. derivative of position with respect to time in case of velocity fields), integrating the data over this parameter provides an intuitive notion of how the information contained by the vector field evolves. The following visualization techniques, therefore, fall into the category of integration based methods. We will further divide them into geometric techniques and texture based techniques according to the approach to visualization. Both these groups utilize particle tracing to obtain integral objects and both the groups are closely interconnected.

While particle tracing (see section 3.3.1) concerns the process of integrating through the vector field and gives a recipe to obtain the integral objects, the geometric (3.3.2) and texture-based (3.3.3) methods exploit these objects within the visualization process. They only differ in the way, how they do it. In the former case, individual objects are displayed as they are, while in the later one, the integral lines are convolved with certain texture. When stated like this, both the attitudes might seem to be diametrically different. But on the other hand, from the conceptual point of view, the path leading from one approach to the other is relatively straight. Moving form geometric to texture-based visualization, we just need to apply a dense seeding strategy. In other words, densely seeded geometric objects result in an image similar to that obtained by dense, texture-based techniques. Likewise, moving from texture-based visualization to visualization using geometric objects can be obtained via applying a sparse texture for the convolution.

3.3.1 Particle Tracing – General Issues

As the title implies, these methods simulate real world experiments, when a particle is injected into a flow and its trace within the region is observed. Inserting a set of such imaginary particles into a vector field will result in a set of integral curves, which will more or less characterize the underlying data. Using appropriate temporal and spatial combinations of injecting the particles (i.e. all at once from different locations or just one at the time but from a constant location) will lead to various alternatives of these integral curves, as discussed in the paragraphs bellow. First, however, some general problems, which can be encountered when tracing an imaginary particle through a vector field, must be clarified. We will do so using time dependent vector fields. Steady flow can be considered as a special case for t constant.

Vector Field Integration

The integration through a vector field is a computational challenge, and since it is crucial for all the methods in this chapter, it should be explained at the beginning. The explanation will be carried out on a flow data – an important, intuitive and probably the most frequent case of a vector field.

Integrating flow brings the respective path $\mathbf{p}(s)$ of an imaginary particle traveling through the field. This path would analytically be defined by

$$\mathbf{p}(s) = \mathbf{p}_0 + \int_{t=0}^{s} \mathbf{v}(\mathbf{p}(t), t + t_0)dt \tag{3}$$

where $\mathbf{p_0}$ represents the seed location of the particle and t_0 equals to the time when the particle was seeded. Such a particle trace through a vector field is called *pathline.* There are also other types of integral curves such as *streamlines*, *streaklines*, *timelines* etc., which will be described bellow. The dependency of vector $\mathbf{v}$ on time t in equation (3) implies that in this as well as in the upcoming cases we consider an unsteady flow, where the vector field changes in time.

In addition to the theoretical specification of integral curves, it is important to note, that respective integral equations like equation (3) usually cannot be resolved for the curve function analytically, and thereby numerical integration methods need to be employed. The most simple approach is to use a first-order Euler method to compute an approximation $\mathbf{p_E}$ – one iteration of the curve integration is specified as

$$\mathbf{p}_E(t + \Delta t) = \mathbf{p}(t) + \Delta t \mathbf{v}(\mathbf{p}(t), t) \tag{4}$$

where Δt usually is a very small step in time and $\mathbf{p}(t)$ denotes the location to start this Euler step from. A more accurate but also more costly technique is the second order Runge-Kutta method, which uses the Euler approximation $\mathbf{p}_E$ as a hint to compute a better approximation $\mathbf{p}_{RK2}$ of the integral curve:

$$\mathbf{p}_{RK2}(t + \Delta t) = \mathbf{p}(t) + \Delta t \cdot (\mathbf{v}(\mathbf{p}(t), t) + \mathbf{v}(\mathbf{p}_E(t + \Delta t), t))/2 \tag{5}$$

Higher-order methods like the often used fourth-order Runge-Kutta integrator utilize even more such steps to better approximate the local behavior of the integral curve:

$$\begin{aligned}
\mathbf{a} &= \Delta t \cdot \mathbf{v}(\mathbf{p}(t), t), \\
\mathbf{b} &= \Delta t \cdot \mathbf{v}(\mathbf{p}(t) + \mathbf{a}/2, t + \Delta t/2), \\
\mathbf{c} &= \Delta t \cdot \mathbf{v}(\mathbf{p}(t) + \mathbf{b}/2, t + \Delta t/2), \\
\mathbf{d} &= \Delta t \cdot \mathbf{v}(\mathbf{p}(t) + \mathbf{c}, t + \Delta t), \\
\mathbf{p}_{RK4}(t + \Delta t) &= \mathbf{p}(t) + (\mathbf{a} + 2\mathbf{b} + 2\mathbf{c} + \mathbf{d})/6
\end{aligned} \tag{6}$$

Obviously, the choice of the step size *Dt* is an issue. Too large step will lead to loss of accuracy, small steps will, on the other hand, increase the time required for the computation. This topic will be discussed a little later in this chapter, because other problems must first be resolved to maintain consistency. As [26] and [39] suggest, problems to deal with include especially domain transformations, point location or cell search, step size selection and velocity interpolation.

Domain Transformations

The process of numerical simulation and visualization of fluid flow is typically performed in three different domains, sometimes also referred to as spaces [26]. These are [39]:

- *physical domain P*, where the equations of motion are defined. The domain is discretized, often into a curvilinear, boundary-conforming grid fitting the surface of objects; the flow variables (velocity, density, pressure, etc.) are computed in the grid points of *P*.
- *computational domain C*, to which the equations of motion are transformed. It is descretized to suit the needs of numerical computation, often into a uniform rectangular grid, and thus deformed with respect to *P*.
- *graphical domain G*, which is also often discretized to suit the needs of graphics processing. There is no generally accepted representation of *G*. As visualization often directly refers to physical reality, the shape of objects in *G* is usually the same as in *P*. Often a regular or hierarchical, rectangular discretization is used. *G* is populated with geometric primitives and attributes, such as shapes and colors, which must be ultimately expressed in pixels.

Transitions between these domains [39] must be performed as depicted in Figure 3. Although the grids can be of several types (structured or unstructured, rectilinear or curvilinear etc.), we will focus on structured grids with regular hexahedral topology. Their geometry can be curvilinear, in which case cells resemble warped bricks, or orthogonal, resulting in cubical or rectangular brick shaped cells.

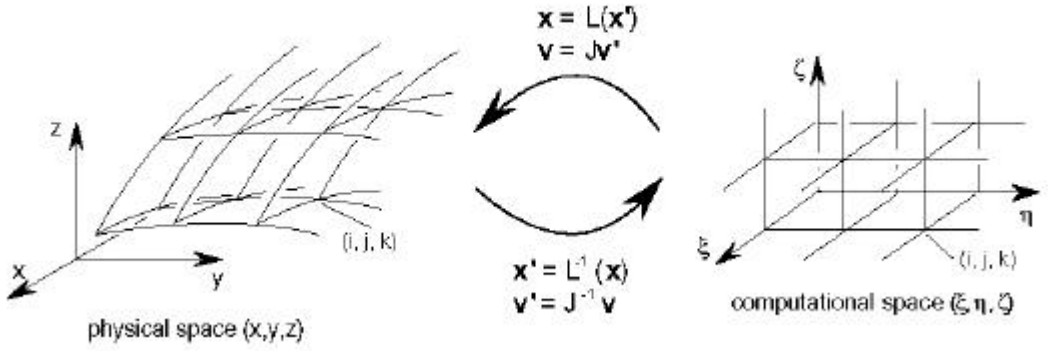


Figure 3: Transformation between the physical and the computational domain (taken from [39])

The discretization in *P* for a flow simulation often lads to a structured curvilinear grid, in each cell of which the grid point nearest to the origin has coordinates (i, j, k), where $i \in \langle 1, n_x \rangle$, $j \in \langle 1, n_y \rangle$, $k \in \langle 1, n_z \rangle$. A general point of *P* is denoted as $\mathbf{x}_p(x, y, z)$. Velocity vectors $\mathbf{v}_p(i, j, k) = (u, v, w)$ are computed at each grid point.

Computational space *C* is usually discretized as a regular orthogonal Cartesian grid with the same cell indices (i, j, k) and points $\mathbf{x}_c(x, h, z)$, where again $i \in \langle 1, n_x \rangle$, $j \in \langle 1, n_y \rangle$, $k \in \langle 1, n_z \rangle$. Velocities at the grid points of *C* are $\mathbf{v}_c(i, j, k) = (u\text{¢}, v\text{¢}, w\text{¢})$. Generally, a single global transformation between *P* and *C* is not known, but for each neighborhood of a grid point (i, j, k) in *P* a local transformation **L**

can be determined. Transformations for other points in P must then be derived by interpolation between grid values.

Matrix **L** specifies the local transformation of a grid point (i, j, k) from C to P as $\mathbf{x}_p = \mathbf{L}(\mathbf{x}_c)$; similarly, $\mathbf{L}^{-1}$ is used to transform a point from P to C (refer to Figure 3). The Jacobian matrix **J** of **L**, defined analytically as $\mathbf{J} = \partial\mathbf{L} / \partial\mathbf{x}_c$, can be used to transform a vector quantity from C to P. For instance, $\mathbf{v}_p = \mathbf{J} \cdot \mathbf{v}_c$. Again, the inverse $\mathbf{J}^{-1}$ is used to transform vector from P to C.

In general, the mappings are only known at discrete points. As a consequence, the Jacobian must be approximated by finite differences. For a grid point (i, j, k) of C, the columns of **J** may, for example, be approximated by:

$$\begin{aligned} \mathbf{Je}_1 &= x_{i+1,\,j,k} - x_{i,\,j,k} \\ \mathbf{Je}_2 &= x_{i,\,j+1,k} - x_{i,\,j,k} \\ \mathbf{Je}_3 &= x_{i,\,j,k+1} - x_{i,\,j,k} \end{aligned} \tag{7}$$

where $\mathbf{x}_{i,j,k}$ are the coordinates of grid point (i, j, k) in C and $\mathbf{e}_i$ the unit vector in the direction of x_i. Another possibility would be to use central differences: $\mathbf{Je}_1 = (\mathbf{x}_{i+1,j,k} - \mathbf{x}_{i-1,j,k}) / 2$; other types of differences can also be used.

As visualization often directly refers to physical reality, G must be undeformed with respect to P. Also, a new discretization is desirable in G, to support the operations in the rendering stage. The transition to another grid usually involves a resampling of the data field and this has several disadvantages. Especially the transition from a boundary-conforming, locally refined curvilinear grid in P to a uniform orthogonal grid in G may lead to a severe waste of storage space or to loss of information, depending on the resolution of the regular grid. In areas where the resolution of the P grid is higher than the G grid, data may be lost, while in low resolution areas of P, oversampling will lead to many identical data points in G. A partial solution is the use in G of a hierarchical grid type of which the resolution can vary locally.

Often, the boundary conformance will be lost, so that object geometry must be represented separately in G. Another important point is the degradation of accuracy as a result of interpolation. Use of higher order interpolation techniques can reduce this problem.

Cell search

During particle integration, it is necessary to determine the grid cell that a particle currently lies in. This requires cell search (also referred to as point location). In computational space, the grid is uniform and the cell in which the particle currently lies can be determined easily. For example, suppose the computational coordinates of the particle's position are (x, h, z), then the particle lies in grid cell (int(x), int(h), int(z)). Although cell search is fast and simple in computational space, there are some disadvantages for tracing in C. Firstly, the velocity needs to be converted from P into C. This requires additional calculation time for the velocity transformation. Secondly, the transforming Jacobian matrices are usually approximations. Therefore, accuracy may be lost during the transformation due to the transformation scheme used. Lastly, if irregularities exist in the grid, then the transformed velocity may be infinite [26]. For these reasons, algorithms for particle tracing performed in physical space were developed as well [21].

As mentioned above, the main reason for tracing in physical space is accuracy. The disadvantage is that cell search is more time consuming in physical space than in computational space. Kenwright and Lane found the time spent in cell search could require more than 25% of the particle tracing time, as it is required whenever the particle moves to a new position. For multi-stage integration methods, such as the fourth order Runge-Kutta described earlier, cell search is required at the intermediate stages of the integration. For example, cell search is needed for $\mathbf{p}(t) + \mathbf{a}/2$, $\mathbf{p}(t) + \mathbf{b}/2$, and $\mathbf{p}(t) + \mathbf{c}$ in equation (6). If the step size Dt is relatively small, then the particle is likely to move within the current cell or jump no more than one cell. Hence, a local cell search can be performed to find the new position. If the grid is multi-zoned, then a global cell search is required when the particle moves to a new block (referred to as grid jumping). Because global cell search is more computationally expensive than local cell search, grid jumping can increase the particle tracing time considerably. Kenwright and Lane [21] managed to improve the speed of particle tracing in physical space by several factors. By decomposing the grid cell into tetrahedra, cell search time was reduced. Furthermore, particle integration, velocity interpolation, and step size control were performed in physical space.

Computational domain cell search

The problem of cell search may be stated the following way. Given $\mathbf{p}$ in the physical domain, the task is to find the corresponding point $\mathbf{c}$ in the computational domain. The first step is to find the grid cell that $\mathbf{p}$ lies in. An intuitive method would be to search for the closest point in the grid using all points. However, this could be expensive if the grid consists of millions of points. Therefore, an algorithm was suggested in [6], which searches along edges of the grid cells to find the closest grid point, and then uses a "stencil walk" approach to find the exact offset of the particle inside the grid cell. The stencil walk approach, which is based on the Newton-Raphson approach, is summarized below [26]:

1. Select the center of the grid cell (i, j, k) as the initial guess of $\mathbf{c}$, where (i, j, k) is the closest grid point to $\mathbf{p}$. Thus, let $\mathbf{c} = (x, h, z)$, where $x = i + 0.5$, $h = j + 0.5$ and $z = k + 0.5$.

2. Convert (x, h, z) to its corresponding physical point $\mathbf{p}(x, h, z)$ using trilinear interpolation (Figure 4):

$$\begin{aligned}\mathbf{p}(x,h,z) = &[(\mathbf{p}_{i,j,k}(1-a) + \mathbf{p}_{i+1,j,k}a)(1-b) + \\ &+ (\mathbf{p}_{i,j+1,k}(1-a) + \mathbf{p}_{i+1,j+1,k}a)b](1-g) + \\ &+ [(\mathbf{p}_{i,j,k+1}(1-a) + \mathbf{p}_{i+1,j,k+1}a)(1-b) + \\ &+ (\mathbf{p}_{i,j+1,k+1}(1-a) + \mathbf{p}_{i+1,j+1,k+1}a)b]g\end{aligned} \tag{8}$$

where $a = x - i$, $b = h - j$ and $g = z - k$.

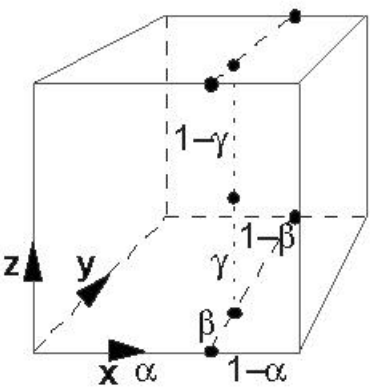


Figure 4: Trilinear interpolation as in [39]

3. Evaluate the difference vector $\Delta\mathbf{p}$, where $\Delta\mathbf{p} = \mathbf{p} - \mathbf{p}(x,h,z)$. This vector indicates, how close $\mathbf{p}(x, h, z)$ is to $\mathbf{p}$ in physical domain.

4. Convert $\Delta\mathbf{p}$ to $\Delta\mathbf{c}$, where $\Delta\mathbf{c}$ is the difference vector mapped into computational domain. Let $\Delta\mathbf{p} = (\Delta x, \Delta y, \Delta z)$ and $\Delta\mathbf{c} = (\Delta a, \Delta b, \Delta g)$. Then

$$\begin{bmatrix}\Delta a\\ \Delta b\\ \Delta g\end{bmatrix} = \mathbf{J}^{-1}\begin{bmatrix}\Delta x\\ \Delta y\\ \Delta z\end{bmatrix}, \text{ where } \mathbf{J} = \begin{bmatrix}x_x & x_h & x_z\\ y_x & y_h & y_z\\ z_x & z_h & z_z\end{bmatrix} \text{ and } \mathbf{J}^{-1} = \begin{bmatrix}x_x & x_y & x_z\\ h_x & h_y & h_z\\ z_x & z_y & z_z\end{bmatrix}. \quad (9)$$

The terms in $\mathbf{J}^{-1}$ are:

$$\begin{aligned} x_x &= (y_h z_z - y_z z_h)/D, \quad x_y = (x_z z_h - x_h z_z)/D, \quad x_z = (x_h y_z - x_z y_h)/D,\\ h_x &= (y_z z_x - y_x z_z)/D, \quad h_y = (x_x z_z - x_z z_x)/D, \quad h_z = (x_z y_x - x_x y_z)/D, \quad (10)\\ z_x &= (y_x z_h - y_h z_x)/D, \quad z_y = (x_h z_x - x_x z_h)/D, \quad z_z = (x_x y_h - x_h y_x)/D, \end{aligned}$$

where D is the determinant of the Jacobian matrix $\mathbf{J}$ and

$$D = x_x y_h z_z + x_z y_x z_h + x_h y_z z_x - x_x y_z z_h - x_h y_x z_z - x_z y_h z_x. \quad (11)$$

The partial derivatives in the Jacobian matrix $\mathbf{J}$ are the partial derivatives of (8), where $\mathbf{p} = (p_x, p_y, p_z)$. For example, $x_x = \partial p_x / \partial x$, $y_x = \partial p_y / \partial x$, $z_x = \partial p_z / \partial x$, etc.

5. Let $a = a + \Delta a$, $b = b + \Delta b$ and $g = g + \Delta g$. If $a, b, g \notin \langle 0,1 \rangle$, then $\mathbf{p}$ is outside the current cell. Increase i by 1 if $a > 1$ or decrease i by 1 if $a < 0$. Update j and k according to b and g respectively, then go to step 1.

6. Let $x = x + \Delta a$, $h = h + \Delta b$ and $z = z + \Delta g$. If $|\Delta c| < e$, where e is the chosen tolerance, then $\mathbf{p}(x, h, z)$ is close enough to $\mathbf{p}$ and its corresponding point $\mathbf{c}(x, h, z)$ in computational space has been found. Otherwise, go to step 2.

The above procedure suits the steady case, for time dependent vector fields, some modifications may need to take place, as the grid may move in time. Particle tracing in moving grids requires additional interpolations. In cell search, to find the current grid

cell containing **p** at time *t*, an interpolated grid cell is generated. The grid cell is simply a linear interpolation of the grid cells at t_l and t_{l+1} if $t_l \le t \le t_{l+1}$. It is not necessary to interpolate the entire grid, only the current grid cell that **p** lies in. Thus, at each intermediate stage of the RK4 integration, an interpolated grid cell is computed for velocity interpolation and cell search.

To transform the velocity from physical space to computational space in unsteady flows with moving grids, equations (9) and (20) (see bellow) need to be modified to consider *t* and the grid velocity (x_r, y_r, z_r). Let

$$\begin{bmatrix} U \\ V \\ W \\ T \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} u \\ v \\ w \\ t \end{bmatrix}, \text{ where } \mathbf{J} = \begin{bmatrix} x_x & x_h & x_z & x_t \\ y_x & y_h & y_z & y_t \\ z_x & z_h & z_z & z_t \\ t_x & t_h & t_z & t_t \end{bmatrix} \text{ and } \mathbf{J}^{-1} = \begin{bmatrix} x_x & x_y & x_z & x_t \\ h_x & h_y & h_z & h_t \\ z_x & z_y & z_z & z_t \\ t_x & t_y & t_z & t_t \end{bmatrix}. \quad (12)$$

Time changes constantly and independently of position and thus the partial derivatives $t_x = t_h = t_z = 0$ and $t_t = t_{l+1} - t_l$ and

$$(x_t, y_t, z_t) = \mathbf{p}^{t_{l+1}}(x,h,z) - \mathbf{p}^{t_l}(x,h,z), \quad (13)$$

where $\mathbf{p}^{t_l}(x,h,z)$ and $\mathbf{p}^{t_{l+1}}(x,h,z)$ are $\mathbf{p}(x,h,z)$ at time t_l and t_{l+1} respectively. The nine metric terms in the upper left corner of $\mathbf{J}^{-1}$ are the same as in (10), and the remaining terms are

$$\begin{aligned} x_t &= (-x_t x_x - y_t x_y - z_t x_z)/t_t D, \\ h_t &= (-x_t h_x - y_t h_y - z_t h_z)/t_t D, \\ z_t &= (-x_t z_x - y_t z_y - z_t z_z)/t_t D, \\ t_x &= t_y = t_z = 0 \quad \text{and} \quad t_t = 1/t_t \end{aligned} \quad (14)$$

where the determinant *D* is given in (11).

Physical domain cell search

As mentioned above, Kenwright and Lane developed an algorithm for cell search in physical space [21]. They employ tetrahedral decomposition for this purpose. This decomposition allows them to quickly find the cell a given point **p** lies in and also its natural coordinates. The natural coordinates, also called barycentric, are local non-dimensional coordinates for a cell.

The trilinear interpolation function (equation (8)) provides the opposite mapping to that required for point location, that is, it determines the coordinates of **p** from a given natural coordinate (x, h, z). Unfortunately, it cannot be inverted easily because of the non-linear products, so it is usually solved numerically using the Newton-Raphson method as described above. Tetrahedral elements, on the other hand allow to use a linear interpolation function to map from natural to physical coordinates:

$$\mathbf{x}(x,h,z) = \mathbf{x}_1 + (\mathbf{x}_2 - \mathbf{x}_1)x + (\mathbf{x}_3 - \mathbf{x}_1)h + (\mathbf{x}_4 - \mathbf{x}_1)z \quad (15)$$

Note that $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$ are the physical coordinates at the vertices of the tetrahedron. The natural coordinates (x, h, z) vary from 0 to 1 in the non-dimensional cell.

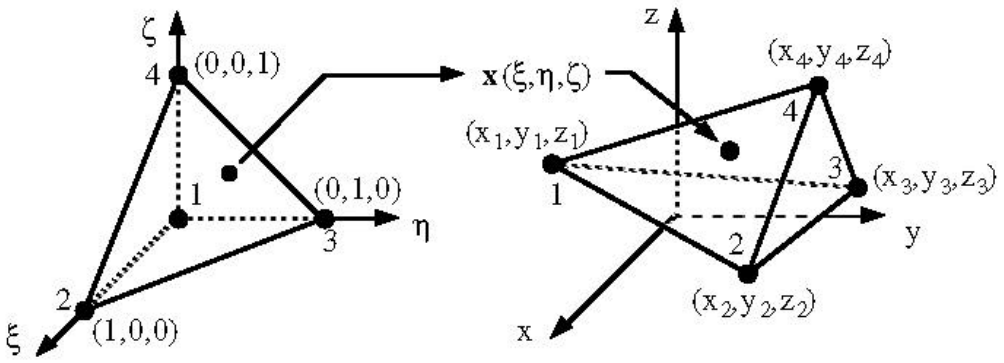


Figure 5: Relation between the natural (left) and physical (right) coordinates

Equation (15) can be inverted analytically because it does not have any non-linear terms thus allowing the natural coordinates to be evaluated directly from the physical coordinates. The solution at a given physical point (x_p, y_p, z_p) is given by:

$$\begin{bmatrix} x \\ h \\ z \end{bmatrix} = \frac{1}{V} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_p - x_1 \\ y_p - y_1 \\ z_p - z_1 \end{bmatrix} \tag{16}$$

The constants in the 3x3 matrix are:

$$\begin{aligned}
a_{11} &= (z_4 - z_1)(y_3 - y_4) - (z_3 - z_4)(y_4 - y_1) \\
a_{21} &= (z_4 - z_1)(y_1 - y_2) - (z_1 - z_2)(y_4 - y_1) \\
a_{31} &= (z_2 - z_3)(y_1 - y_2) - (z_1 - z_2)(y_2 - y_3) \\
a_{12} &= (x_4 - x_1)(z_3 - z_4) - (x_3 - x_4)(z_4 - z_1) \\
a_{22} &= (x_4 - x_1)(z_1 - z_2) - (x_1 - x_2)(z_4 - z_1) \\
a_{32} &= (x_2 - x_3)(z_1 - z_2) - (x_1 - x_2)(z_2 - z_3) \\
a_{13} &= (y_4 - y_1)(x_3 - x_4) - (y_3 - y_4)(x_4 - x_1) \\
a_{23} &= (y_4 - y_1)(x_1 - x_2) - (y_1 - y_2)(x_4 - x_1) \\
a_{33} &= (y_2 - y_3)(x_1 - x_2) - (y_1 - y_2)(x_2 - x_3)
\end{aligned} \tag{17}$$

and the determinant V (actually 6 times the volume of the tetrahedron) is given by:

$$\begin{aligned}
V = {} & (x_2 - x_1)[\,(y_3 - y_1)(z_4 - z_1) - (z_3 - z_1)(y_4 - y_1)\,] + \\
& + (x_3 - x_1)[\,(y_1 - y_2)(z_4 - z_1) - (z_1 - z_2)(y_4 - y_1)\,] + \\
& + (x_4 - x_1)[\,(y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1)\,]
\end{aligned} \tag{18}$$

The natural coordinates (x, h, z) can be evaluated by implementing the equations above. The common terms can be precomputed before evaluating the matrix coefficients and determinant.

The large size of the time-dependent flow data sets makes it impractical to decompose the whole grid. On the contrary, the decomposition is applied on the fly. When a particle enters a hexahedral cell, it is divided into five tetrahedra, which is the minimal possible number. This decomposition is not unique because the diagonal edges alternate across a cell (see Figure 6). Since the faces of a hexahedron are usually non-planar, it is important to ensure that adjoining cell's diagonals match to prevent gaps. This is achieved by alternating between an odd and even decomposition. In a curvilinear

grid, the correct configuration is selected by simply summing up the integer indices (i, j, k) of a specific node (e.g. the node with the lowest indices). Choosing the odd configuration when the sum is odd and the even configuration when the sum is even guarantees continuity between cells.

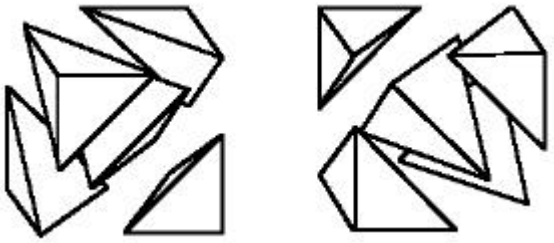


Figure 6: Two ways to decompose a cubic cell into five tetrahedra

Having computed the natural coordinates (x, h, z) in equation (16), there are four conditions, whose validity shows, whether the point $\mathbf{p}(x_p, y_p, z_p)$ lies within the tetrahedron or not. These are:

$$x \geq 0; \quad ? \geq 0; \quad ? \geq 0; \quad 1 - ? - ? - ? \geq 0 \tag{19}$$

If any one of these is invalid then the point is outside the tetrahedron. In particle tracing algorithms, this happens when particles cross cell boundaries. The problem then arises of which tetrahedron to advance to next. The solution is quite simple since the natural coordinates tell you which direction to move. For example, if $x < 0$, the particle would have crossed the $x = 0$ face. Similarly, if $h < 0$ or $z < 0$, the particle would have crossed the $h = 0$ or $z = 0$ face respectively. If the fourth condition is violated, i.e. $(1 - x - h - z) < 0$, then the particle would have crossed the diagonal face. The cell-search proceeds by advancing across the respective face into the adjoining tetrahedron, which can be found quickly using look-up tables.

Occasionally, two or more of these conditions may be violated if a particle crosses near the corner of a cell or if it traverses several cells at once. In such cases, the worst violator of the four conditions is used to predict the next tetrahedron. Even if the bounding tetrahedron is not the immediate neighbor, by always moving in the direction of the worst violator the search will rapidly converge upon the correct tetrahedron.

It is necessary to point out that this approach should only be used when the sought cell lies in close neighborhood of the current one. This condition is hardly ever violated since particles usually do not cross more than one cell at a time. An exception is the very start of tracing a particle and also, in case of multizoned grids, when a particle jumps from one block to another. In such situations global search using a different method should be applied.

The authors of this method claim that the performance of the particle tracing is not degraded on larger data sets, because particle advection only requires local cell searches and interpolations. Multi-zone grids, on the other hand, cause some performance penalty because global searches are required when particles move into new grids.

Step size selection

We will now return to the problem of step size selection, which was mentioned in the chapter devoted to vector field integration, and the explanation of which we postponed for this chapter.

The distance that a particle traverses at each integration step is based on the step size *Dt* (in equations (4), (5) and (6) or other, depending on the integration scheme employed) and the velocity at **p.** The larger *Dt* is, the further **p** traverses. If *Dt* is too large, then the resulting particle trace can be inaccurate because the particle may miss important flow features. This is especially true if the flow changes direction rapidly. Likewise, if *Dt* is too small, then particles may unnecessarily take too many steps to traverse the grid, which would increase the computation time. A good rule for selecting *Dt* is based on the velocity at the current grid cell: if the velocity is large, then *Dt* should be small. Buning [6] suggested letting $\Delta t = c / \max(|U|, |V|, |W|)$ where (*U*, *V*, *W)* represent the computational velocity at **p** and

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{20}$$

Matrix $\mathbf{J}^{-1}$ is given in (9). The computational velocity is used so that the number of steps in each cell is consistent. For example, if $c = 0.2$, then the particle will traverse no more than one-fifth of a computational cell at each step. Small *c* yields small steps. A common scheme for adaptively setting *Dt* is step doubling, which successively reduces *Dt* until some desired accuracy is obtained. As this approach is computationally too expensive [21], Kenwright and Lane suggest another scheme for determining *Dt*, which is based on the curvature of the particle trace. If the curvature is high, then *Dt* should be small.

Spatial velocity interpolation

In particle tracing, the velocity at the current position of the particle is required to advance it further. Velocity interpolation is performed at each stage of the RK4 integration. The velocity at **p** can be interpolated using the velocities at the corners of the grid cell that contains **p**. A fast and simple scheme is trilinear interpolation. If **p** is in grid cell (*i*, *j*, *k*) and **p** has the fractional offsets (*a*, *b*, *c*) from the grid point at (*i*, *j*, *k*), then

$$\begin{aligned} \mathbf{v}(i+a, j+b, k+g) = [&(\mathbf{v}_{i,j,k}(1-a) + \mathbf{v}_{i+1,j,k}a)(1-b) + \\ &+ (\mathbf{v}_{i,j+1,k}(1-a) + \mathbf{v}_{i+1,j+1,k}a)b](1-g) + \\ &+ [(\mathbf{v}_{i,j,k+1}(1-a) + \mathbf{v}_{i+1,j,k+1}a)(1-b) + \\ &+ (\mathbf{v}_{i,j+1,k+1}(1-a) + \mathbf{v}_{i+1,j+1,k+1}a)b]g \end{aligned} \tag{21}$$

where $\mathbf{v}_{i,j,k}$ is the velocity at grid point (*i*, *j*, *k*). The trilinear interpolant assumes that the velocity varies linearly across the edges of the cell. Though trilinear interpolation is simple, accuracy may be lost if the grid cell is deformed. Higher order interpolation would help to reduce this disadvantage, however, at the expense of higher number of velocity vectors needed and higher demands on computational resources.

The computation is even simpler in the case of tetrahedral decomposition described in the previous section. Using the numbering convention in Figure 5, the linear basis function for spatial velocity interpolation is:

$$\mathbf{v}(x,h,z) = \mathbf{v}_1 + (\mathbf{v}_2 - \mathbf{v}_1)x + (\mathbf{v}_3 - \mathbf{v}_1)h + (\mathbf{v}_4 - \mathbf{v}_1)z \qquad (22)$$

where x, h and z are the natural coordinates computed in equation (16) and $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ and $\mathbf{v}_4$ are the velocity vectors at the vertices.

Temporal velocity interpolation

Steady flows do not change in time and we can assume that the number of time steps is infinite. Thus, the velocity function is defined for any t. However, unsteady flows vary in time and the velocity function is only known at time steps $t_1, ..., t_n$. At time t, if $t \neq t_l$ for $l = 1, ...,n$, then a temporal interpolation of velocity is performed prior to the spatial interpolation given in (21). If $t_l \leq t \leq t_{l+1}$, then let

$$\mathbf{v}_{i,j,k}(d) = (1-d)\mathbf{v}_{i,j,k}^{t_l} + d\mathbf{v}_{i,j,k}^{t_{l+1}} \quad \text{and} \quad d = (t-t_l)/(t_{l+1}-t_l), \qquad (23)$$

where $\mathbf{v}_{i,j,k}^{t_l}$ and $\mathbf{v}_{i,j,k}^{t_{l+1}}$ are the velocities at grid point (i, j, k) at time t_l and t_{l+1}, respectively. After the above temporal interpolation of velocity, equation (21) can be evaluated by letting $\mathbf{v}_{i,j,k} = \mathbf{v}_{i,j,k}(d)$. The above interpolation assumes that the flow varies linearly between the time steps and it is only second order accurate in time over a complete RK4 integration.

Overall Particle Tracing Scheme

At this point, we can outline the essential steps in a (time-dependent) particle tracing algorithm as described in [21]. The scheme is general, disregarding the computational versus physical domain problem:

1. Specify the injection point for a particle in physical space (x, y, z, t).
2. Perform a point location to locate the cell that contains the point.
3. Evaluate the cell's velocities and coordinates at time t by interpolating between simulation time steps.
4. Interpolate the velocity field to determine the velocity vector at the current position (x, y, z).
5. Integrate the local velocity field using selected integration scheme (e.g. RK4 described by equation (6)) to determine the particle's new location at time $t + Dt$.
6. Estimate the integration error. Reduce the step size and repeat the integration if the error is too large.
7. Repeat from step 2 until particle leaves flow field or until t exceeds the last simulation time step.

It is important to note that step 5 may involve repeated applications of steps 2, 3 and 4 depending on which numerical integration scheme is used. The 4th order Runge-Kutta scheme used in this study actually requires three repetitions to advance from time t to $t + Dt$.

Basic integral objects

Having explained the problems arising from the need to integrate through a vector field and having presented the overall particle tracing scheme, we will now look at the individual types of integral curves, which can be computed and displayed. The basic integral objects available in 2D as well as 3D include:

- *path line* or *particle trace* – a trajectory, along which a particle travels through the vector field
- *stream line* – a line that is everywhere tangent to the vector field
- *streak line* – arises when a set of particles is being continuously injected into the vector field from a constant location over certain period of time. In [28], some implementation details are given and a comparison to streamlines can be found.
- *time line* – joins the positions of particles released at the same instant in time from different insertion points. These points are usually located on a line perpendicular to the vector field. Once released to the flow, these lines are moved and transformed by the vector field, thus showing the evolution. By calculating the distance between neighboring timelines, one can determine the velocity of the particles in the flow [28].

While stream lines, streak lines and path lines coincide in steady flows, in the case of time varying fields these curves show different trajectories. As Kenwright and Lane [21] claim, streak lines, also called filament lines, are the most popular visualization technique and also the simplest to generate. Stream lines are not generally used to visualize unsteady flows because they do not show the actual motion of particles in the fluid but rather the theoretical trajectories of particles with infinite velocity.

3.3.2 Geometric Visualization of Integral Objects

The previous sections have brought a recipe for obtaining the integral objects from a vector field, outlined problems associated with the computation and presented their known solutions. However, having computed the integral objects, there are also various approaches how to visualize them. These approaches may be sorted into two main categories. Firstly, it is the visualization of integral objects as geometric bodies, which will be examined in this section. Secondly, we talk about texture-based methods whose description will be given in section 3.3.3.

In section 3.3.1, some basic integral objects have already been mentioned. Displaying these objects by the geometric methods will be discussed here in detail and some more advanced geometric objects, commonly used for vector field visualization, will also be added. When using these methods, inconvenient spatial distribution of geometric objects will result in chaotic, cluttered views. Especially in 3D, appropriate seeding strategy constitutes a crucial condition for comprehensive output. As mentioned in the heading paragraphs of the Integration Based Visualization chapter, even the relation between the geometric and texture-based visualization may be expressed in terms of choosing an appropriate seeding strategy. For this reason, seeding strategies will be introduced together with the integral objects being described.

Particle Rendering

Quite a straight forward way is to consider particles, traced by the above methods, to be a kind of geometrical primitive [39]. If a set of such particles is rendered simultaneously, the visualization process is called *particle system*. The most frequent

use of particle systems consists in depicting fuzzy entities with irregular, complex or blur geometry, like grass, tree, fire or smoke. Besides its motion dynamics attributes (like position, speed and motion direction), each particle communicates information by other means as well (shape, size, color, reflectivity and transparency). Individual particles also have their life cycle. They are born, have certain lifetime and die.

The particles' attributes determine the rendering technique. They can be rendered like light emitting particles, which means that during rendering, intensities of particles falling into a single pixel are integrated to obtain its value. If they are, however, rendered as light reflecting objects, shading computations must be performed. For large datasets, the deterministic approach is exchanged for probabilistic attitude, where the particle's position and orientation only act as parameters for computing the probability that the particle is lit directly and the ambient, diffuse and specular components are assigned according to this probability [39].

Streamlets

Streamlets are generated by integrating the flow vectors for a very short time. Even though short, streamlets already communicate temporal evolution along the flow. Figure 7 illustrates an example of inspecting 2D flow field by several streamlets. This technique is easily extendable to 3D, although perceptual problems may arise due to distortions resulting from the rendering projection. Thus, seeding becomes more important in 3D.

Figure 7: The field from Figure 1 visualized by streamlets (also taken from [30])

Löffelmann and Gröller [31] use a thread of streamlets along characteristic structures of 3D flow to gain selective, but importance-based seeding. They employ a probability distribution function assuring the streamlets to be distributed uniformly around a selected base trajectory. The function is designed so that with increasing distance from this trajectory, the distribution of streamlets fades out. As the authors claim, the shape of the field of streamlets will then directly depict flow properties like local convergence / divergence or rotational behavior with respect to the base trajectory. Moreover, the streamlet length represents a perfect mean to intuitively visualize flow velocity.

Streamlines

Performing longer integration, in comparison to streamlets, results in obtaining *streamlines*. These offer an intuitive semantics, because users naturally understand that flows evolve along such integral objects. This statement, however, holds in the case of stable vector fields only. For time-varying data, streak line visualization works better

(see section Streak Lines and their Seeding bellow for details). Concerning the extension to 3D, the same condition stands for streamlines as well as for streamlets – careful seeding is necessary. Otherwise, visual clutter can easily become a problem and the results might be difficult to interpret. The streamline seeding strategies are discussed in a separate section bellow.

To avoid visual chaos and improve perception of various fieldlines in 3D, Zöckler et al. present illuminated streamlines [63][54]. It is necessary to note that this technique could be applied on streaklines and timelines as well. Proper illumination greatly improves spatial perception of complex scenes. However, approximating streamlines (or any other field lines) by cylindrical tubes, which can be illuminated in a straight-forward way using graphics libraries like OpenGL, introduces too many triangles into the scene, thus decreasing the maximum number of streamlines, which can be displayed interactively.

The authors, therefore, present a way of applying Phong-like local illumination model directly to one-dimensional lines, although such an approach is not directly supported in OpenGL. They implement the effect via texture-mapping, utilizing hardware acceleration. For every vertex of a line segment the line's tangent vector is specified as a three-component texture coordinate. Texture coordinates are transformed by a texture transformation matrix before the actual texture lookup is performed. Thus, by initializing this matrix with the current light and view vectors the inner products required for proper illumination can be computed on the fly. By making the streamlines partially transparent, the authors also address the problem of occlusion. Concerning the seeding strategy, they recommend an interactive probe, which can be moved by users according to their immediate needs. Distributing the lines near potential objects of interest was also shown.

Streamline Seeding Strategies

The streamline seeding strategies are of high importance for the informative as well as aesthetic quality of the outcoming image, thus being a subject to an investigation. This topic is, in our opinion, worth a separate section. In the following paragraphs, therefore, these approaches will be categorized and discussed.

First of all, however, the goals that an ideal seeding strategy attempts to reach will be specified together with their priorities.

Goals for Seeding

The aims to achieve in development of the streamline seeding strategy might be put as follows [59]:

- *Coverage*: The streamlines should not miss any interesting regions in the vector field. The interesting regions are those that we would like to study in the vector field, e.g. critical points, separation, and re-attachment lines. In addition, streamlines should cover the entire region of the field. Hence, even if the field is more or less uniform in certain region, some streamlines should indicate the uniform nature of the flow in such area. This goal is considered to be of greatest priority because from a scientific point of view the information content of any visualization is the most important aspect. It is also easier to achieve than the other goals because one can always generate a lot of streamlines such that nothing important is missed. However, simply populating the field with

more streamlines is not acceptable because some areas in the flow field, such as convergent regions, will force streamlines to cluster together, making it difficult to distinguish among individual streamlines. Moreover, it suppresses the characteristic of uniformity as described next.

- *Uniformity*: The streamlines should be more or less uniformly distributed over the field. This is a more challenging goal to achieve because while we can control where to place the seeds, we do not know how the resulting streamlines will behave. Uniformity is directly related to the density of streamlines crossing a unit area of the flow field. Hence, density of streamlines is an important parameter. This goal of a uniform spatial distribution of streamlines is important only to the extent that it does not interfere with the most important goal of achieving a good coverage.
- *Continuity*: It is desirable from the point of view of aesthetics that the streamlines show continuity in the flow. Hence, one would prefer fewer long streamlines over many short streamlines. The latter tend to give the impression of "choppiness" while the former tend to give an impression of smooth continuous flow. In general, given an arbitrary flow field, the longer the streamlines, the higher the likelihood that they will tend to crowd together in some areas and disperse in other areas, thereby making it difficult to meet both the uniformity and continuity criteria simultaneously. Therefore, this parameter needs to be balanced against the uniformity criterion. Although the aim to achieve an aesthetically pleasing visualization has its merits, it should not compromise the other two goals (coverage and uniformity), hence it is lowest on our priority scale.

Grid Aligned Streamline Seeding

Since most flow fields are defined over a grid, a popular seeding strategy is to seed at the grid points. With such approach, no important features are missed but, on the other hand, too many streamlines must be traced. Furthermore, the streamlines tend to clutter unpredictably. Even if the grid is sub-sampled to reduce the density of streamlines, cluttering is still difficult to avoid. Finally, in case of regular grids, such seeding may produce visual artifacts that are not present in the flow field, because the underlying regular structure can be perceived in the output image (Figure 8 left). It is also necessary to point out that regular streamline seeding does not imply regular streamline distribution (Figure 8 right), which also discourages form applying methods focusing on uniform seeding, which have been developed for placing arrow plots and glyphs (e.g. [11] briefly reviewed in section 3.2.2). Depending on the flow field, plain seeding on a regular grid often brings cluttered images, where the individual streamlines can be difficult to distinguish in important regions, for instance, around critical points. The above described requirements of coverage and uniformity may be violated by this approach.

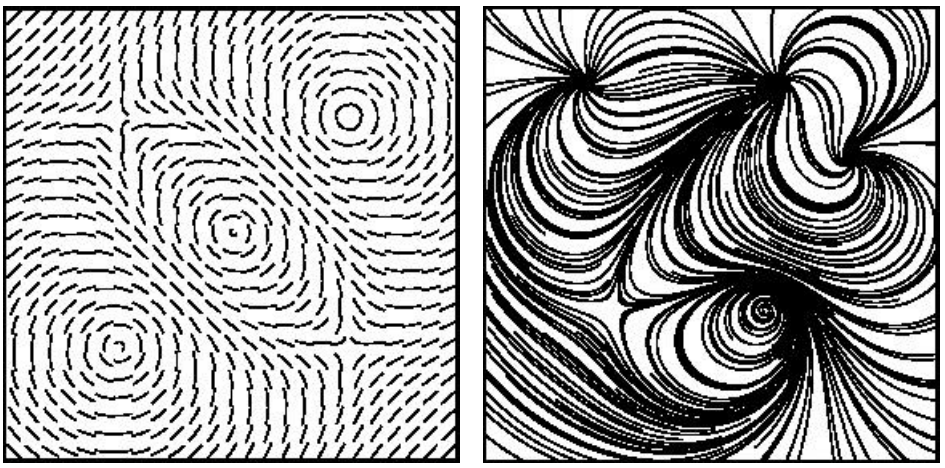


Figure 8: Regular seeding may introduce disturbing artifacts to the picture (left) or lead to unbalanced streamline distribution (right) [59]

Density Guided and Image Guided Streamline Seeding

The density driven seeding strategies as well as the image guided techniques focus on the problem of cluttering and address the aesthetic aspects of a flow visualization using streamlines. These methods also enforce a uniform spatial distribution of streamlines.

In [33] Max et al. present, beside other, a way to cover a 3D surface (not necessarily tangential to the field) with a set of streamlines. Once a seed point has been selected in the field, they make a streamline growing beyond that point back-ward and forward. The growing process is stopped when the streamline reaches an edge of the surface, a singularity in the field (source or sink) or becomes too close to another streamline. The streamline is then divided into a set of small segments of contrasting color and projected onto the surface. Although this method was intended to visualize a flow on a 3D surface, it can be generalized to all kinds of steady 2D fields.

In [57] the placement of streamlines at a specified density is reached via minimization of an energy function. The method uses a low-pass filtered version of the current image to measure the difference between this image and the desired density value. The energy is reduced iteratively by changing the positions and lengths of streamlines, merging streamlines, and creating new ones. The resulting placement has a hand-placed appearance and the streamlines appear to be neither too sparse nor too crowded. Computation time for their method is significant.

Jobard and Lefer [19] extend the Max's approach [33] showing how the algorithm can be controlled by the user to produce a wide range of flow fields images, ranging from hand-drawing to LIC-like style. They aim to produce long and evenly spaced streamlines. An important feature of the algorithm is that, unlike Turk's progressive refinement approach, it works in a single pass. To compute an image, a number of streamlines are calculated until a user-fixed density level has been obtained. When computing a new streamline, a new seed point is chosen at a minimal distance apart from all existing streamlines. Then a new streamline is integrated beyond the seed point backward and forward until it gets too close to some other streamline or it leaves the 2D domain in which the computation takes place. The algorithm ends when no more valid seed points can be found. The quality of this method's outputs is comparable to that of [57]. The demands on computational time, however, are significantly lower.

All these methods manage to avoid clutter, present easy to understand pictures and also dispose of disturbing artifacts that might lead to a misinterpretation of the flow field. However, from the scientific point of view, the most important property of a

visualization technique consists in its ability to display all the important features of the underlying phenomena. Methods in this group do not assure such behavior and in some cases, they may fail to satisfy the coverage criterion.

Flow Guided Streamline Seeding

As already mentioned above, the major drawback of the methods above is that they do not take guidance from the important features of the flow. None of the methods guarantees that the resulting streamlines will capture all the essential features of the flow field. Therefore, flow-guided streamline placement strategy was developed [59], which takes the advantage of the knowledge of important features contained by the flow.

The main idea consists in seeding around flow field's *critical points* (consult section Critical or Fixed Points in subchapter 3.4.1 for explanation) using *seed templates*, which correspond to individual types of these points. The procedure is straight-forward and consists of four essential steps:

1. Compute critical point locations and determine their type.
2. Segment the flow field into regions, each containing a single critical point, by constructing a Voronoi partitioning over the set of all these critical points.
3. Perform seeding at the vicinity of the critical points by applying the appropriate templates conforming to the individual types of critical points.
4. Randomly insert additional seed points in the field using a Poisson disk distribution to minimize closely spaced seed points.

Definition and classification of critical points will be given in the Feature Extraction chapter together with an overview of methods focusing on extracting these critical points from vector fields. Since the flow-guided seeding strategy exploits critical points to a large extent, its description requires the knowledge of this term as well as the knowledge of individual kinds of critical points. Briefly, we can say that critical points are locations in a vector field where the vectors are null. Critical points are then classified according to the behavior of the flow in their vicinity, thus forming centers, saddles, attracting and repelling foci (i.e. sources and sinks) and attracting and repelling spirals are recognized. (see Figure 9 for illustration).

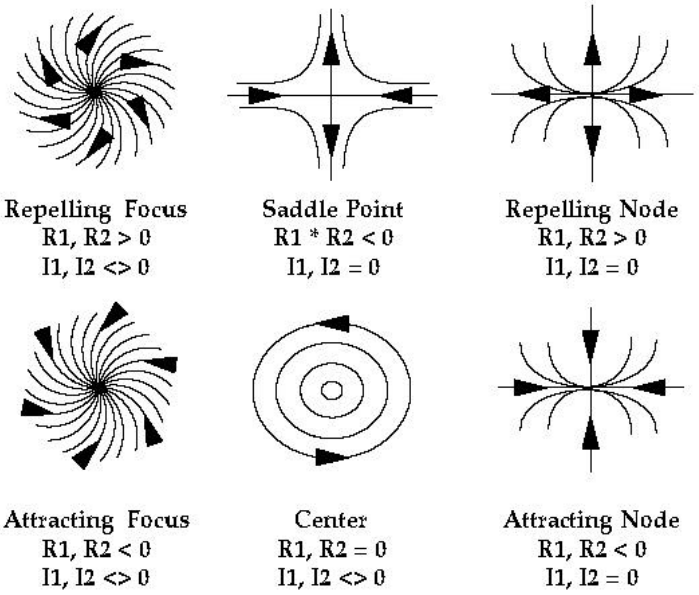


Figure 9: Critical point classification in 2D (from [38])

The authors of [59] have found out that in close surrounding of a critical point, the flow field usually resembles an ideal flow pattern for that type of critical point. The further we move from the critical point, the higher is the influence of the other critical points, which makes the flow behave less ideally in these areas. To be able to decide, which critical point influences a given position in a vector field the most, Voronoi segmentation is used to divide the flow field. Each Voronoi region then contains one critical point and the size of such Voronoi region is an approximation of the extent of this point's influence.

Deciding about the size and shape of the seed templates depends on the type of the critical point and was suggested as follows:

- *center, spiral*: For center and spiral type of critical points, the algorithm finds the line segment that joins the critical point to the closest point on the Voronoi boundary and seeds along this line segment. Although other line segments might also be used, the authors claim, that the ideal flow pattern of the critical points fades rather quickly, and thus other possibilities mostly result in too many streamlines and hence clutter.
- *source, sink*: For source and sink types of critical points, the algorithm seeds along a circle's perimeter. This circle has its center at the critical point and it is the largest circle that would fit completely inside the critical point's Voronoi region. Hence, the radius of this circle is equal to the distance between the critical point and the closest point on the Voronoi boundary. In contrast to centers and spirals, the ideal flow pattern of sources and sinks seem to extend further out.
- *saddle*: For saddles, seeds are placed along two lines. These lines are the bisectors of the principal eigenvector directions. The extent of these lines is decided by their intersection with the Voronoi boundary. The saddles are the trickiest to seed because if the seed

closest to the saddle's location along the bisectors is not close enough then the saddles are not captured properly. For this reason, it is wise to seed two special streamlines very close to the saddle. The seeds for these two streamlines are chosen to lie on the same bisector but on the opposite sides of the saddle's center. The distance of these special seeds from the center is chosen to be equal to one half the cell size of the grid.

In practice, the size of the templates can in fact be halved. Sufficient radius of the circle for seeding around source and sink critical points is then equal to half the distance of the critical point to the closest point on the Voronoi boundary.

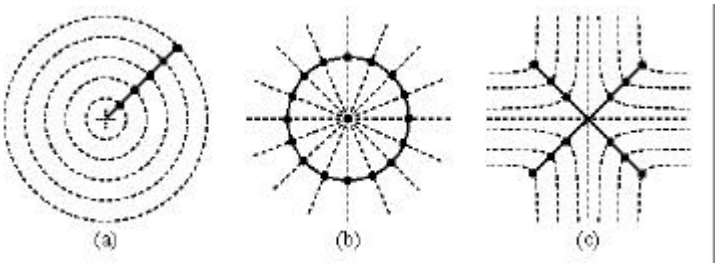


Figure 10: Critical point seeding templates

Seeding only in a surrounding of critical points, however, leaves large blank areas in the image. These regions do not contain any additional important features, hence the seeding strategy can be less careful here. Verma et al. fill these regions by random seeding driven by a Poisson disk distribution. Instead of searching for the blank areas, the authors first determine the, so called, regions of influence of the critical points (i.e. circles around the critical points with recommended, experimentally found, radius equal to 0.8 times the template radius) and then perform the random seeding outside these regions. Tracing from these random points introduces a few very short streamlines to the output picture. These streamlines can be omitted from the visualization. Their exclusion improves the aesthetic aspect and does not reduce the informative qualities of the image.

There are two important consequences of seeding with the seed template before the random seeds. By giving priority to seed templates, the coverage of flow patterns near critical points is assured. Furthermore, the algorithm terminates a streamline when it comes close to an existing streamline, thus earlier streamlines will tend to be longer than later streamlines. Hence, streamlines traced from the seed templates are longer than those traced using seeds placed randomly to fill in blank spaces. Such a strategy ensures that the regions in the flow field close to critical points are given more importance than other regions.

The authors compared this technique to the one of Turk and Banks presented in [57] and briefly described above. Although the uniformity goal seems to be better satisfied by the image-guided approach making the flow-guided method generated pictures look slightly worse from the aesthetic point of view, the coverage criterion, which was claimed to be the primary visualization goal, is definitely better met by the flow-guided technique.

Moreover, it is necessary to realize that with the increasing number of traced streamlines, the iterative nature of the image-guided algorithm causes a significant growth of computational time. Potential further savings of computational time with the flow-based technique arises from the fact that it is view-independent, unlike the image-

guided approach. This will show when users need to inspect the flow field from different view-points.

Stream Ribbons, Polygons, Tubes, Balls, Surfaces, Arrows et cetera

Many extensions of streamlines have been developed because of various demands on the visualization. We will only mention them briefly here, together with a short description and a reference to the sources, where the methods were presented.

Stream ribbons [58] are in fact stream lines with a winglike strip, whose orientation shows the rotational character of the flow.

Stream polygons [47] were developed to visualize tensor information in a vector field. The stream polygon is a regular n-sided polygon oriented normal to a local 3D vector in a given point and is deformed by either the whole deformation tensor, just a component of it, or an additional derived tensor (e.g. the vorticity tensor for a velocity field). Polygon's shape, radius, number of sides and rotation reflect the vector field quantities.

Stream tube [58] is constructed by computing a streampolygon in every point of a streamline.

Stream balls [4] serve for visualization of divergence and convergence as well as acceleration and slow down via splitting and merging.

Stream surfaces [17] are surfaces, which are tangent to the flow. Stream surfaces can be approximated by connecting a set of streamlines along timelines.

Stream arrows [31][32] visualize flow direction, convergence or divergence and other flow properties by cutting arrow-shaped pieces out from the stream surfaces, thus leaving transparent arrow-shaped holes in these surfaces. Not only that the holes reflect some flow features, they also allow the user to see through the surface in the front and discover surfaces, which would normally be hidden behind it.

Streak Lines and their Seeding

As already mentioned, streak lines are integral curves, produced by simulating continuous injection of particles into the flow field from constant location over certain period of time. Displaying streak lines serves, at the first place, for unsteady flow data visualization, for, in case of steady flows, streaklines coincide with streamlines and pathlines. Visualizing unsteady vector data requires proper choice of the method to use. Streamlines, for instance, bring certain amount of information about the underlying field too, however, the results might be misleading in a way. Streamline computation always exploits vector information from just a single instant of time. A trace obtained this way, therefore, describes the trajectory of an imaginary massless particle moving through the flow field at infinite speed, which diverges from what users would usually expect. Such technique is called *instantaneous*. Streak lines, on the other hand, belong among *time-correlated methods*, which progressively include information from consecutive temporal instants letting the integral curve develop in time.

The nature of the instantaneous methods causes problems when used for visualizing time-varying data via animations. Since they only exploit information from one temporal level for creating a frame of the animation, they suffer from a lack of coherence between individual images. A comparison of instantaneous and time-correlated methods can also be found in [27] and [28], where some implementation details about computing streaklines and timelines are mentioned.

The disturbing problem with lack of temporal coherence appears for example if LIC (Line Integral Convolution) images are computed for each time step separately and then put together, one after another, to make up an animation. An improvement has

been reached by Forssell and Cohen [13], who replaced streamlines with pathlines. Their method produces better visualization of time-varying vector fields, yet it lacks temporal coherence, where the flow is fairly unsteady. With time correlated methods, this is not the case. The first approach of this sort is probably UFLIC (Unsteady-Flow LIC), which employs certain kind of temporal convolution. Both, LIC and UFLIC will be explained later, because they fall into the category of texture base methods. We have mentioned them here to stay consistent, since the following methods draw on their findings.

Sanna et al. [43] propose a geometric approach, which builds upon the experience of the methods above and overcomes the temporal coherency problem. Their algorithm follows streaklines in order to produce an image for each time step of an animation. They also came up with an important finding that streak lines, as they develop in time, overlap easier then streamlines and their orientation is less evident, if just a single frame is inspected. Thus, the results might confuse the user.

To fix this flaw, another improvement was reached in [44] by designing more adequate seeding strategy. The density of traced streaklines differs according to the local *vorticity* of the vector field (refer to section Vorticity, Rotation or Curl in 3.4.1). In the areas where higher velocity gradients appear, a larger number of traces is displayed, while in the regions with smoother flow, a smaller number of streaklines is traced. Moreover, each particle of a streakline is denoted by a color depending on the character of the local vorticity in the area around such particle.

A set of insertion points, from which the particles are released, will correspond to a set of pixels on the output texture. The insertion points are kept constant for the whole animation. At each time step a new particle is released from each location and the positions of the previously emitted particles are updated. In this way, each frame of the animation maintains the coherence with the previous ones and the resulting sequence can effectively show the evolution of the field in time.

Similarly as in the flow-guided streamline seeding strategy (see above), it is necessary to avoid too sparse or even blank areas in the final image. Therefore, in order to guarantee a minimum level of details all over the resulting image, the streaklines starting from a subset of insertion points are traced independently of the vorticity values. On the other hand, the particles released from the other insertion points will affect the output frame only if they are placed in field zones where the vorticity is greater than a user predefined threshold. In this way, the user can tune the magnitude of the flow field details to be displayed.

Finally, the maximum length of the streak lines (L) can be set by the user before the visualization starts. After releasing new particles in certain time step, the positions of the previously emitted particles must be updated according to their locations inside the vector field. Should the streak line exceed the length limit that is should it consist of at least L+1 particles, the oldest one is dropped.

Time Surfaces

Time surfaces are the 3D equivalents of timelines in 2D. Time surface is produced by inserting a set of particles from a 2D patch into the flow in one instant of time. The evolution of the time surface's shape reflects the character of the vector field.

3.3.3 Texture Based Methods

As well as the geometric visualization techniques, texture based algorithms utilize integral curves, the computation of which was discussed in 3.3.1. Instead of displaying them as individual geometric entities, a convolution with some kind of input texture is

performed. The type of the input texture as well as the integral curve used then makes the difference between individual techniques. Apparently, texture base techniques need two input structures, these being the vector field and the texture to convolve it with. An overview of texture based vector field visualization methods can be found in [45].

Spot noise

Spot noise [61] is considered to be the first texture based vector field visualization technique. The name is derived form the texture type used for the convolution. It contains small, spots resembling, intensity functions distributed over the data domain. Each of these spots then becomes a little dispersed by the influence of the vector field and also moved on a path $\mathbf{x}_i(t),\ t \in [t_1, t_2]$. Therefore, the image of a single particle can be obtained as:

$$h_i(\mathbf{x}) = \int_{t1}^{t2} \mathbf{A}(\mathbf{x} - \mathbf{x}_i(\mathbf{t}))d\mathbf{t}, \tag{24}$$

where **A** is a particle spread function. Image h_i is the, so called, spot (or streak) of a single particle. The sum of all spots characterizes the whole texture:

$$f(\mathbf{x}) = \sum_i a_i h(\mathbf{x} - \mathbf{x}_i), \tag{25}$$

where **x** is a random starting point for each spot, and a_i is a random scaling factor with a zero mean. The shape of the intensity function h_i is deformed according to the vector field, which stretches the spots elliptically in the direction of the local field. The algorithm was ineffective when visualizing regions with high velocity gradients.

Line Integral Convolution (LIC)

Line integral convolution (LIC) first presented in [8] and latter reviewed in [7] stands for probably the best known texture based integral method for visualizing vector fields. On the input, it takes the vector field to be visualized and a texture – usually some kind of random image like for example white noise of the same resolution as the vector field grid. These are then convolved together producing an image resembling surface oil patterns, which can be seen in real world experiments. In other words, the white noise texture is smeared along the direction of the streamlines. The convolution, LIC performs, is thus one dimensional, which effectively correlates pixels located along stream lines and leaves pixels in the transverse direction uncorrelated. A low pass filter is used to restrict the convolution only to streamline segments of selected length.

To be precise, equation (3) defined for unstable flows must first be rewritten to define an integral curve[1] in a time independent vector field:

$$\mathbf{p}(s) = \mathbf{p}_0 + \int_{t=0}^{s} \mathbf{v}(\mathbf{p}(t))dt. \tag{26}$$

Having expressed a streamline $\mathbf{p}(s)$, LIC computes the pixel intensity at $\mathbf{x}_0 = \mathbf{p}(s_0)$ as

[1] In the following text, this integral curve will be reffered to as a streamline. As mentioned above, streamlines, streaklines and pathlines coincide in time independent vector fields.

$$I(\mathbf{x}_0) = \int_{s_0-L}^{s_0+L} k(s-s_0)T(\mathbf{p}(s))ds\,, \tag{27}$$

where T denotes an input texture. The filter kernel k is assumed to be normalized to unity. In [53], the filter length L equal to 1/10th of the image width is recommended. Animating and coloring LIC images (for instance according to the velocity magnitude) can further extend the informative value of the results.

As a powerful method, LIC draws researchers' attention. Thus, a method for comparative analysis of this algorithm and visualizing streamlines by using non white noise texture was suggested in [60] and is called PLIC (Pseudo-LIC). Shen et al. [51] on the other hand developed a visualization software based upon line integral convolution called GLIC (Graphical Line Integral Convolution). LIC has also been extended to 3D surfaces in [1]. Besides these, LIC was upgraded to successfully visualize time dependent vector fields in [50]. This is an important modification, which will be described bellow in a separate section. First, however, some improvements to the LIC technique itself will be described.

Despite the unquestionable effectiveness of the basic LIC algorithm, Stalling and Hege [53] brought significant advances to this technique. They focused especially on the speed up, smooth animations and detail enlargement.

Speeding up LIC

In traditional LIC a separate stream line segment and a separate convolution integral are computed for each pixel in the output image. Since a single stream line usually covers lots of image pixels, it redundant to always recompute large parts of a stream line. Furthermore, for a *constant* filter kernel k very similar convolution integrals occur for pixels covered by the same stream line. In [53], exploiting these observations is proposed.

Consider two points located on the same stream line, $\mathbf{x}_1 = \mathbf{p}(s_1)$ and $\mathbf{x}_2 = \mathbf{p}(s_2)$ separated by a small distance $\Delta s = s_2 - s_1$. For constant k then obviously:

$$I(\mathbf{x}_2) = I(\mathbf{x}_1) - k \cdot \int_{s_1-L}^{s_1-L+\Delta s} T(\mathbf{p}(s)) \cdot ds + k \cdot \int_{s_1+L}^{s_1+L+\Delta s} T(\mathbf{p}(s)) \cdot ds \tag{28}$$

The intensities differ by only two small correction terms that are rapidly computed by a numerical integrator. By calculating long stream line segments that cover many pixels and by restricting to a constant filter kernel, both the redundancies can be avoided.

Animating Stable Flows

Changing the shape and location of the filter kernel k over time causes the LIC images to be animated, thus reflecting not only the tangential but also the *directional* information. The above restriction to constant filter, however, requires a different approach. This can be achieved by rotating the box filter, which however introduces disturbing artifacts, when the boxes reenter the interval. To see this, consider two points p_1 and p_2 on a single stream line that are half a filter length apart. The corresponding pixel intensities initially have a 50% correlation because half of the texture cells being convolved are covered by both filter boxes. When the filter boxes reenter the interval, correlation suddenly drops to zero, as demonstrated in Figure 11.

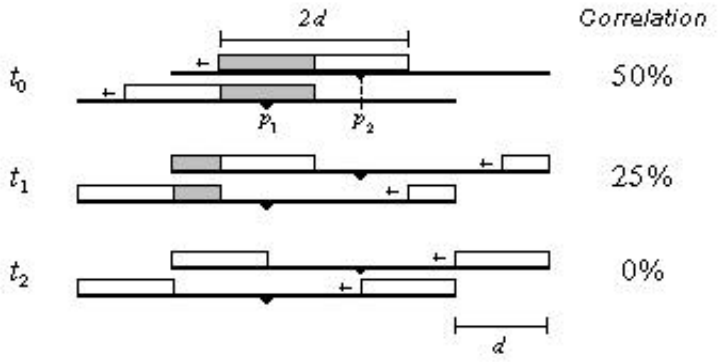


Figure 11: Correlation drop off with the box filter kernel shifting [53]

Since changing the filter kernel, which would solve this problem, is not possible with the fast version of LIC, the authors suggest using frame blending. In this technique, a sequence of images B_n, n = 0, 1,..., N-1, is computed, with the box filter running just once along a streamline segment. Such sequence is not periodic any more, but exhibits a constant intensity correlation. A periodic sequence A of length $N/2$ may be obtained by smoothly blending between two phase-shifted B-frames:

$$A_n = w_1(n)B_{n \bmod N} + w_2(n)B_{(n+1/2) \bmod N}. \tag{29}$$

Weights w_1 and w_2 are chosen as:

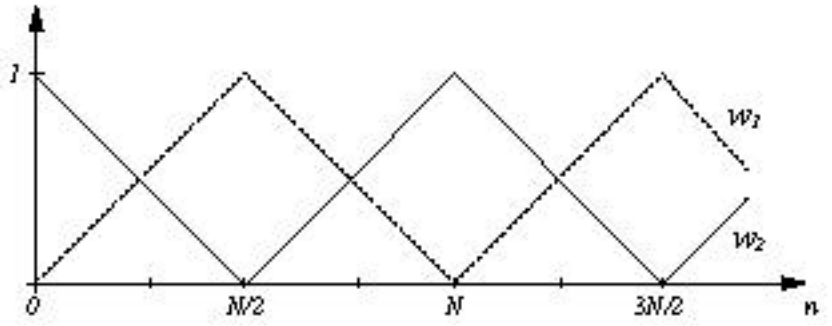


Figure 12: Weights for frame blending [53]

Then, pictures near the critical positions will have small weights and the transition from one cycle to another will be smooth.

A great advantage can be seen in the fact, that this technique (unlike the standard filter cycling approach) can also be applied when it is necessary to animate pictures with *variable* velocities for individual pixels. Thus, vector magnitude $|\mathbf{v}|$ can be recorded as well.

Level of Detail

The traditional LIC requires the output image to be of the same size as the input texture. Fast LIC on the other hand allows to choose the output picture size independently of the input image and the vector field resolution. Therefore, it is easy to zoom in and out to the vector field in case of necessity. Since the fast LIC algorithm utilizes the fourth order Runge-Kutta integration scheme described by equation (6), only the time step size Δt needs to be adjusted adequately when changing resolution.

Unsteady Flow Line Integral Convolution (UFLIC)

As mentioned in the section devoted to LIC, it is possible to animate the output images to create a notion of a real flow. Referring to section 3.1.1, however, we recall that such visualization only illustrates one temporal dimension, resulting from the differential character of vector data. Yet, the data themselves are invariant in time. Another temporal dimension is encountered, when dealing with time-dependent data, where the vector field itself changes in time. We have already encountered this situation in section Streak Lines and their Seeding in subchapter 3.3.2, where we have explained the tricky physical meaning of streamlines in time-dependent data, defined the *instantaneous* and *time-correlated* methods and drawn the distinction between them. In the same section, we have also mentioned, why it is necessary to base time dependent data visualization on streaklines, rather than streamlines. The LIC derived method tackling this problem is called UFLIC and will now be briefly outlined.

This algorithm extends the LIC method by devising a new convolution algorithm that simulates the advection of flow traces globally in unsteady flow fields. The input texture (white noise as for LIC) is advected over time to create directional patterns of the flow at every time step. The advection is performed by using a new convolution method, called time-accurate value scattering scheme. In the time-accurate value scattering scheme, the image value at every pixel is scattered following the flow‘s pathline trace, which can be computed using numerical integration methods. At every integration step of the pathline, the image value from the source pixel is coupled with a timestamp corresponding to a physical time and then deposited to the pixel on the path. Once every pixel completes its scattering, the convolution value for every pixel is computed by collecting the deposits that have timestamps matching the time corresponding to the current animation frame. To track the flow patterns over time and to maintain the coherence between animation frames, a process is devised, called successive feed-forward, which drives the convolutions over time. In the process, the time-accurate value scattering is repeated at every time stamp. Instead of using the white noise image as the texture input every time, the algorithm takes the resulting texture from the previous convolution step, performs high-pass filtering and then uses it as the texture input to compute the new convolution.

3.4 Feature Extraction

This chapter might also be called Topological Representation of Vector Fields. Methods described here are based on extracting some topologically important features from the vector field and their subsequent visualization. The advantages are apparent. Displaying only the important features will significantly reduce the amount of visualized data. Since the original datasets are usually rather huge, such reduction is necessary from at least two reasons. Visualizing all the data values would, on the first place, be too slow. Secondly, occlusion and clutter caused by the quanta of more or less uninteresting data might prevail and dominate the final views.

A disadvantage is that if some of the important features, which are present in the original dataset, remained unrecognized by the extraction algorithm, it might lead to misinterpreting the character and content of the information the vector field contains.

3.4.1 Feature Extraction Dictionary

The task of this section is to introduce basic concepts and some important and frequently used terms from the feature extraction field. In the first place, the relation

between vector fields and general dynamical systems will be drawn along with the recipe, how to represent a vector field so that the dynamical system analysis tools can be applied to investigate it.

Vector Fields as Dynamical Systems

As already mentioned in sections 3.1.2 and 3.1.3, vectors in linear fields can be expressed in the form $\mathbf{v} = \mathbf{Ap}$, where vectors $\mathbf{v}$ are functions of the spatial locations $\mathbf{p}$, with $\mathbf{A}$ being a constant n×n-matrix (in case of time independent fields). Such a description classifies the vector field as a dynamical system, but it is only available in case of analytical models. In practice, one usually works with a set of discrete samples of the data. Numerical methods must thus be employed to obtain an approximation of matrix $\mathbf{A}$, whose analysis plays a crucial role for investigating the vector field via the feature extraction methods. Before describing this process, the terminology will be clarified using [32] and [30].

Gradient and Jacobian

Gradient and *Jacobian* are denoted by operator ∇, which produces a vector of partial derivatives of its operand as shown in equation (30), where $\nabla f(\mathbf{x})$ is called gradient for scalar operand $f(x)$ and $\nabla\mathbf{v}(\mathbf{x})$ Jacobian matrix for vector function $\mathbf{v}(\mathbf{x})$.

$$\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n}\right), \text{grad } f(\mathbf{x}) = \nabla f(\mathbf{x}), \mathrm{J} = \nabla \mathbf{v}(\mathbf{x}) = \partial \boldsymbol{v} / \partial \mathbf{x}, \qquad (30)$$

Divergence

Divergence div $\mathbf{v}(\mathbf{x})$ of a flow is a frequently used scalar quantity, which can be described as $\nabla \cdot \mathbf{v}(\mathbf{x})$ or as the trace Tr of the Jacobian $\nabla \mathbf{v}(\mathbf{x})$. Symbolically:

$$\text{div } \mathbf{v}(\mathbf{x}) = \nabla \cdot \mathbf{v}(\mathbf{x}) = Tr(\nabla \mathbf{v}) = \sum_i (\partial \mathbf{v} / \partial \mathbf{x})_{i,i} \ . \qquad (31)$$

Divergence determines the local amounts of incoming and outgoing flow and equals to zero if these amounts are the same.

Vorticity, Rotation or Curl

Vorticity, *rotation* and *curl* all denote a vector w = rot $\mathbf{v}(\mathbf{x})$, which points in the direction of the axis of the flow's rotation and its length equals to twice the angular velocity [44]:

$$\boldsymbol{?} = \text{rot } \mathbf{v}(\mathbf{x}) = \text{curl } \mathbf{v}(\mathbf{x}) = \nabla \times \mathbf{v}(\mathbf{x}) . \qquad (32)$$

Stream Vorticity

The cosine of the angle enclosed by the vorticity vector rot $\mathbf{v}(\mathbf{x})$ and the flow vector $\mathbf{v}(\mathbf{x})$ defines the scalar term *stream vorticity W*. Stream vorticity equal to 1 thus implies a flow rotating around the flow vector $\mathbf{v}(\mathbf{x})$, while *W* equal to zero identifies a location, where either the flow rotates in the plane containing the $\mathbf{v}(\mathbf{x})$ vector or where there is no rotation at all. Symbolically:

$$\Omega = \frac{\mathbf{v} \cdot \boldsymbol{?}}{|\mathbf{v}| \cdot |\boldsymbol{?}|} = \frac{\mathbf{v} \cdot (\nabla \times \mathbf{v})}{|\mathbf{v}| \cdot |\nabla \times \mathbf{v}|} . \qquad (33)$$

Helicity or Helicity Density

Helicity and *helicity density* again denote the same quantity H_d, which defines the same as no stream vorticity if equal to zero. Helicity

$$H_d = \Omega \cdot |\mathbf{v}| \cdot |\mathbf{?}| = \mathbf{v} \cdot \mathbf{?} = \mathbf{v} \cdot (\nabla \times \mathbf{v}), \tag{34}$$

however, increases proportionally to the length of w and **v**.

Circulation

Circulation Γ_c equal to zero for any closed curve C indicates that a potential function *f* exists, for which grad *f*(**x**) = **v**(**x**), and this function can then be used for analysis instead of **v**. If $\Gamma_c = 0$ for $\forall$C, then there is no rotation in the field at all. Mathematically:

$$\Gamma_c = \oint_c \mathbf{v}(\mathbf{x})\, ds = \int_S \text{rot}\ \mathbf{v}(\mathbf{x})\, dS\,, \tag{35}$$

with *S* being the surface of an arbitrary volume containing closed curve C.

Critical or Fixed Points

Critical points, sometimes also called *fixed points*, are locations within the vector field, where vector **v** vanishes to zero. These points are important topological places and their location provides a basis for many visualizing methods.

3.4.2 Analyzing the Transformation Matrix

At the beginning of the previous section, vectors **v** were expressed as a function of positions **p**, where the relation was described by matrix **A** as **v** = **Ap**. Hence, **A** can be regarded as a transformation matrix between the spaces of **p** and **v**. Analyzing this matrix will therefore bring valuable information about the flow, sufficient to investigate it. First, however, it is necessary to describe, how to extract such matrix from the discrete samples if the analytical model of the system is unknown.

Approximation in the Discrete Case

If the analytical model is unknown, the Taylor series expansion must be utilized locally to find the relation between **v** and **p**, supposing the flow **v** to be sufficiently smooth and differentiable. In such case, the expansion of **v** about point $\mathbf{x_0}$ is [14]:

$$\mathbf{v} = \mathbf{v_0} + (\mathbf{x} - \mathbf{x_0}) \frac{\partial \mathbf{v}}{\partial \mathbf{x}} + O(\Delta \mathbf{x}^2)\,. \tag{36}$$

Omitting the remainder term, equation (36) can be rewritten using the matrix notation as [20]:

$$\mathbf{v} = \mathbf{v_0} + \mathbf{J} \cdot \Delta \mathbf{x}$$

$$\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} v_{0,x} \\ v_{0,y} \\ v_{0,z} \end{pmatrix} + \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{pmatrix} \cdot \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} \tag{37}$$

where (dx, dy, dz) is the Cartesian coordinate vector assuming $\mathbf{x_0}$ to be the origin. Vector $\mathbf{v_0}$ is the velocity vector in the origin and $\mathbf{J} = \nabla\mathbf{v}$ the Jacobian matrix. The coefficients of $\mathbf{v_0}$ and those in the Jacobian matrix are constants. If the discrete mesh decomposes into tetrahedral elements, these constants have an analytic solution as a tetrahedron has four vertices, precisely the right number of datum points to evaluate the 12 coefficients of a 3D linear interpolation function. For linear vector fields, like that described by equation (37), the flow's general shape can be determined. Matrix **J** only has local validity (i.e. it is different for each tetrahedron) and it is most often examined in critical points, where $\mathbf{v_0}$ equals to a zero by definition and **J** is thus identical to matrix **A** in the definition for the analytical models. The advantage of investigating **J** near critical points is that in these locations, non-linear fields can also be studied, because the higher order terms in the Taylor series expansion tend to vanish near the critical points, thus the linearization of the non-linear field by disregarding these terms does not cause significant inaccuracies (see [32] for details). The information encoded in the Jacobian matrix **J** will be revealed in the following paragraphs.

Eigenvalue/Eigenvector method

One way to examine the behavior of a vector field is to compute the eigenvalues l_i (from $\det(\mathbf{J} - l_i \cdot \mathbf{I}) = 0$) and corresponding eigenvectors $\mathbf{e_i}$ (from $\mathbf{J} \cdot \mathbf{e_i} = l_i \cdot \mathbf{e_i}$) of **J** in the critical points. When interpreting the Jacobian matrix as a transformation, its eigenvectors will point in the directions, which are invariant to this transformation. The way such a line itself is transformed is given by the corresponding eigenvalue l_i.

The constellation of the eigenvalues and eigenvectors gives rise to critical point classification, for the 2D case already mentioned in the Flow Guided Streamline Seeding section in subchapter 3.3.2. In 3D, the situation is analogical, as depicted in Figure 13. Although more combinations are possible, the principle is the same.

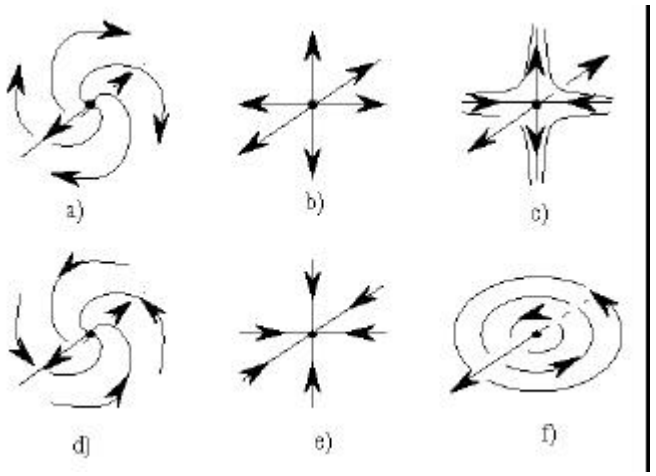


Figure 13: Vortex classification in 3D

When a critical point is located, its type must be determined. This is done by examining the three eigenvalues of **J**, which can be either three real numbers or one real and two complex conjugate numbers. The two eigenvectors derived from the complex conjugate eigenvalues define a plane, which contains the swirl of the flow. The third one is the axis of the swirl and if subtracted from the vector field a purely rotational flow will be obtained.

As already mentioned, critical points stand for a topologically important feature and they are often used as the starting location for integrating streamlines, thus describing the field comprehensively.

Jacobian Matrix Decomposition

Another possibility to extract the information from the Jacobian matrix **J**, is to decompose it into a symmetric matrix $\mathbf{J}^+$ and an antisymmetric matrix $\mathbf{J}^-$ [32], where:

$$\mathbf{J}^+ = (\mathbf{J} + \mathbf{J}^\mathbf{T})/2, \qquad \mathbf{J}^- = (\mathbf{J} - \mathbf{J}^\mathbf{T})/2\,. \tag{38}$$

The elements of these two matrices then have the following meaning:

$$\mathbf{J}^+ = \begin{pmatrix} d_x & \bullet & \bullet \\ \bullet & d_y & \bullet \\ \bullet & \bullet & d_z \end{pmatrix}, \text{ and } (d_x + d_y + d_z) = \text{div } v(x), \tag{39}$$

with the elements $\bullet$ represent the shear strain,

$$\mathbf{J}^- = \frac{1}{2}\begin{pmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{pmatrix}, \text{ and } \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} = \text{rot } v(x)\,. \tag{40}$$

J in Local Coordinate System

If the flow's Jacobian **J** is studied at some point of a trajectory of the flow, it can also be transformed to conform the local coordinate system determined by the Frenet-Frame ($\mathbf{J} \rightarrow \mathbf{J}^{\text{loc}}$), whose elements are given in

$$\mathbf{J}^{loc} = \begin{pmatrix} a & s & s \\ c & d & d,t \\ c & d,t & d \end{pmatrix}, \tag{41}$$

where elements *a*, *s* and *c* describe changes of the flow parallel to **v**(**x**). More precisely, *a* gives the acceleration, *s* the shear strain and *c* the curvature. Elements *d* and *d*, *t*, on the other hand, specify changes in the direction perpendicular to **v**(**x**). Splitting this 2×2 part of the matrix into a symmetric and an antisymmetric parts, elements *d* would denote divergence and *t* the torsion of the flow.

3.4.3 Extraction of Features

Using the above described knowledge of the vector field, various physical characteristics can be searched within the data. These characteristics are called features. The choice of features to look for within a data set depends heavily on the application, taking into account the character of the vector data and the purpose of visualization. Probably the most frequently used examples from the field of flow visualization are vortex cores, shock waves, separation and attachment lines etc. Numerous methods exist to locate and extract these features within a vector fields and for tracking these features over time in case of time-dependent data. For an overview of the main approaches, refer to [38] and especially to [41].

3.5 Derived (Scalar) Value Visualization

Another approach of visualizing vector fields is based upon the fact that in some situations it is possible or even necessary to represent the vector data by some scalar values. A very intuitive example is searching for a region in a flow field, where velocity magnitude reaches certain user specified limit. Similarly as in the methods discussed in the previous chapter, where some objects (integral or topological) were first extracted from the vector field and then visualized, here scalar values are derived from the vector data first and these values are then processed and visualized. The most common operation with the scalar values being isosurface extraction. The whole process is sometimes also called dimension contraction. In multidimensional data visualization, reducing dimension is a generally popular concept, as we will see later in the chapter devoted to tensor field visualization.

3.5.1 Types of Derived Values

In this section we outline a few examples of scalar values, which may be derived from vector data and which may be requested for visualization by the user. These examples come from [39]:

- The magnitudes of all velocity vectors $\|\mathbf{v}\|$ define a scalar field.
- The kinetic energy density is $\frac{1}{2} \cdot r \cdot \|\mathbf{v}\|^2$.
- The scalar product of two vectors is a measure of the angle f between them:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cdot \cos f \tag{42}$$

  This can be used to find the components of all velocity vectors in a given direction, or to find the changes in direction at two neighboring points.

- The magnitude of the *vorticity* w may be used to find vortices. Using w, *helicity density* is computed as: $H_d = \mathbf{v} \cdot \mathbf{?}$.
- The scalar f_{shock} is defined for compressible media as:

$$f_{\text{shock}} = \frac{\nabla p}{|\nabla p|} \cdot \frac{v}{c}, \tag{43}$$

  in which c is the speed of sound. The isosurface for $f_{\text{shock}} = 1$ shows shock waves.

3.5.2 Visualizing Derived Values

Once computed, these scalar values may be depicted and investigated using common methods for scalar volumetric data visualization. The term “Once computed”, however, does not mean that the whole scalar field needs to be computed first in preprocessing and only then visualized. In fact, with regard to the increasing size of data sets, these scalar values must often be derived from the vector data on the fly. On the other hand, should the user need to, for instance, extract multiple isosurfaces, it is wise to consider precomputing the whole scalar field in advance and storing it together with the vector field if possible. Simply, the ubiquitous trade-off between size and speed must be considered before choosing the right attitude for implementation.

We will not describe all the numerous techniques devoted to scalar field visualization (iconic visualization, direct volume rendering etc.), because it falls behind the scope of this work. On the other hand, we will briefly touch probably the most popular approach, isosurface extraction and visualization (chapter 6), since it is one of the topics we have dealt with.

4 TENSOR FIELD VISUALIZATION

Similarly as vector fields, tensor datasets also represent a crucial form of storing information in numerous engineering and physics disciplines. To be precise, vector (or scalar) field is nothing else but a field of first (or zero) order tensors. This chapter, however, only deals with second and higher order tensors. In this respect, the most frequently used tensor datasets are fields of second order tensors, on which we will focus primarily. Yet higher order tensors can also be encountered and therefore we will give them some attention as well. An example of higher order tensor would be describing the piezoelectric properties of a crystal by $27 = 3^3$ quantities, thus using a third order tensor. Elasticity of anisotropic body requires $81 = 3^4$ numbers, which means a fourth order tensor [35].

As one might have expected, multiple approaches for visualizing tensor fields have appeared. The information encoded in tensors is, however, very complex and, as compared to vectors, much less intuitive for humans. Lucidity is very hard to reach in this field of visualization and thus the methods need to be rather smart, not to have the user misinterpret the results and gone astray.

4.1 Theoretical Background

To prove that the motivation for visualizing tensor fields is not artificial, a list of some commonly used physics tensor quantities, which deserve visualizing, will be presented first. Afterwards, some basic tensor theory will be shortly explained. A comprehensive description with examples can be found in [35] (in Czech). Illustrative images and explanatory videos concerning concrete results of advanced techniques for tensor data visualization can be found at [62].

4.1.1 Physical Tensor Quantities

In [10] some physics tensor quantities from the fluid flow area are listed. We will also use them for illustration (see Table 3).

$v_{i,k} = \frac{\partial v_i^*}{\partial x_k}$	Velocity gradient (u)
$e_{ik} = v_{i,k} + v_{k,i}$	Rate-of-strain tensor (s)
$s'_{ik} = he_{ik}^{**}$	Viscous-stress tensor (s)
$s_{ik} = -pd_{ik} + s'_{ik}$	Stress tensor (s)
$\Pi'_{ik} = pd_{ik} + rv_i v_k$	Reversible momentum flux density tensor (s)
$\Pi_{ik} = \Pi'_{ik} - s'_{ik}$	Momentum flux density tensor (s)

* In non-Cartesian coordinate system, covariant derivatives must be used instead
** In compressible flows, there is an additional term involving the divergence of the velocity field.

p = pressure
r = mass density
n_i and n_k = velocity components
η = viscosity
δ_{ik} = Kronecker symbol
(u) = unsymmetrical
(s) = symmetric

Table 3: Tensor fields in fluid flows

4.1.2 Tensor Fundamentals

More definitions of tensor exist depending on which feature should be emphasized. The most common way is to define tensor as an object, which obeys a specific transformation rules under a change of coordinate system. But it can also be defined as a multi-linear map between vector spaces [3]. For the purposes of the following paragraphs, the second definition suits better. A second order tensor quantity is defined to be a bilinear map $w \otimes n$ such that:

$$(w \otimes n) \cdot (\mathbf{a}, \mathbf{b}) = (w, \mathbf{a}) \cdot (n, \mathbf{b}) = 1 \tag{44}$$

Here w and n are co-vectors, i.e. linear maps (dot products) on the **a** and **b** vectors such that $\omega \cdot \mathbf{a} = 1$ and $\nu \cdot \mathbf{b} = 1$.

In more familiar matrix notation for a 3-dimensional space, equation (44) can be expressed as:

$$\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} w_1 n_1 & w_2 n_1 & w_3 n_1 \\ w_1 n_2 & w_2 n_2 & w_3 n_2 \\ w_1 n_3 & w_2 n_3 & w_3 n_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = 1 \tag{45}$$

Thus a second order tensor takes the form of a square matrix and associates state with two directions in space. Provided we have a tensor $\mathbf{T} = w \otimes n$ and the vectors **a** and **b**, the original co-vectors may be computed form the following expression:

$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} = \begin{bmatrix} w_1 n_1 & w_2 n_1 & w_3 n_1 \\ w_1 n_2 & w_2 n_2 & w_3 n_2 \\ w_1 n_3 & w_2 n_3 & w_3 n_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{46}$$

and

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} w_1 n_1 & w_2 n_1 & w_3 n_1 \\ w_1 n_2 & w_2 n_2 & w_3 n_2 \\ w_1 n_3 & w_2 n_3 & w_3 n_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{47}$$

Indeed, the inner product of **T** and any linear combination of **a** and **b** produces the appropriate linear combination of *w* and *n*. Thus the tensor is a specific map between the vector space spanned by **a** and **b** and the vector space spanned by *w* and *n*.

For example, the stress tensor at a point is a set of components containing stress state information for any arbitrarily oriented plane passing through the point. Multiplication of a unit vector representing a plane normal by the stress tensor gives one of the two *stress vectors* (also known as a *traction vectors*) representing the stress on that plane (Multiplication by the negative normal would yield the second stress vector acting on this plane):

$$\mathbf{s} = \mathbf{S} \cdot \mathbf{n}$$
$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} \tag{48}$$

In this case, we are mapping between the set of 3D plane normals and the set of 3D stress vectors acting on those planes.

Tensor Decomposition

General second order tensors contain nine independent scalar quantities. It is desirable to reduce this dimensionality in a meaningful way as an aid in understanding the physical state represented by a tensor. To do so, some visualization techniques exploit S*ymmetric-Antisymmetric* decomposition. Another possibility is a *Polar Decomposition*. Sometimes it is also desirable to filter out a background isotropic contribution, which, if dominant, may suppress interesting features. Decomposition to *isotropic* and *deviator* tensors is used for that purpose. These three operations are described in the following paragraphs.

Symmetric-Antisymmetric Decomposition

Any second order tensor may be decomposed into the sum of a symmetric tensor **S** and an antisymmetric tensor **A** [35]. Symbolically written:

$$\mathbf{T} = \mathbf{S} + \mathbf{A} = \frac{1}{2}(\mathbf{T} + \mathbf{T}^t) + \frac{1}{2}(\mathbf{T} - \mathbf{T}^t) \tag{49}$$

Where,

$$\mathbf{S} = \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{12} & s_{22} & s_{23} \\ s_{13} & s_{23} & s_{33} \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} 0 & a_{12} & a_{13} \\ -a_{12} & 0 & a_{23} \\ -a_{13} & -a_{23} & 0 \end{bmatrix}. \tag{50}$$

The antisymmetric tensor **A** is also called *axial*.

Polar Decomposition

Any second order tensor may be expressed as a product of a stretch tensor **V** and an isometric transformation tensor **Q**. There are two equivalent ways to do so (Figure 14). Symbolically written:

$$\mathbf{T} = \mathbf{Q}\mathbf{V}^1 = \mathbf{V}^2\mathbf{Q}. \tag{51}$$

Here, **Q** is an orthogonal tensor and both $\mathbf{V^1}$ and $\mathbf{V^2}$ are symmetric positive definite tensors (i.e. symmetric tensors with real and positive eigenvalues). Regions, where **Q** reduces to a plus or minus identity matrix, the field is symmetric. It is also necessary to note that where det(**T**) ? 0, there is a unique correspondence between matrix **T** and the set of matrices {**Q**, $\mathbf{V^1}$, $\mathbf{V^2}$}. If the determinant equals zero, **Q** can not be computed, so visualization methods would have to bypass such a region by e.g. interpolation [10].

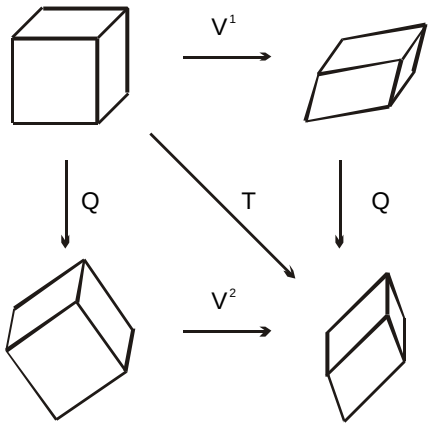


Figure 14: Polar decomposition – two equivalent ways are possible

Deviator-Isotropic Decomposition

Any second order tensor may be decomposed into the sum of a deviator tensor **D** and an isotropic tensor **U**. Symbolically written:

$$\mathbf{T} = \mathbf{D} + \mathbf{U} \tag{52}$$

Where,

$$\mathbf{D} = \begin{bmatrix} t_{11} - \frac{1}{3}q & t_{12} & t_{13} \\ t_{12} & t_{22} - \frac{1}{3}q & t_{23} \\ t_{13} & t_{23} & t_{33} - \frac{1}{3}q \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} \frac{1}{3}q & 0 & 0 \\ 0 & \frac{1}{3}q & 0 \\ 0 & 0 & \frac{1}{3}q \end{bmatrix}. \tag{53}$$

Here, $q = \sum_{i=1}^{3} \mathbf{T}_{ii}$ is the trace of **T**.

The deviator-isotropic decomposition can be applied to tensors in general, or to the symmetric part of a symmetric-antisymmetric decomposition. The deviator has no meaning in the context of the antisymmetric portion of a tensor since the diagonal elements are null.

Fluid Flow Example

Some of the tensor visualization methods make use especially of the symmetric-antisymmetric decomposition, the deviator-isotropic tensor decomposition or their combination. This will be demonstrated on the example of velocity gradient, which must sometimes be investigated when dealing with fluid flows. From this physical quantity, represented by a second order tensor, many other useful quantities can be derived (see Table 3).

The velocity gradient can be obtained using the first order Taylor's series expansion of the velocity at a point:

$$\mathbf{v} = \mathbf{v_0} + \frac{\partial \mathbf{v}}{\partial x}dx + \frac{\partial \mathbf{v}}{\partial y}dy + \frac{\partial \mathbf{v}}{\partial z}dz \qquad \text{or} \qquad \mathbf{v} = \mathbf{v_0} + (\nabla \mathbf{v}) \cdot d\mathbf{r}\,, \tag{54}$$

where

$$\nabla \mathbf{v} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix}. \tag{55}$$

Applying the symmetric-antisymmetric decomposition yields the following tensors:

$$\mathbf{S} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{1}{2}(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}) & \frac{1}{2}(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}) \\ \frac{1}{2}(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}) & \frac{\partial v_y}{\partial y} & \frac{1}{2}(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}) \\ \frac{1}{2}(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}) & \frac{1}{2}(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}) & \frac{\partial v_z}{\partial z} \end{bmatrix} \tag{56}$$

$$\mathbf{A} = \begin{bmatrix} 0 & \frac{1}{2}(\frac{\partial v_x}{\partial y} - \frac{\partial v_y}{\partial x}) & \frac{1}{2}(\frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}) \\ \frac{1}{2}(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}) & 0 & \frac{1}{2}(\frac{\partial v_y}{\partial z} - \frac{\partial v_z}{\partial y}) \\ \frac{1}{2}(\frac{\partial v_z}{\partial x} - \frac{\partial v_x}{\partial z}) & \frac{1}{2}(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}) & 0 \end{bmatrix}. \tag{57}$$

The Taylor's series expansion from (54) can then be also decomposed to

$$\mathbf{v} = \mathbf{v_0} + \mathbf{S} \cdot d\mathbf{r} + \mathbf{A} \cdot d\mathbf{r}\,. \tag{58}$$

From equation (58) we can see that the velocity consists of and can be decomposed into local translation ($\mathbf{v_0}$) plus local rate of strain ($\mathbf{S} + \delta\mathbf{r}$) plus local rigid body rotation ($\mathbf{A} + \delta\mathbf{r}$). The symmetric part ($\mathbf{S}$) has six independent components, three in either the upper or lower triangular matrix plus three components on the diagonal. This multivariate data may be visualized, for instance, by hyperstreamlines (see section Hyperstreamlines in subchapter 4.2). Since the diagonal components of the antisymmetric portion ($\mathbf{A}$) are zero, it has only three independent components. This corresponds to a rotation vector that can be visualized as hedgehogs or ribbons imposed

on visualizations of the symmetric tensor. This technique was described in [10] and will be discussed later in the Hyperstreamlines section.

Before visualization, however, the deviator-isotropic tensor decomposition can be performed to filter out the background isotropic component, which is uniform in all directions and might suppress the deviator [25]. Here, quantity q in the two equations (53) would then equal to $q = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} = \nabla \cdot \mathbf{v}$, which is the velocity divergence. Thus, for the velocity gradient tensor, the deviator corresponds to removing one third of the velocity divergence from each of the diagonal elements.

4.2 Second Order Tensor Visualization

Visualization methods for second order tensor fields will be discussed in this section. As already mentioned, second order tensors are produced by numerous applications in physics and engineering such as fluid flow simulations [10], mechanics and material science but also in seismology [46], biology [24] etc. In this section, we will describe current approaches.

Coloring Coding

These methods offer an aid for understanding the tensor data by displaying its scalar components in a two or three dimensional form. More precisely, the scalar values are mapped to color and then applied on the orthogonal planar slices through the volume. These colored slices are usually presented in a 3 x 3 panel layout. Three dimensional second order tensors consist of 9 scalars, thus one slice corresponds to one of the 9 scalar components (Figure 15). The picture demonstrates that although all the scalar values are depicted, this method does not provide an intuitive understanding of tensors. An inexperienced user will have difficulties to mentally integrate the visual information and interpret its meaning.

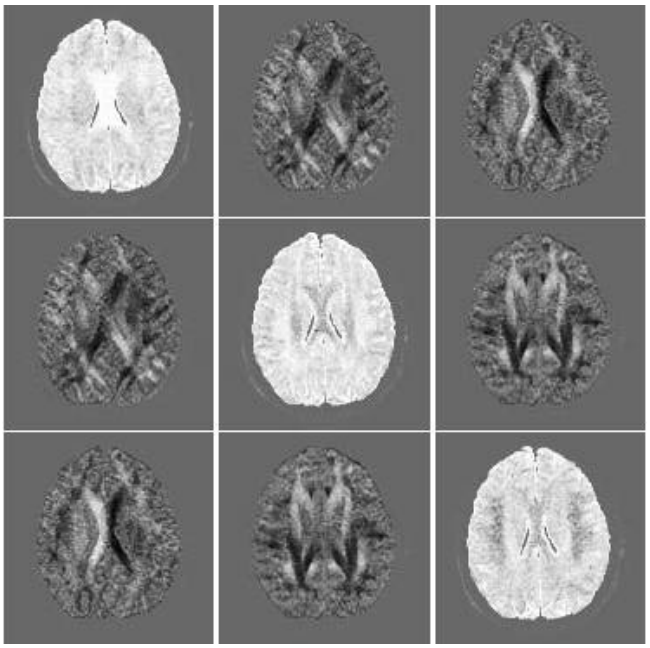


Figure 15: The color coded slices in a 3x3 panel layout [22]

Another way is to apply color coding on the images of the tensor‘s eigenvalues. Although the scalar values in these images are rotationally invariant and they communicate some geometric information, no directional cue is involved [24]. However, the directional information is too valuable to be omitted. From this reason, color coding tensors can not be considered effective.

Tensor Glyphs

These methods depict selected data via simple local icons representing, for example, eigenvalues and eigenvectors at seed positions. Such discrete icons are usually called tensor glyphs and their design and placement need to be done wisely. Glyph choice and seeding are crucial for the understandability and the informative value of the resulting images. Tensor glyphs map tensor information from discrete locations within the field onto a geometric object. As mentioned above, mapping the three eigenvectors as principal axes of an ellipsoid is a good example (Figure 16).

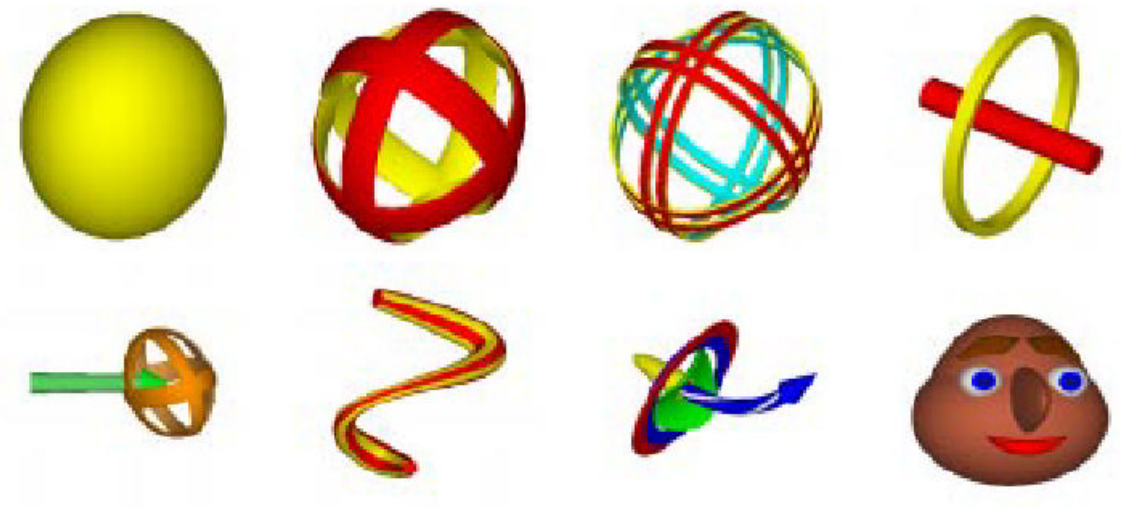


Figure 16: Examples of glyphs to map various quantities on (taken from [40])

Other additional derived information such as shear, convergence/divergence and curvature can be also added onto a flow probe depicted in Figure 17. Another technique of this kind is using deformed cube, which displays a Frenet coordinate frame to show local relative stretch, shear, and rigid body rotation at a point. Glyphs allow the possibility of comprehensive displaying all the tensor information at a particular point. Their discrete nature, on the other hand, does not allow to show the information continuously. Furthermore, improper seeding may cause glyphs overlapping and thus clutter.

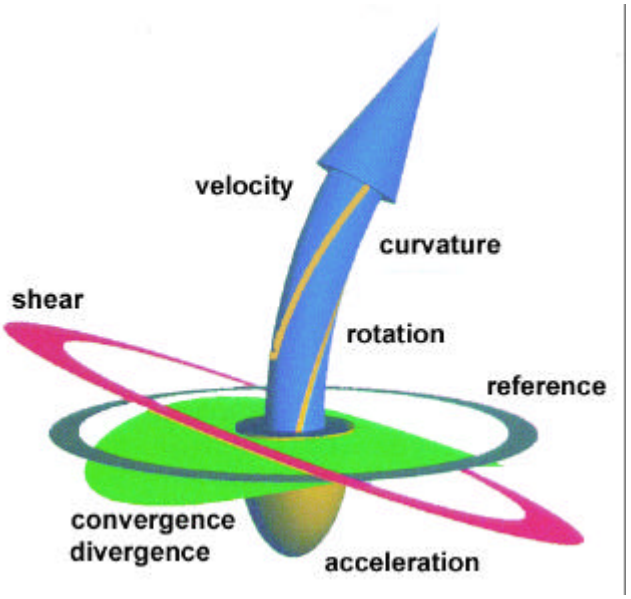


Figure 17: A tensor probe

Hyperstreamlines

Tensor field lines and hyperstreamlines [10] are extensions of vector streamlines into tensor fields. For symmetric tensor fields, the three orthogonal eigenvector components are sorted into largest, median, and smallest eigenvalues. Tensor field lines and hyperstreamlines are then generated by integrating along one of these eigenvector fields. General hyperstreamlines allow the two other eigenvectors and their corresponding eigenvalues to modulate an ellipse along the principal hyperstreamline.

For non-symmetric tensor fields, where the three eigenvector components are not necessarily orthogonal to each other, the tensor field is first decomposed into a symmetric tensor field and an accompanying axial vector as shown in equation (50). Ribbons along the hyperstreamlines are then added to show the rotational effects of the axial vector.

Visualization based upon the eigenvectors and eigenvalues ensures that all the directional and amplitude information will be included in the final result. On the other hand, only one of the eigenvector fields is used for integrating the hyperstreamline. Therefore, there are two other possible hyperstreamlines that can result from a single seed point, so the understanding of the tensor field must be done separately for each eigenvector component. The user must integrate and interpret these three different views mentally.

Topological Approach

This approach aims to provide a global structural representation of the tensor field by first identifying degenerate points (trisectors and wedge points), which are locations, where at least two of the tensor's eigenvalues are equal to each other, and connecting them with topological skeletons (hyperstreamlines). The result of this approach is a display of the important features in the tensor field at the same time showing the continuity (and discontinuities) in the field. Topological tensor field visualization is a direct extension from topological vector field visualization. Morover, as [16] claim, the tensor field topology is often simpler than that of a vector field. While this class of methods draws the user's attention to the special features in the field, the user still has to

mentally reconstruct the rest of the field around these degenerate and critical points and skeletons.

Deformation Visualization

Boring and Pang [3] suggest visualizing symmetric second order tensor fields by letting the tensors deform a geometric object, plane for instance. First, the so called resolute vector of the tensor, which determines the tensor's impact on an interrogation object I with normal $\mathbf{n}$, is computed at all points defining I using equation (48). Where the tensor is unknown, trilinear interpolation is used to approximate it. The resolute vector is then applied. The displacement of point $I(\mathbf{x})$ of the interrogation object to a new position $O(\mathbf{x})$ follows the following rule:

$$O(\mathbf{x}) = I(\mathbf{x}) + s[\mathbf{T}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})], \tag{59}$$

where $\mathbf{x}$ denotes position, s is a scale factor, $\mathbf{T}(\mathbf{x})$ tensor at $\mathbf{x}$ and $\mathbf{n}(\mathbf{x})$ is the user selected normal at the position of $\mathbf{x}$. The product $\mathbf{T}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})$ is the resolute vector. The deformed object illustrating the impact of the tensor field's influence is then visualized using appropriate visualization techniques. Should we need to separate the normal and shear component of the resolute vector $\mathbf{r}(\mathbf{x}) = \mathbf{T}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})$, equation (59) must be modified to

$$\begin{aligned} O(\mathbf{x}) &= I(\mathbf{x}) + s[\mathbf{r}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})]\mathbf{n}(\mathbf{x}) \qquad \text{or} \\ O(\mathbf{x}) &= I(\mathbf{x}) + s[\mathbf{r}(\mathbf{x}) - [\mathbf{r}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})]\mathbf{n}(\mathbf{x})] \end{aligned} \tag{60}$$

for the normal or shear components respectively.

Instead of really deforming the object, it is also possible to keep its geometry untouched but replace its normals with the resolute vectors at the corresponding locations. Adjusting the light position and direction, the areas, where the resolute vectors are shear or normal, can be recognized quickly.

This approach can, in fact, be classified as an attitude based on dimension contraction. By selecting the normal vector $\mathbf{n}$, the problem is reduced to visualization of vector fields. As the field of the resultant vectors is derived from a tensor field and depends on the selected normal vector, common vector visualization techniques might fail to depict all the information, which is why the object deformation is used.

4.3 Higher Order Tensors

For higher order tensor field visualization, contracting the dimension is crucial. A way to do so is choosing certain parameters to eliminate the independent variables, for the depiction of which there would be no more visualization means left. It is similar as in the Deformation Visualization section above, where the user had to choose a specific surface normal to transform the second order (deformation) tensor field visualization problem to a problem of depicting a field of resolute vectors. An example of higher order tensor visualization is given in [23], where a fourth order stiffness tensor studying symmetries of the propagation of waves in different anisotropic crystal class symmetries is described:

$$[\mathbf{C}_{ijkl} \cdot \mathbf{?}_i \cdot \mathbf{?}_j - r \cdot \mathbf{v}^2 \cdot d_{kl}] \cdot \mathbf{p}_k = 0\,. \tag{61}$$

In this equation, vector $\mathbf{p}_k$ is the vibration direction and the two vectors u_1 and u_2 denote the propagation direction of the wave and they are contracted with the fourth order stiffness tensor $\mathbf{C}_{ijkl}$, thus reducing the problem by two orders. For illustration, we can write:

$$(\mathbf{ß}_{kl} - b \cdot \mathbf{d}_{kl}) \cdot \mathbf{n}_k = 0\,, \tag{62}$$

where

$$\mathbf{ß}_{kl} = \mathbf{C}_{ijkl} \mathbf{?}_i \mathbf{?}_j\,, \quad \mathbf{n}_k = \mathbf{p}_k \quad \text{and} \quad b = r\mathrm{v}^2\,. \tag{63}$$

Having performed the contraction, the problem reduces to a second order tensor eigenvalue problem. Note, however, that b_{kl} is not just a simple second order tensor but it derives its properties from a higher order tensor and hence enjoys a much richer surface topology then a simple stress quadric.

For visualization of b_{kl}, tensor glyph is used (Figure 18). In accordance with the statement in the previous paragraph, the depicted glyph is not just a simple quadric like, for instance, the ellipsoids mentioned in section Tensor Glyphs in subchapter 4.2. It must reflect the properties of the fourth order tensor $\mathbf{C}_{ijkl}$. Indeed, the glyph in Figure 18 iconically characterizes all of the components of $\mathbf{C}_{ijkl}$ via its shape (eigenvalues) and color (eigenvectors).

As Kriz et al. claim in [23], sixth order tensors associated with the strain cubed terms of the strain energy density function can also be visualized by modifying equation (61) to include the effect of load induced anisotropy. This problem is also an eigenvalue problem but now with sixth order tensors. With an applied load, the resulting glyph in Figure 18 should deviate from a symmetric shape. This shape change not only represents a change in elastic anisotropy but should also show the load direction to the viewer.

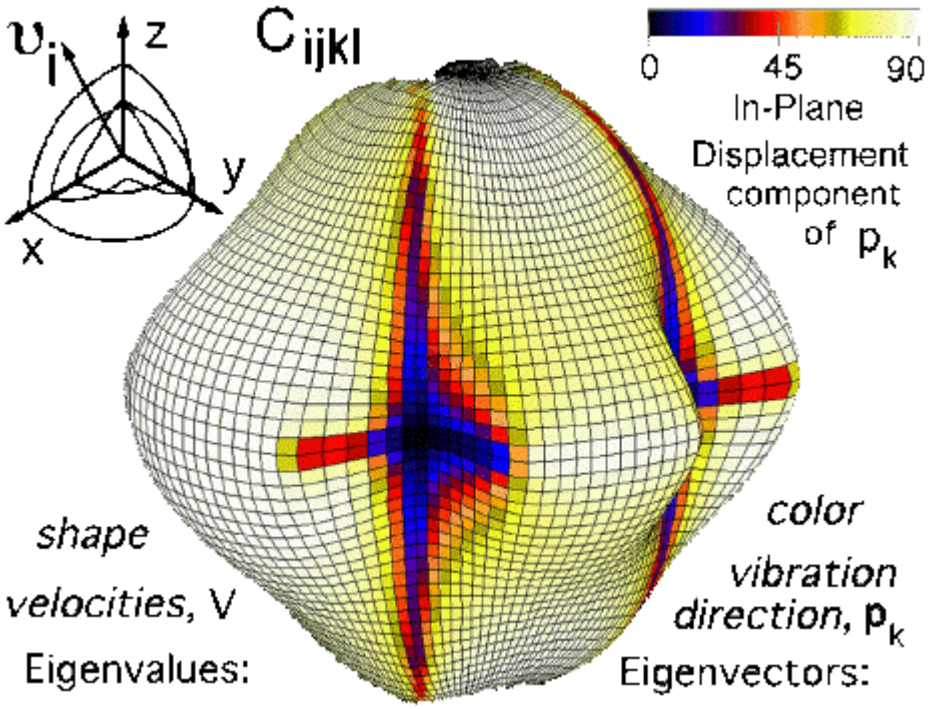


Figure 18: A glyph for visualizing higher order tensors

5 MULTI-SCALAR DATA

The last type of multi dimensional data, the visualization of which will be discussed in this work is multi-scalar information. In other words, the following paragraphs briefly describe, how to visualize datasets consisting of several scalar fields put together. An example of such data is a field containing pressure, density and temperature at each sample point. In the case of multi-scalar data, no explicit relation among the values is usually known. Therefore, the visualization methods concentrate particularly on producing views, which would reveal the existence and the character of such relations.

5.1 Parallel Coordinates

The parallel coordinates technique, presented in [18], differs from all the other approaches in its nature, because it uses parallel coordinates for depiction instead of orthogonal ones. Although we will not describe it in detail, nor its further extensions, yet it has to be mentioned here.

The underlying idea is mapping an nD space on a 2D surface, where each dimension corresponds to one vertical axis, as can be seen in Figure 19.

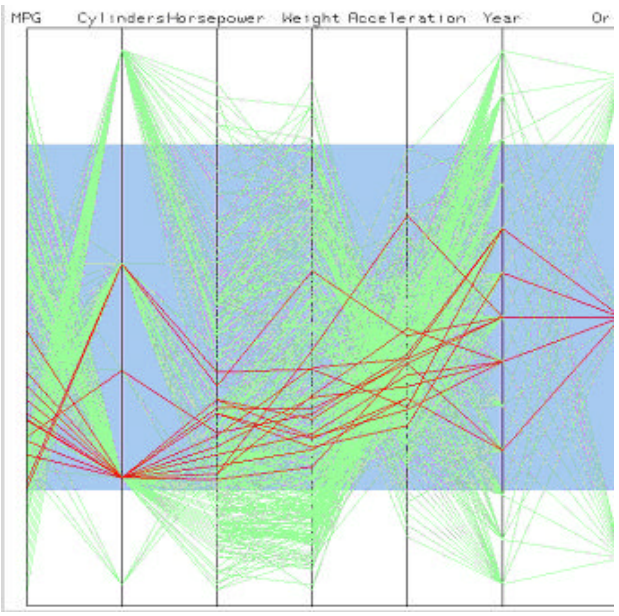


Figure 19: Depiction using parallel coordinates

To understand the outcomes of this approach, the user has to cognitively investigate the results. Yet, this technique is very helpful in searching for relations in the data. If we look at Figure 20 and imagine that the left co-ordinate represents temperature while the right one pressure, then the left image depicts three data samples, for which it holds that the higher the temperature, the higher is the pressure. The right image would, on the hand, mean that in case of the five samples depicted, the temperature and pressure are inversely proportional. Moreover, the color coding may imply, whether the values fall into certain user defined interval or not.

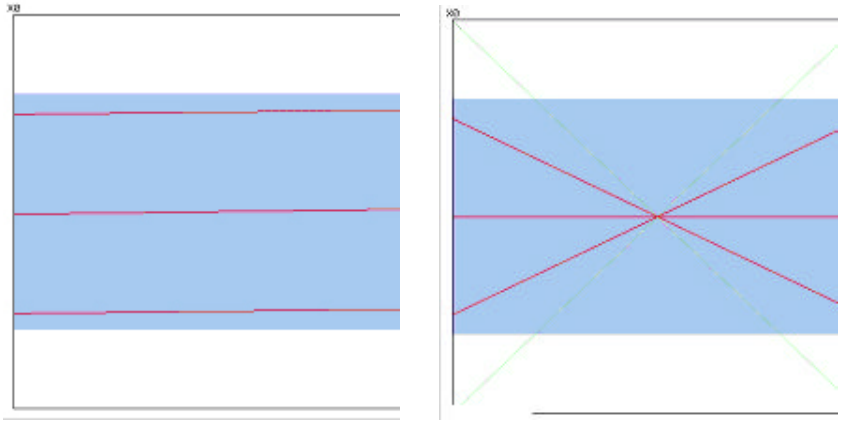


Figure 20: Using the parallel coordinates

This approach is not intended to observe the data as a unit but rather for revealing relations hidden within. The main disadvantage consists in the fact that with increasing amount of visualized information, occlusion may appear and the image quality decreases.

5.2 Color Coded Isosurfaces

Another interesting approach to visualizing multiple scalar fields at the same time and searching for relations among the quantities was proposed in [23]. A single scalar field is usually depicted by extracting isosurfaces or volume rendering methods. This can be done for each value independently, but the results will not show any information about potential correlation between them. This approach tries to bypass this limitation.

The method can be explained using, for example, seven parameters (P_1 through P_7), which can be then cognitively compared in the same visual space where four of the seven properties P_1, P_2, P_3 and P_4 are chosen as independent variables (not necessarily coordinate space and time). Hence this method provides a common basis from which to test for the existence of relationships between the remaining properties P_5, P_6, and P_7.

Figure 21 illustrates the procedure. The first three parameters P_1, P_2 and P_3 are independent (orthogonal) variables that are visually defined as perpendicular axes in this figure. The fourth property is reserved as another independent variable, e.g. P_4 = time, that is uniform everywhere, but cannot be drawn as the orthogonal fourth axis. The task remains to find relationships, if any, of the remaining properties P_5, P_6, and P_7 that must all be functions of P_1, P_2, P_3, and P_4. For the simplest case this method can be reduced to a single function, where P_4 is assumed to be constant everywhere. On the other hand, this method can also be generalized to more than three functions i.e. for P_8, P_9, ..., P_n, where *n* is “the viewer’s cognitive limit”.

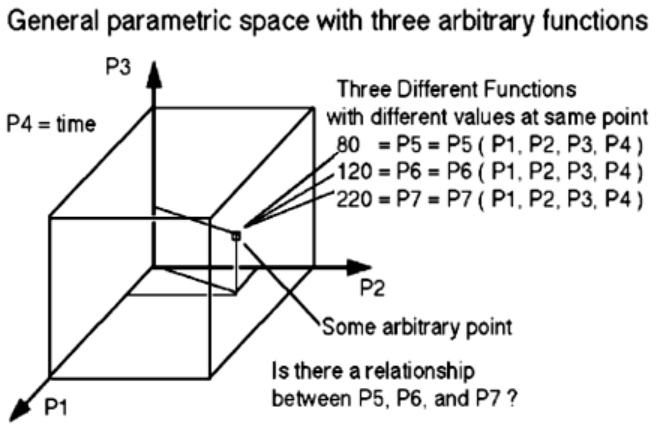


Figure 21: Illustration of the relations between the functions

The method for searching the relation between P_5, P_6 and P_7, starts at some point, where these quantities reach unique values Figure 21. In the vicinity of this point, the values will differ with certain gradient. Although all possible values, normally visualized through direct volume rendering, can not be seen in the same region for all three functions, a quantitative isosurface can be seen for each function as a separate shaded surface. The change of the surfaces' shapes following a slow change of the isovalue would then give a gradient notion as well.

The mutual relation between the properties is then reached through drawing two (e.g. P_5 and P_6) of the three properties as two unique intersecting isosurfaces as shown in Figure 22. If the surfaces do not intersect, there can be no relationship between the functions. If they do, further investigation has to be done to find the nature of the relation. This investigation, however, is rather experimental. The user has to guess the mathematical relation from the visual pattern the functions show. The authors describe a case for detecting linear proportionality and inverse proportionality of P_5, P_6 and P_7, i.e. searching for the relation corresponding to the expression $P_5 \cdot P_6 \cdot P_7 = const$.

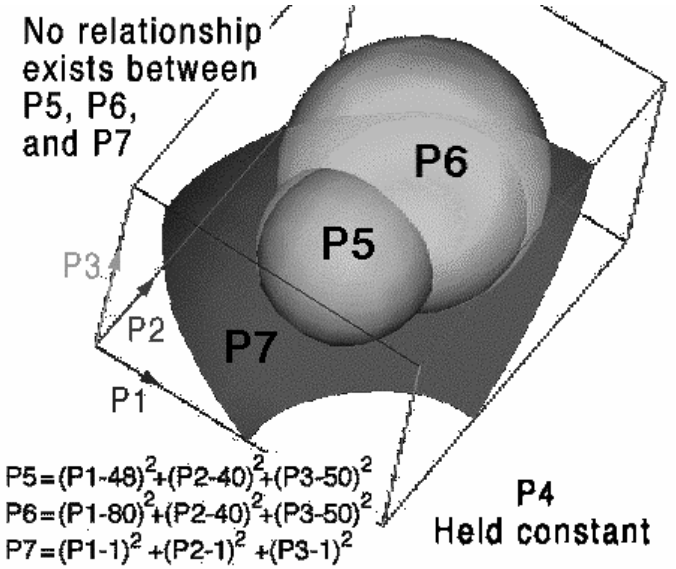


Figure 22: No relation between P5 and P6

Such relation is identified, when P_7, mapped as color on the two isosurfaces (i.e. isosurfaces of P_5 and P_6), has constant shade around the curve, where P_5 and P_6 intersect

Figure 23. Having found such pattern, the user must still find out, which of the following three situations turned up:

$$P_5 \cdot P_6 \cdot P_7 = const.,\qquad P_5 \cdot P_6 = const. \cdot P_7,\qquad P_5 = const. \cdot P_6 \cdot P_7. \tag{64}$$

This is done by varying individual isovalues and observing the result.

The authors claim that with this method it is possible to successfully find many more complex functions. In all cases, just as in finding solutions to differential equations, the user must guess at possible solutions, as already mentioned. Of course only significant functional components will be detected and extracted much like the dominant terms in a series solution. Mentally the observer first sees a pattern related to the function and can then deduce the function mathematically. Hence the pattern of a function occurs first and becomes the cognitive mechanism that allows the investigator to confirm the existence of suspected functional relationships.

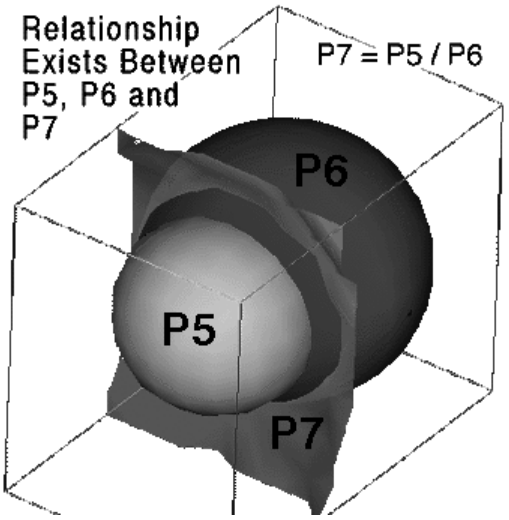


Figure 23: In this case, the three function are related

Although far from accurate and reliable, this method may be an interesting tool for observing dependencies in multi-scalar data. The advantage is that for seeing the relation, functions P_5, P_6 and P_7 do not have to be reconstructed. Also, the user gets a notion of the behavior of the whole system.

6 ISOSURFACES, NORMALS AND GRADIENTS

Although isosurface extraction was originally developed for scalar volumetric data visualization, as can be seen from the chapters above, this well-known technique can be successfully applied on multidimensional data as well, especially in combination with dimension contraction. We have shortly studied this field and we have found the motivation here to experiment with the issues of accuracy and temporal demands of computing normal vectors in the vertices of triangle meshes representing the isosurfaces. This has also brought us to examining methods for gradient estimation and proposing an improvement. In the following three subchapters, this work will be outlined.

6.1 Isosurface Extraction

As mentioned above, we have had some experience with isosurface extraction. In this chapter, however, we will not present a study about known methods and techniques. The focus of our effort was vertex normal computation and gradient estimation, so we will only outline a short classification of the isosurface extraction methods, explain our motivation for computing normals and gradients and we will move to these topics.

Fundamental Algorithms

Marching cubes and marching tetrahedra stand for the fundamental isosurface extraction algorithms. They were designed for regular grids and, although they have experienced a lot of modifications over time, the original versions will be briefly outlined here.

Marching Cubes (MC)

Probably the best known technique for isosurface extraction is marching cubes. It was developed for volumetric data organized to a regular Cartesian grid. The algorithm marched through all the cubic cells in the grid one after another, compared the values at the cell's vertices with a user selected isovalue and recognized, whether the isosurface intersects the cell and how. For this purpose, a table of 256 possible cases of interaction between an isosurface and a grid cell was introduced. Exploiting the symmetry of the cells, this table was later reduced to 16 items. The drawback of this method were holes, which sometimes appeared on the surface due to ambiguous meaning of some of the interaction cases. This method has experienced many improvements.

Marching Tetrahedra (MT)

One of the attempts to fix the problems with holes was subdividing the cubic cells into smaller, tetrahedral ones utilizing a pattern, which ensures that two adjacent tetrahedra will share either nothing, a vertex, a whole edge or a whole facet. The situation with two tetrahedra sharing only a part of an edge or facet had to be avoided. Having decomposed the grid cell, marching tetrahedra was used instead of marching cubes. The advantage is that there are only three possibilities, how an isosurface can intersect a tetrahedral cell. The disadvantage, on the other hand, was that too many triangles were produced.

Optimized Isosurface Extraction

One of the reasons, why the MC and MT algorithms became so famous, might have been their simplicity. On the other hand, with increasing sizes of the volumetric datasets, some optimized methods had to appear. The broadness of this topic does not allow to provide a survey of these methods here. We will therefore only outline the basic concepts.

These optimized methods are mostly based upon the fact that just a fraction of the grid cells is usually intersected by the isosurface. Ways, how to identify these cells without examining the whole grid, are different for each of these methods. This is difficult to do in the common *geometric space*. Yet, some approaches appeared, which fist located one intersected cell, extracted isosurface patch from it and then examined the neighboring cells, thus moving along the isosurface. Discovering all the parts of the isosurface was, however, not assured by this approach.

An alternative attitude to the common *geometric* space is the, so called, *value* space. Techniques utilizing this attitude start with a preprocessing step, during which they determine for each cell the minimum and maximum values associated to the cell's vertices. From this point, cells are treated according to the range of values, they cover, rather then according to their geometric coordinates, from which the term *value* space arises. A cell, which is intersected by an isosurface must have its maximum value higher, than the threshold while the minimum value must be lower. Thus, if the cells are sorted according to these extreme values and stored in some convenient data structure, they can be identified quickly without the need to investigate the unintersected cells.

The examples of the value space techniques, which exploit various data structures to optimize the search for the intersected cells are Sweeping Simplices [49], Interval Tree [9] and Span Space [29].

Isosurface Shading

Our work in this field was focused in a little different direction than what the above paragraphs describe. Obviously, once the isosurface is extracted, it also has to be rendered. To be able to employ Gouraud rendering technique, normal vectors in the vertices must be known. We have thus focused on how to get these normal vertices as precise as possible, so that smooth look of the rendered object can be reached (see section 6.2).

The other goal arises from the fact that [42] the gradient vector $\nabla f = \frac{\partial f}{\partial x}\vec{i} + \frac{\partial f}{\partial y}\vec{j} + \frac{\partial f}{\partial z}\vec{k}$ in location $Q[x_0, y_0, z_0]$ of the function $f(p)$, which describes the values of a scalar field, is perpendicular to the field's isosurface passing through $Q[x_0, y_0, z_0]$. The gradients might therefore be pre-computed in the preprocessing step and then, during the extraction, just interpolated to form the vertex normals of the extracted surface. This requires the gradient vectors to be estimated exactly enough including their length and not only the direction (see section 6.3).

6.2 Vertex Normal Computation

In the following paragraphs, three methods for vertex normal computation will be described, tested and compared. In [iii], we studied two more approaches. Unlike the techniques described bellow, they were not restricted to triangle meshes. On the other hand, however, they could only be applied on surfaces, which can be defined as $z = f(x, y)$ and it is not the case here. Although other techniques for surface normal

computation exist as well, some of which are briefly described in the appendix of [iii], we will restrict to methods focusing precisely on estimating normal vectors in the vertices of a general triangle mesh.

6.2.1 Theoretical Background

All the three methods for triangle mesh vertex normal computation discussed bellow share the basic idea. They all compute the vertex normals by combining normal vectors of the triangles adjacent to the vertex being computed. While the basic idea is common for all the three methods, the difference consists in how the triangle normals are weighted.

There is one more aspect to be pointed out. The purpose of using these methods is to make the surface shaded smoothly and to avoid those edges between adjacent polygons that do not occur on the original object and that are caused by the surface approximation by the polygonal mesh. However, the rendered body may also contain some real edges. Smoothing out these edges would rather decrease than increase the realism of the output image. The real edges therefore should be rendered. If these edges are not marked within the input data, the rendering algorithm should attempt to recognize them. The method that can help to overcome this drawback is based on defining certain "decision angle". If the angle between two adjacent polygons is less sharp than the decision angle, the edge is considered to be an artifact and is smoothed. Otherwise, the edge probably represents a real edge on the rendered object and should be displayed sharply.

No Weighting

This method has been described in [15] by Gouraud, who suggests computing normals in the vertices of a triangle mesh as the average of the normal vectors of the facets that share the vertex being computed. In his approach, all the facets, which contribute to the vertex normal computation, are weighted equally. Mathematically,

$$N = \frac{\sum_{i=1}^{n} N_i}{\left| \sum_{i=1}^{n} N_i \right|}, \tag{65}$$

where N is the normal in the vertex and N_i are the normals of the *n* triangles that share it.

Weighting by Angle

Thurmer and Wuthrich [55] aim to improve the accuracy of the vertex normal computation method suggested by Gouraud. They claim that the results of the Gouraud's method strongly depend on the topology of the mesh around the vertex being processed. In other words, if we start with certain triangle mesh, choose one of its vertices and compute the normal vector there, then if we restructure the surrounding of this vertex and then we re-compute the vertex normal, the result should ideally be the same. By restructuring the vertex surrounding we mean using a different triangulation upon the same set of vertices. This can be reached for example by adding new vertices on the edges of some of the existing triangles thus dividing these triangles into two or more smaller ones while keeping the overall shape of the surface untouched (see Figure 24).

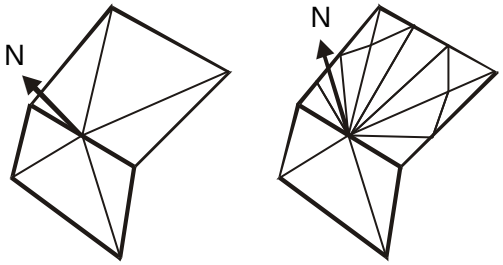


Figure 24: Illustration of the normal's dependency on the meshing

The authors of this article try to reach the independency on the mesh structure through weighting the contribution of each facet's normal by the size of the angle enclosed by the facet's edges incident to the computed vertex. This can be expressed as

$$N = \frac{\sum_{i=1}^{n} \alpha_i N_i}{\left| \sum_{i=1}^{n} \alpha_i N_i \right|}, \tag{66}$$

where α_i is the angle between the two edges of the facet that lead to the vertex.

Weighting by Area

In this method, the normal N is computed as a normalized weighted sum of the unit length normal vectors N_i of the facets that belong to the cycle around the vertex being processed similarly as in Gouraud's [15] and Thurmer's [55] approach. This time, each facet's area S_i is considered as the weighting function for the corresponding normal. Thus, the larger facets have higher influence than the smaller ones. Symbolically expressed:

$$N = \frac{\sum_{i=1}^{n} S_i N_i}{\left| \sum_{i=1}^{n} S_i N_i \right|}, \tag{67}$$

The idea of area weighting comes from [64] (p. 149) and it serves for the computation of a normal vector of a surface interleaved among points, which generally do not lie in one plane. The purpose of including this method was to find out, how accurate it would be to consider this normal to be a normal in the vertex, whose direct neighbors are interleaved by the surface.

6.2.2 Implementation

Besides other, all the above-described methods for vertex normal computation were implemented within one application developed under the Borland Delphi environment and the tests were run on a system with the Intel Pentium III @ 448MHz CPU and 1024MB RAM.

Testing Data

The testing data for the vertex normal computation were produced via the MVE[2] system using the *DTLib* module. For each of the three kinds of 2D triangle meshes available in this module (i.e. regular meshes, irregular meshes with uniformly distributed vertices and irregular meshes with vertices distributed randomly), input files were generated for 1,000 / 10,000 / 100,000 / 250,000 / 500,000 / 750,000 and 1.000,000 vertices.

For the computation of the z coordinate, three different functions were used. These being:

- $f_1(x, y) = \sqrt{2 - x^2 - y^2}$
- $f_2(x, y) = x^2 + 0.25 \cdot \sin(2 \cdot p \cdot y)$
- $f_3(x, y) = 1 + 0.25 \cdot \sin(p \cdot x) + 0.25 \cdot \cos(p \cdot y)$

The examined vertex normal estimating methods can be applied to various triangle meshes, which have no common characteristic. Thus there is no principal criterion to be used for designing the testing functions for the z coordinate generation. Since $x, y \in \langle 0,1 \rangle$ for all the vertices of the generated 2D triangle mesh, the only limitation is that the function f must be defined for any point **p** from this region (i.e. $\forall \mathbf{p} \in \{\mathbf{p}[x, y] \in R^2 : x, y \in \langle 0,1 \rangle\}, \exists z \in R : z = f(\mathbf{p})$). All the three functions listed above fit this condition. When choosing the functions, the aim was to test the methods on surfaces that are curved just slightly as f_1, as well as surfaces, where the curvature changes quite a lot f_3. Function f_2 is something in between.

For the testing purposes, the boundary vertices of the mesh were not included in the statistics.

6.2.3 Results

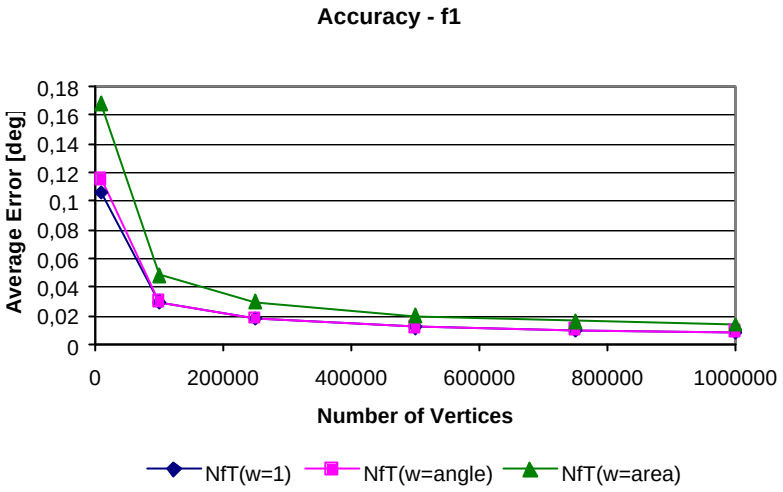
Notation

In the following text, the three methods described in 6.2.1, which compute vertex normals from adjacent triangles' normal vectors, will be marked as NfT(w=1), NfT(w=angle) and NfT(w=area) respectively.

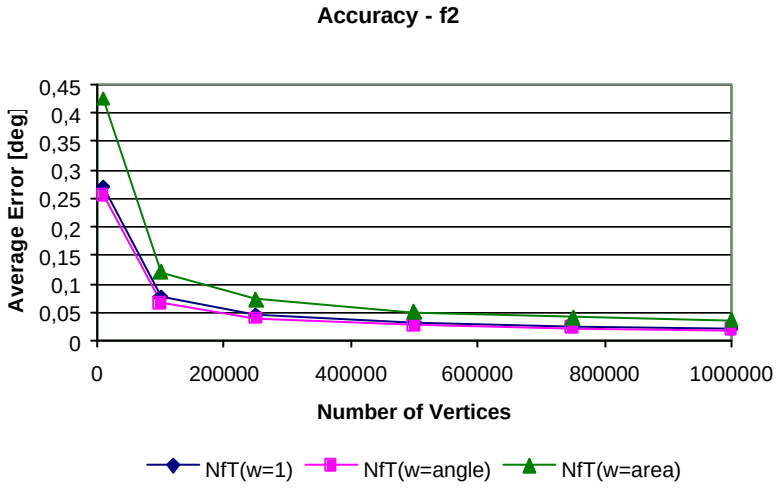
Accuracy Statistics – Varying the z-Function (Surface Shape)

One of the important aspects, which influence the accuracy of the estimation of the triangle mesh vertex normal, is the shape of the examined surface. As one might intuitively expect, it is easier to estimate the normal vector in a vertex of a plane or some slightly wavy surface than to estimate such normal for strongly curved meshes. To confirm or disconfirm this belief experimentally, the examined methods were tested on surfaces constructed from planar 2D meshes by defining the z coordinate via the functions listed above. Each of the three graphs displayed below describes the average errors produced by individual methods when applied on the three differently curved surfaces. As expected, most precise results were obtained on the surfaces produced by f_1, where the curvature was minimal. For f_3, on the other hand, the average measured errors were approximately four times as big. The function f_2 appeared to be half way between f_1 and f_3.

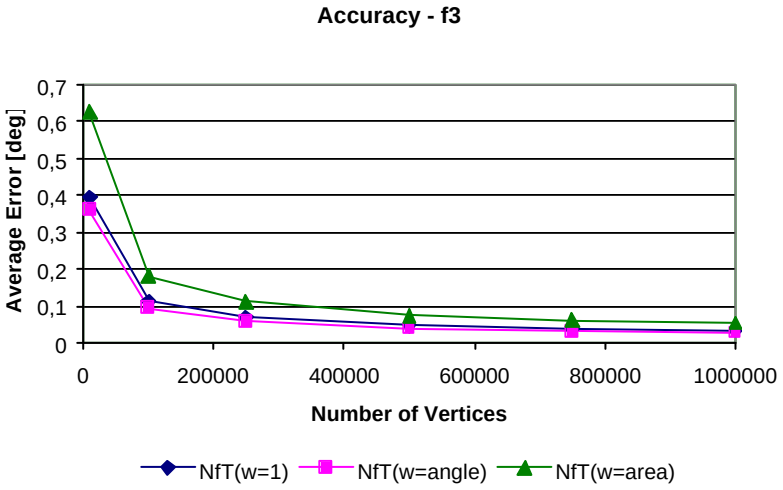
[2] MVE (Modular Visualiztion Environment) was developed by the Centre of Computer Graphics and Data Visualization at the Department of Computer Science, University of West Bohemia in Pilsen.



Graph 1: The estimation errors for f_1



Graph 2: The estimation errors for f_2



Graph 3: The estimation errors for f_3

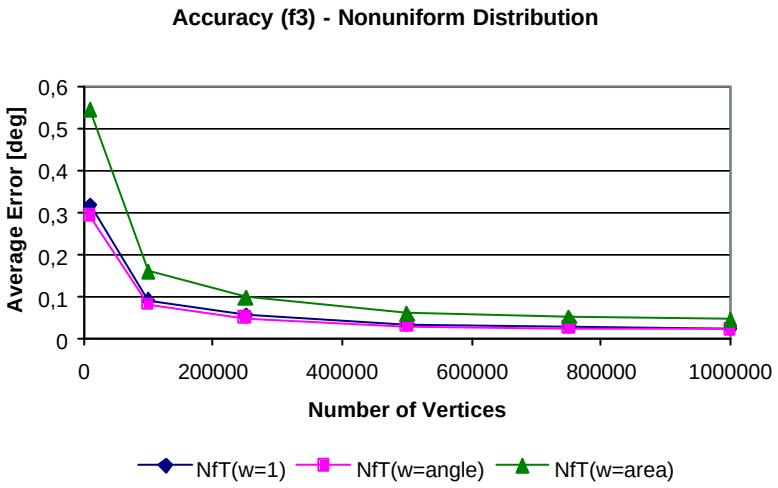
Another aspect that plays an important role is the density of the mesh. When the number of vertices increases, more information is contained in the mesh, smaller regions are used for the estimation and more precise results can be obtained. This fact is also illustrated by the three graphs above.

However, the most important fact these measurements reveal is that, regardless of the number of vertices or the function used, the best results were obtained using the NfT(w=1) and NfT(w=angle) methods, which scored rather equally. We point out the fact that weighting by area performs significantly worse than no weighting at all, which implies that using the facet area as a weighting function for computing vertex normals decreases rather than increases the resulting precision.

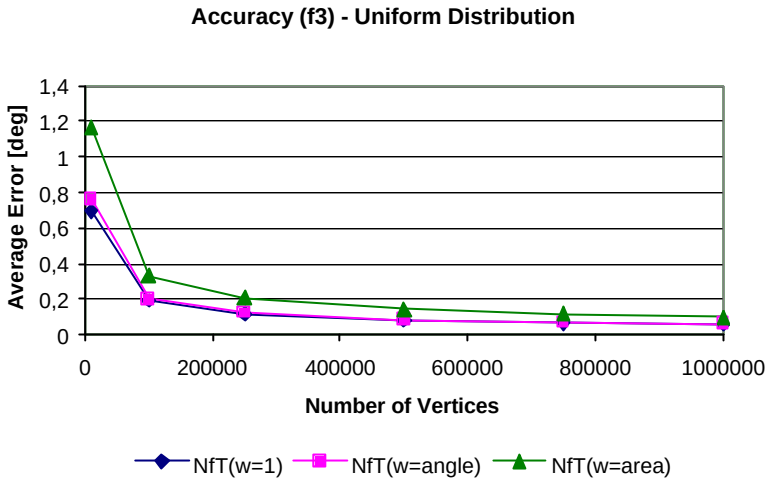
Accuracy Statistics – Varying Vertex Distribution (Surface Internal Structure)

In the previous section, the influence of the overall shape and the density of the mesh on the accuracy of the vertex normal computation was examined. Here, concern will be put on the internal structure of the mesh and its relation to the accuracy of the vertex normals computed by individual tested methods.

For these tests, meshes constructed upon non-uniformly, uniformly or regularly distributed vertices were created. The following two graphs show that using meshes with randomly distributed vertices, whether uniformly or not, does not make a big difference from the point of view of which method performs better.



Graph 4: Estimation from meshes with non-uniform vertex distribution.

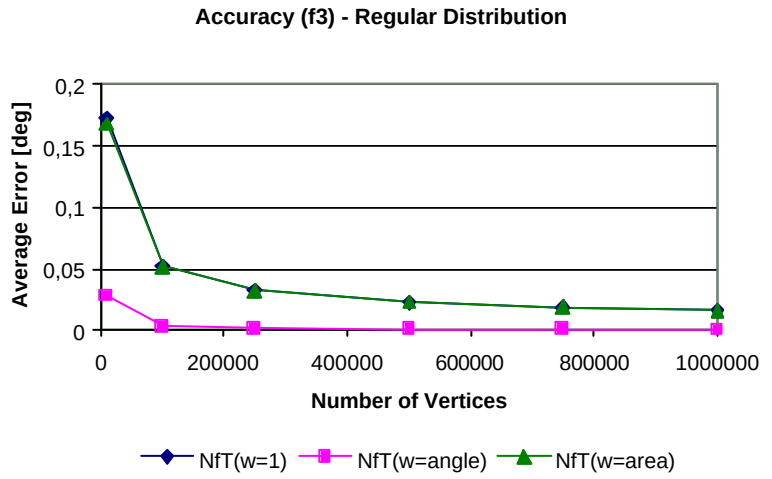


Graph 5: Estimation from meshes with uniform vertex distribution.

Note, however, that the precision was better for all the methods using the non-uniform distribution, regardless of the number of vertices the mesh consisted of.

Furthermore, if we start with a 2D mesh containing regularly distributed vertices (see Figure 25), then we can see that weighting by angle brings significantly better results than the other two methods. For correctness, it is necessary to point out that after the transformation to 3D by applying one of the three functions listed in section 6.2.2, the vertex distribution will not be regular any more. Yet, the character of the mesh will be preserved, which is sufficient for our purpose.

Graph 6 describes the average error of vertex normal vector estimation on a mesh with such regular vertex distribution. For this purpose, function f_3 was used, but as other measurements have shown, using the other functions leads to similar results. Thus the graph tells us, that for this type of meshes, the method of weighting triangle normals by angle brings a significant improvement, as compared to the standard Gouraud technique. NfT(w=1) worked with roughly the same quality as NfT(w=area).



Graph 6: Estimation from meshes with regular vertex distribution.

The reason resides in the structure of the original 2D mesh. Figure 25 shows a real example of such mesh. Although the vertex distribution is regular, the triangulation of

the vertices is not. On the contrary, there are only a few vertices, the nearest surrounding of which is triangulated symmetrically. If we recall the principle of NfT(w=angle) as described in the Theoretical Background, it is obvious that the computation of vertex normal vectors from facet normals without using any weighting function, which is the case of NfT(w=1), does not take the constellation of the surrounding polygons into account. Although the transformation into 3D deforms the mesh partially, it is not surprising that NfT(w=1) produces similar results as NfT(w=area) for such mesh, since all the triangles cover roughly the same area.

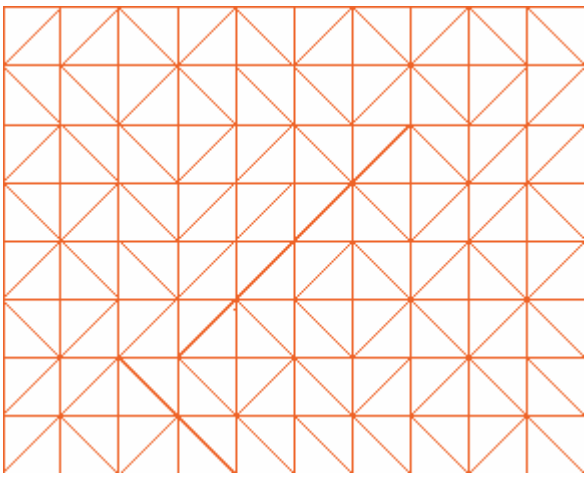
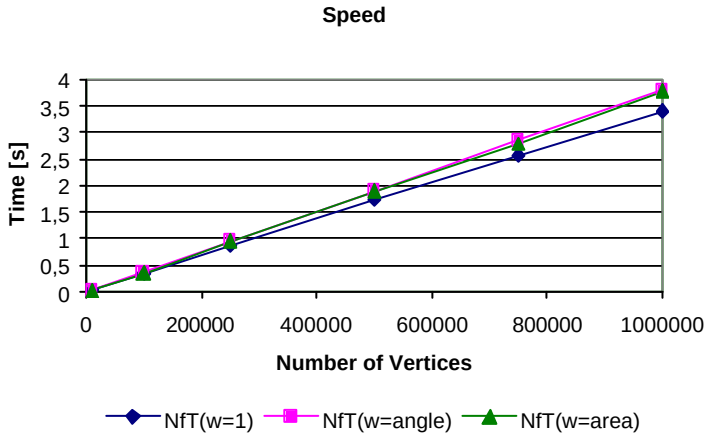


Figure 25: An example of a 2D mesh with a regular vertex distribution.

Speed Statistics

The time requirements of individual methods are illustrated by Graph 7. Obviously, all the three algorithms performed roughly the same concerning temporal demands, the subtle differences were caused by the need to compute the additional information for weighting.



Graph 7: The time requirements of the tested methods.

6.2.4 Conclusion & Recommendations

In the preceding paragraphs, three methods for computing triangle mesh vertex normal vectors from the normals of the adjacent polygons have been described, implemented and their results were compared with focus on accuracy. Considering all the information gathered about the behavior of the tested methods, using angle weighted normals, proposed by Thurmer and Wuthrich, seems to be the best solution. Although in some cases, it performs the same as Gouraud, in other (depending partially on the structure of the input mesh) it works better, yet without any significant temporal penalty.

6.3 Gradient Estimation

Our motivation to deal with gradient estimation has already been described in section Isosurface Shading in subchapter 6.1. Here, we will present an extension of the 4D linear regression method for the scalar irregularly distributed volumetric data gradient estimation. The aim is to reach higher accuracy and the main tool is using quadratic regression function. The results will be compared to the original method as well as to the approach based on the generalization of the finite differences method presented in [34]. The performance of all the three methods will be examined from different points of view.

6.3.1 Theoretical Background

In the following paragraphs, the principles of 4D linear regression method for gradient estimation will be described and the approach utilizing quadratic approximation function for the linear regression will be proposed.

4D Linear Regression using Linear Approximation Function

This method for gradient estimation from regular as well as irregular volumetric data proposed in [37] tries to find a 3D regression hyper plane $f(x,y,z) \approx A \cdot x + B \cdot y + C \cdot z + D$ with minimal error. The error function is represented as the summed squares of the difference between the original values in the interpolated vertices and the values that the solution of the hyper plane equation would give in these points. Mathematically:

$$E(A,B,C,D) = \sum_{k=0}^{n} w_k (A \cdot x_k + B \cdot y_k + C \cdot z_k + D - f_k)^2 \,, \qquad (68)$$

where x_k, y_k and z_k are the coordinates of the vertices involved in the approximation (the computed vertex being considered as the origin of the coordinate system) and f_k are the values in these points. A, B and C make up the vertex gradient that we search for and D is the filtered value in the computed vertex. The w_k symbol represents the weighting function, which should be spherically symmetric and monotonically decreasing as the distance from the origin (i.e. from the computed vertex) grows.

To minimize the error function E, its partial derivatives along A, B, C and D must be equal to zero:

$$\frac{\partial E}{\partial A} = 2 \cdot \sum_k w_k \cdot (A \cdot x_k + B \cdot y_k + C \cdot z_k + D - f_k) \cdot x_k = 0 \,,$$

$$\frac{\partial E}{\partial B} = 2 \cdot \sum_k w_k \cdot (A \cdot x_k + B \cdot y_k + C \cdot z_k + D - f_k) \cdot y_k = 0 ,$$

$$\frac{\partial E}{\partial C} = 2 \cdot \sum_k w_k \cdot (A \cdot x_k + B \cdot y_k + C \cdot z_k + D - f_k) \cdot z_k = 0 ,$$

$$\frac{\partial E}{\partial D} = 2 \cdot \sum_k w_k \cdot (A \cdot x_k + B \cdot y_k + C \cdot z_k + D - f_k) = 0 .$$

This system of simultaneous linear equations can be rewritten in a matrix notation the following way:

$$\begin{bmatrix} \sum w_k x_k^2 & \sum w_k x_k y_k & \sum w_k x_k z_k & \sum w_k x_k \\ \sum w_k x_k y_k & \sum w_k y_k^2 & \sum w_k y_k z_k & \sum w_k y_k \\ \sum w_k x_k z_k & \sum w_k y_k z_k & \sum w_k z_k^2 & \sum w_k z_k \\ \sum w_k x_k & \sum w_k y_k & \sum w_k z_k & \sum w_k \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} \sum w_k f_k x_k \\ \sum w_k f_k y_k \\ \sum w_k f_k z_k \\ \sum w_k f_k \end{bmatrix} . \qquad (69)$$

Solving the system for *A, B, C* and D gives the hyper plane normal vector, which is considered to be the estimation of the gradient analytically defined as $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$.

4D Linear Regression using Quadratic Approximation Function

In order to reach higher accuracy of estimated gradient vectors, it is necessary to apply a nonlinear approximation function. In our approach we use a general quadratic function of the following form:

$$g(x, y, z) = [x, y, z, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} , \qquad (70)$$

instead of the original linear function $f(x, y, z) \approx A \cdot x + B \cdot y + C \cdot z + D$. For the further description, the non-matrix notation will be more illustrative:

$$g(x, y, z) = A_{11} x^2 + A_{12} xy + A_{13} xz + A_{14} x + A_{22} y^2 + A_{23} yz + A_{24} y + A_{33} z^2 + A_{34} z + A_{44} .$$

Now we need to express the error function:

$$E(A_{11}, \ldots, A_{44}) = \sum_k w_k (A_{11} x^2 + A_{12} xy + A_{13} xz + A_{14} x + A_{22} y^2 + A_{23} yz + A_{24} y + A_{33} z^2 + A_{34} z + A_{44} - f_k)^2$$

and find the partial derivatives according to all the ten unknown parameters A_{11} through A_{44}:

$$\frac{\partial E}{\partial A_{11}} = 2 \cdot \sum_k w_k (A_{11} x_k^2 + A_{12} x_k y_k + A_{13} x_k z_k + A_{14} x_k + A_{22} y_k^2 + A_{23} y_k z_k + A_{24} y_k + A_{33} z_k^2 + A_{34} z_k + A_{44} - f_k) \cdot x_k^2 ,$$

$$\frac{\partial E}{\partial A_{12}} = 2 \cdot \sum_k w_k (A_{11} x_k^2 + A_{12} x_k y_k + A_{13} x_k z_k + A_{14} x_k + A_{22} y_k^2 + A_{23} y_k z_k + A_{24} y_k + A_{33} z_k^2 + A_{34} z_k + A_{44} - f_k) \cdot x_k y_k ,$$

.
.
.

$$\frac{\partial E}{\partial A_{44}} = 2 \cdot \sum_k w_k (A_{11}x_k^2 + A_{12}x_k y_k + A_{13}x_k z_k + A_{14}x_k + A_{22}y_k^2 + A_{23}y_k z_k + A_{24}y_k + A_{33}z_k^2 + A_{34}z_k + A_{44} - f_k) \cdot 1 .$$

These partial derivatives must be equal to zero, thus we get a 10 x 10 matrix describing the set of simultaneous equations, which are linear in respect to the A_{11} through A_{44} parameters.
The gradient of the function can be described by the following formula:

$$\nabla g(x,y,z) = (\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial z}) = (2 \cdot A_{11}x + A_{12}y + A_{13}z + A_{14}; 2 \cdot A_{22}y + A_{12}x + A_{23}z + A_{24}; 2 \cdot A_{33}z + A_{13}x + A_{23}y + A_{34}) . \quad (71)$$

As the active vertex is always shifted to the origin of the coordinate system, the *x*, *y* and *z* coordinates are zero. Thus computed the A_{11} through A_{44} parameters, the gradient vector can be obtained from a simple formula:

$$\nabla g(0,0,0) = (A_{14}, A_{24}, A_{34}). \quad (72)$$

6.3.2 Implementation & Testing

Both the above-described approaches were implemented within one application. Moreover, as announced in the Introduction section, the 3D version of the gradient estimation method based on the generalization of the finite differences method, presented in [34], was implemented for the purpose of comparison. All the implementations were done in the Borland Delphi environment and the tests were run on a system with the Intel Pentium III @ 448MHz CPU and 1024MB RAM.

Testing Data

The application requires the volumetric data to be structured to constitute a tetrahedra mesh whether regular or not. The tetrahedra structure only serves to determine each vertex's surrounding, which should be involved in the computation, and is not necessary for the approach itself.

Our tests have been performed on meshes constructed upon the sets of 5000, 10000, 15000 and 20000 vertices using Delauney approach, maximal number of tetrahedra and minimal number of tetrahedra. To show, how the mesh structure influences the results, estimations from meshes constructed upon clusters of vertices have also been tested.

To be able to make comparisons and evaluations we need the exact gradient vectors. Thus it is necessary to use some known function to generate the scalar field values. However, the estimation methods are meant to search for gradients of general data with no common characteristic known in advance (e.g. empirically measured data). Therefore, there was no definite criterion for choosing the testing function. The strategy was chosen to test the methods on some simple function (i.e. f_1 – see below), then on some simple function (i.e. f_2) with higher order then the order of the approximation functions used in the estimation method and eventually on a relatively complex function (i.e. f_3), which would be rather distant to those approximation functions thus at least partially substituting the empirically obtained data:

- $f_1(x,y,z) = x^2 + y^2 + z^2$,
- $f_2(x,y,z) = x^3 + y^3 + z^3$,
- $f_3(x,y,z) = 3 \cdot x^4 \cdot y^2 \cdot e^z + 8 \cdot y + 5 \cdot x \cdot e^y + 16 \cdot z^5$.

These functions will be referenced as f_1, f_2 and f_3.

Error Measurement

For the testing purposes, the boundary vertices of the mesh were filtered out from the statistics. The main measure of accuracy was the average error angle computed the following way. For each vertex, the angle in degrees between the exact gradient and the estimated one was found. Their arithmetic average then determined the average error:

$$E_a = (\sum_{i=0}^{N-1} a_i) / N \tag{73}$$

The secondary measure was the error of the vector length. In this case, the error computation consisted of the following steps. First, the difference in the length of both the vectors was enumerated for each mesh vertex. The ratio of this distance and the length of the exact gradient vector in that vertex was then expressed. Finally, the arithmetic average of such ratios was computed:

$$E_l = \sum_{i=0}^{N-1} \frac{\left\| \vec{u}_i \right| - \left| \vec{v}_i \right\|}{\left| \vec{v}_i \right|} / N \tag{74}$$

where E is the average error, u_i and v_i are the estimated and the exact gradient vectors respectively and N is the number of evaluated vertices.

Instead of (73) and (74), it would also be possible to measure the error as the length of the error vector (75), which would be the distance between the end points of the exact and the computed vector. Symbolically written:

$$E = \frac{\sum_{i=0}^{N-1} \frac{\left| \vec{u}_i - \vec{v}_i \right|}{\left| \vec{v}_i \right|}}{N} \tag{75}$$

where E is again the average error, u_i and v_i are the estimated and the exact gradient vectors respectively and N is the number of evaluated vertices. However, some applications are only interested in the error angle and do not require the gradient length to be correct. For this reason, we have used the first evaluation procedure applying equations (73) and (74).

The following picture should make the geometrical meaning of individual error expressions clear:

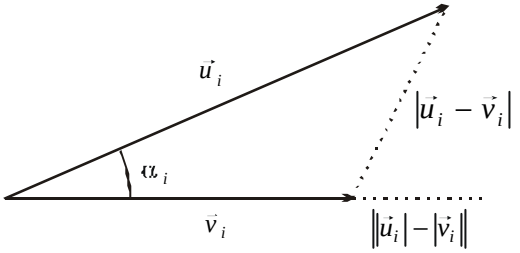


Figure 26: Error measurements illustration

6.3.3 Results

In the following paragraphs, where the implemented three approaches will be examined from several points of view and their results compared, *LR-Lin* denotes the method that uses linear regression based on linear approximation function, *LR-Nonlin* stands for linear regression based on quadratic approximation function and *FDM* represents the method based on generalization of the finite difference method described in [34].

Accuracy Statistics – Varying Sampling Functions

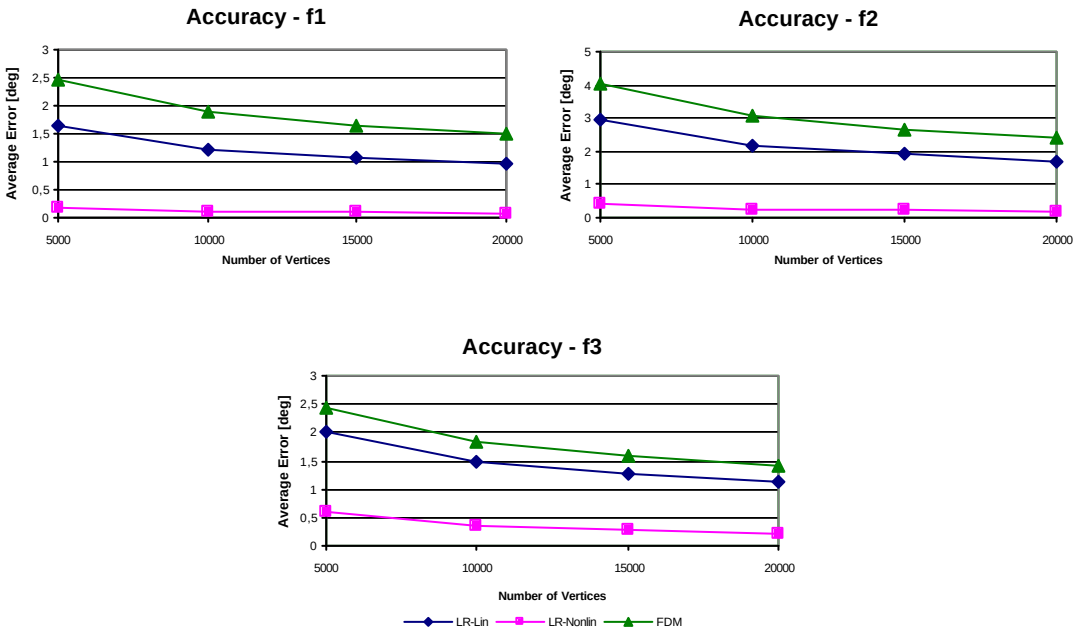
A good basic notion of how the methods perform can be acquired just by testing them on a mesh of only 1000 vertices using the Delaunay criteria. Table 4 shows that, from the accuracy point of view, LR-Nonlin performs best (marked by darker shading). The exception was using it for estimating linear function (plane) gradients, where the FDM reached the best results. The reason is that LR-Nonlin attempts to approximate sample values in the vertices by a quadratic function. Therefore, when applied on a simple plane, it performs worse than the linearly oriented methods. In all the other cases, however, LR-Nonlin reached the most accurate results while FDM the worst, LR-Lin being in the middle. The linear sample function (plane) will not be included in further testing as the results balance on the edge of computational numerical precision and are not of high importance, for in practice, linear sampling function can hardly be expected.

Error Angle in Degrees	LR-Lin	LR-Nonlin	FDM
x (plane)	1.31E-15	1.29E-12	9.30E-16
$x^2 + y^2 + z^2$ (sphere)	2.55	0.45	3.89
$x^3 + y^3 + z^3$	4.66	0.92	6.32
$3\,x^4\,y^2e^z + 8\,y + 5\,x\,e^y + 16\,z^5$	3.36	1.54	3.86

Table 4: Tests on Delaunay tetrahedra mesh with 1000 vertices.

Accuracy Statistics – Varying Data Density

The following three graphs (one graph for each of the three sample functions) show how the estimation results improve when supplying more information by using a denser mesh. Although the graphs look quite similar, it is necessary to note that the scale on the *y* axis differs to keep the graphs legible.



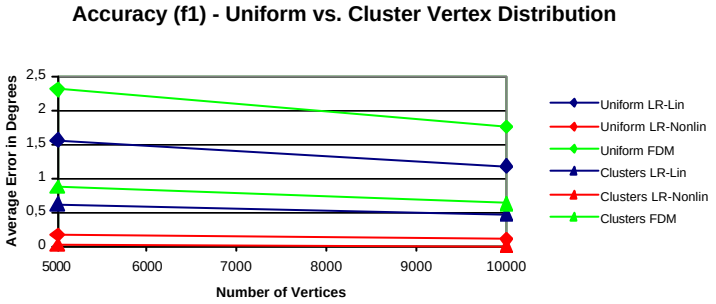
Graph 8: Estimation accuracy for f1, f2 and f3

Each value in the graphs has been obtained as an average of three measurements, each using a different tetrahedra mesh (i.e. Delaunay mesh and meshes with maximal and minimal number of tetrahedra). It is obvious from the graphs above that the denser

sampling is available the more accurate gradient estimation can be expected. As well, the graphs confirm that (except for the linear sample functions) the LR-Nonlin method returns best results. On the other hand, comparing these three graphs to each other reveals an interesting fact that more complex sampling function does not imply lower estimation accuracy. Regardless of the estimation method used, the results of gradient computation are more precise for f_3 than for f_2. In fact, in case of FDM, the results for f_3 are even slightly better than those for f_1. These, maybe a little surprising, results are promising for practice, where the samples will probably not approximate simple neat functions.

Accuracy Statistics – Varying Vertex Distribution

In this section the influence of the structure of the input mesh on the accuracy of the estimation will be demonstrated. For this purpose, a pair of Delaunay tetrahedra meshes was generated upon 5000 and 10000 vertices distributed in clusters Graph 9 shows the average estimation error for all three methods on both the uniform as well as the clustered vertices, meshed by the Delaunay method. Although the graph was meant primarily to illustrate the influence of the mesh structure on the results, we can also notice that the LR-Nonlin method performed best again with significant advance to LR-Lin, let alone FDM. Since the graphs for different sample functions f1, f2 and f3 resembled each other, only one of them will be presented here. For easier orientation, the marks at the ends of the lines are rectangular for the estimation from the mesh with uniformly distributed vertices and triangular for the mesh on clusters.

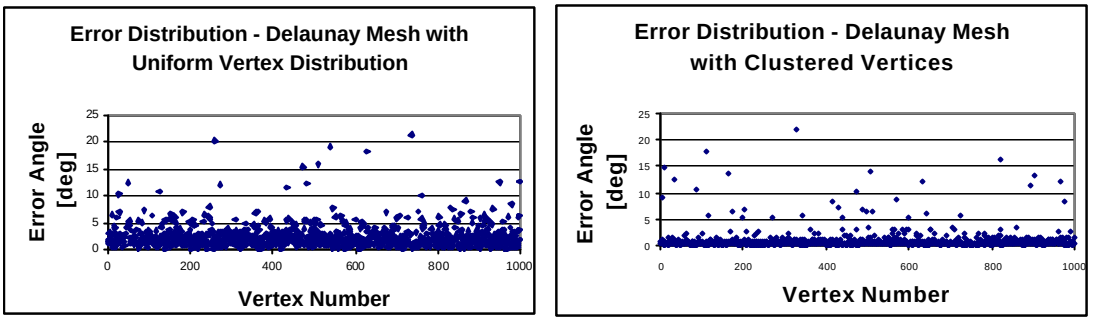


Graph 9: The average error of the estimation on meshes with clustered and uniform vertex distribution.

At the first glance it might seem rather surprising, but the graph shows that the gradient estimation applied on the tetrahedra mesh generated upon the clusters of vertices gives better results than the mesh with the same number of vertices distributed uniformly. Closer analysis shows that such results in fact correspond to what was described in the previous section. When the vertices are grouped in clusters, some of them are positioned at locations, where the clusters are connected to each other. In these locations, big errors can be expected as the surrounding of these vertices consists of small tetrahedra in the direction of the cluster, on the boundaries of which the vertex resides, and large tetrahedra in the other direction, where the cluster is connected to the other clusters. This unbalanced distribution of information around these vertices causes the failure of all the gradient estimation methods. The estimated gradient vectors are strongly inaccurate in such locations. Yet, this situation applies to only a small percent of vertices. The majority is located inside the clusters, where their density is higher than in case of uniform distribution, which leads to better estimations. The bigger errors are

compensated and the overall average error is lower for the clustered data than for the uniformly distributed vertices, where some error peaks appear as well, especially in the locations close to the surface.

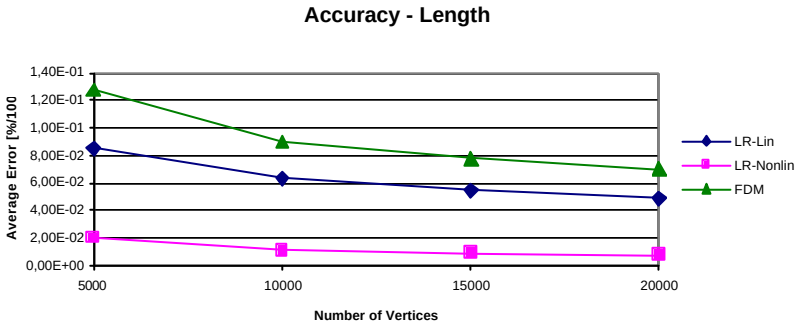
The distribution of the error within the data is illustrated by Graph 10. To keep the graphs understandable, files with only 1000 vertices have been used. The vertices, where the error has its peaks, are easily recognizable and the situation in some of these "problematic" vertices has been analyzed visually. This analysis was the ground to the explanations described above.



Graph 10: The distribution of the error for uniformly distributed and clustered vertices.

Accuracy Statistics – Vector Length

So far, we have only been concerned in measuring the error angle between the exact and the estimated gradient vectors. It is however necessary to realize that, unlike for example surface normal vector, gradient is determined by its length as well. Therefore the methods were also tested from this point of view and the results have been summarized in Graph 11.

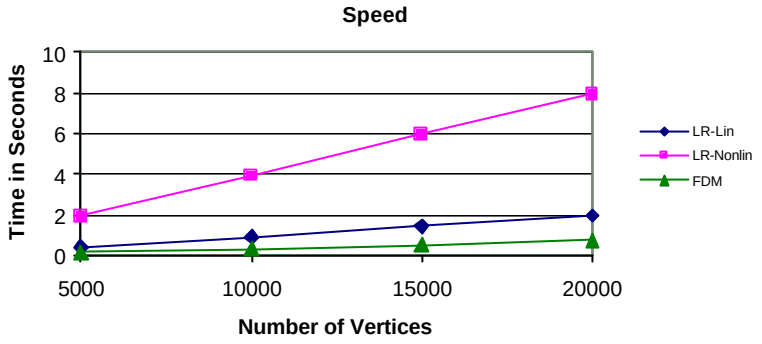


Graph 11: The average error of the estimated gradients' length in percents of the exact vector length.

Each value in this graph was obtained as the arithmetic average of nine measurements combining the usage of three sampling functions on the three types of meshes described above. We can see that the LR-nonlin method gives the most precise results being far ahead of the other two. The LR-lin estimations were approximately four times less accurate and those of FDM more than six times. Using different mesh types did not lead to significant differences here. Concerning the sampling function, results for f_1 were a little better than results for f_2, f_3.

Speed Statistics

Although we have adopted the accuracy of individual methods as the main criterion according to which the methods should be judged, the time requirements of the computations are usually much too important to bee ignored. To be consistent, the temporal needs of the three approaches were measured on the same data files as in the section Accuracy Statistics – Varying Data Density and the results are summarized in Graph 12. Each value in the graph has thus been obtained as an average of three measurements on different tetrahedra meshes. When compared to each other, the results of these three measurements differed just slightly i.e. at most around 10 percent in one direction or the other.



Graph 12: Comparison of the methods according to their time requirements

Apparently, the more accurate the method is, the more time it needs to do the computation. While producing the best results, the linear regression method utilizing quadratic regression function required about four times more time than the method using linear approximation function and up to ten times more than the finite difference method. Yet, all three methods have linear time complexity *O(N)*, where N is the number of vertices.

Conclusion

In this chapter, the behavior of the gradient estimation methods has been examined in terms of accuracy while varying several aspects as the mesh internal structure, density and the sample function defining the scalar field's values. From this point of view, the linear regression method with quadratic approximation function turned out to be the best, providing significantly better results than the other two. On the other hand, it was the most time demanding one. Although the time complexity is linear for all these methods, the growth of the time requirements is steeper for the method using quadratic function. Thus, an imaginary ratio accuracy/speed is roughly the same for all the methods.

7 CONCLUSION

In this work a study of the multidimensional data visualization approaches has been presented. Although the broadness of the topic and the variety of existing techniques did not allow to make this survey exhaustive, the main attitudes have been described and the most important aspects have been discussed in detail. After overviewing the kinds of multidimensional data in chapter 2, focus has been put especially on the approaches and techniques for the analysis and depiction of vector fields, tensor fields and also fields of multi-scalar values, as these seem to be the most important types for representing scientific as well as industrial data. In addition, our work related to this topic was outlined in the last chapter.

Concerning our current work and future goals, at present we are about to finish the accuracy and time requirements tests of estimating isosurface vertex normals using the standard techniques known for triangle meshes as compared to estimating these normals by interpolating scalar field's gradients, computed prior to the isosurface extraction step. We have also been preparing a publication on this subject, which is our primary short term aim.

Although our work has not been focused primarily on multidimensional data visualization so far, it has been related and we would like to shift the scope of our future efforts to this area, with special concern on vector, and possibly tensor, fields. We see our long term goal in searching for improvements of the visualization methods in respect to the practical needs of the researches. Rather than trying to speed up existing approaches, we will focus on the qualitative aspects. That is, how to visualize more information yet avoiding visual clutter. A concrete example of the problems, we would like to deal with, is visualization of the transition areas, where a flow of steam transforms to a stream of moving liquid droplets. Another problem we find interesting is comparative visualization.

REFERENCES

[1] Battke, H., Stalling, D., Hege, H. C.: Fast Line Integral Convolution for Arbitrary Surfaces in 3D, Visualization and Mathematics, pages 181--195. Springer-Verlag, Heidelberg, 1997.

[2] Boring, E., Pang, A.: Directional flow visualization of vector fields, Proc. Visualization'96, pp. 389-392, 1996.

[3] Boring, E., Pang, A.: Interactive deformations from tensor fields, Proceedings IEEE Visualization 98, pages 297-304, 1998.

[4] Brill, M., Hagen, H., Rodrian, H. C., Djatschin, W ., Klimenko, S. V.: Streamball techniques for flow visualization, IEEE Visualization 94, pp. 70-76, 1994.

[5] Brodlie K.W., Carpenter L.A., Earnshaw R.A. ,Gallop J.R.,Hubbold R.J., Mumford A.M.,Osland C.D., Quarendon P.: Scientific Visualization - Techniques and Applications, Springer-Verlag, 1992.

[6] Buning, P.: Numerical Algorithms in CFD Post-processing, Computer Graphics and Flow Visualization in Computational Fluid Dynamics, von Karman Institute for Fluid Dynamics Lecture Series 1989-07, 1989.

[7] Cabral, B., Leedom, C.: An Introduction to Line Integral Convolution, Siggraph 97, 1997.

[8] Cabral, B., Leedom, C.: Imaging vector fields using line integral convolution, Siggraph'93, 1993.

[9] Cignoni, P., Marino, P., Montani, C., Puppo, E, Scorpio, R.: Speeding Up Isosurface Extraction using Interval Tree, IEEE Transactions on Visualization and Computer Graphics, 1997.

[10] Delmarcelle, T., Hesselink, L.: Visualizing Second-Order Tensor Fields with Hyperstreamlines, IEEE Computer Graphics and Applications, 13(4): 25-33, 1993.

[11] Dovey, D.: Vector Plots for Irregular Grids, Proc. Visualization'95, pp.248-253, IEEE Computer Society, 1995.

[12] Engel, K., Ertl, T.: Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware, EUROGRAPHICS 2002, Saarbrucken, 2002.

[13] Forssell, L., Cohen, S.: Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows, IEEE Transactions on Visualization and Computer Graphics Volume 1, 1995.

[14] Globus, A., Levit, C., Lasinski, T.: A Tool for Visualizing the Topology of Three-Dimensional Vector Fields, Proc. Visualization'91, pp.33-39, 1991.

[15] Gouraud, H.: Continuous Shading of Curved Surfaces, IEEE Transactions on Computers, Vol. 20, No. 6, pp. 623--629, June 1971.

[16] Hesselink, L., Levy, Y., Lavin, Y.: The Topology of Symmetric, Second-Order 3D Tensor Fields, IBEE. Trans. Vis&CG, pp. 1-11, Vol. 3, No. 1, 1997.

[17] Hultquist, J. P. M.: Constructing Stream Surfaces in Steady 3D Vector Fields, Proc. Visualization'92, pp. 171-178, 1992.

[18] Inselberg, A.: The plane with parallel coordinates, The Visual Computer 1, New York: Springer, pp. 69-91, 1985.

[19] Jobard, B., Lefer, W.: Creating evenly-spaced streamlines of arbitrary density, Proc. of 8th Eurographics Workshop on Visualization In Scientific Computing, pp. 45-55,1997.

[20] Kenwright, D. N., Haimes, R.: Automatic Vortex Core Detection, IEEE CGA, 18(4): 70-74, 1998.

[21] Kenwright, D. N., Lane, D.: Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition, IEEE TVCG, 2(2): 120-129, 1996.

[22] Kindlmann, G., Weinstein, D.: Hue-Balls and Lit-Tensors for Direct Volume Rendering of Diffusion Tensor Fields, 10th IEEE Visualization Conference, 1999.

[23] Kriz, R. D., Glassgen, E. H., MacRae, J. D.: Eigenvalue-Eigenvector Glyphs: Visualizing Zeroth, Second, Forth and Higher Order Tensors in a Continuum, Workshop on Modelling the Development of Residual Stresses During Thermoset Composite Curingm, 1995.

[24] Laidlaw, D. H., Ahrens, E. T., Kremers, D., Avalos, M. J., Jacobs, R. E., Readhead, C.: Visualizing Diffusion Tensor Images of the Mouse Spinal Cord, IEEE Visualization 98, pp. 127-134, 1998.

[25] Lavin, Y., Levy, Y., Hesselink, L.: Singularities in Nonuniform Tensor Fields, Proceedings of Visualization '97, pp. 59-66, 1997.

[26] Lane, D. A.: Scientific Visualization of Large-scale Unsteady Fluid Flows, Scientific Visualization: Overviews, Methodologies, and Techniques, chapter 5, pp. 125-145, 1997.

[27] Lane, D. A.: Visualization of Numerical Unsteady Fluid Flows, Sixth International Symposium on Computational Fluid Dynamics, 1995.

[28] Lane, D. A.: Visualizing Time-Varying Phenomena in Numerical Simulations of Unsteady Flows, 34th AIAA Aerospace Sciences Meeting, AIAA 96-0048, 1996.

[29] Livnat, Y., Shen, H. W., Johnson, C. R.: A Near Optimal Isosurface Extraction Algorithm Using the Span Space, IEEE Transactions on Visualization and Computer Graphics, 1996.

[30] Loffelmann, H.: Visualizing Local Properties and Characteristic Structures of Dynamical Systems, PhD Thesis, TU Wien, 1998.

[31] Loffelmann, H., Groller, E.: Enhancing the Visualization of Characteristic Structures in Dynamical Systems, Proceedings of 9th EUROGRAPHICS Wokshop on Visualization in Scientific Computing, pp. 35-46, 1998.

[32] Loffelmann, H., Szalavari, S., Groller, E: Local Analysis of Dynamical Systems - Concepts and Interpretation, Proceedings of the Fourth International Conference in Central Europe on Computer Graphics and Visualization, pp.170-180,1996.

[33] Max, N., Crawfis, R., Grant, C.: Visualizing 3D velocity fields near contour surfaces, Proceedings of Visualization '94, IEEE Press, pp. 248-255, 1994.

[34] Meyer T.H., Eriksson M., Maggio R.C.: Gradient estimation from irregularly spaced data sets, Mathematical Geology, 33: (6) 693-717, August 2001.

[35] Míka, S.: Matematická analýza III – Tenzorová analýza, edicní stredisko ZCU Plzen, 1993.

[36] Moran, P., Henze, C., Ellsworth, D., Bryson, S., Kenwright, D.: The Field Encapsulation Library (FEL), http://www.nas.nasa.gov/Groups/VisTech/projects/fel.

[37] Neumann L., Csbfalvi B., Knig A., Groeller E.: Gradient Estimation in Volume Data using 4D Linear Regression. EUROGRAPHICS 2000, 2000.

[38] Post, F. H., Vrolijk, B., Hauser, H., Laramee, R. S., Doleisch, H.: Feature Extraction and Visualisation of Flow Fields, EUROGRAPHICS 2002, Saarbrucken, 2002.

[39] Post, F. H., van Walsum, T.: Fluid Flow Visualization, Focus on Scientific Visualization, pp 1-40, 1993.

[40] Post, F. J., van Walsum, T., Post, F. H.: Iconic Techniques for Feature Visualization, Proceedings Visualization '95, IEEE Computer Society Press, pp. 288-295, 1995.

[41] Reinders, F.: Feature-Based Visualization of Time-Dependent Data, PhD thesis - ASCI, dissertation series number 61, 2001.

[42] Rektorys, K.: Prehled užité matematiky. SNTL – Nakladatelství technické literatury, Praha, 1981.

[43] Sanna, A., Montrucchio, B., Arinaz, R.: On Time-Varying Flow Fields: a streakline-based visualization method, Eurographics'99 Short Papers and Demos Proceedings, pp. 30-33, 1999.

[44] Sanna, A., Montrucchio, B., Arinaz, R.: Visualizing unsteady flows by adaptive streaklines, WSCG 2000 Conference Proceedings, 2000.

[45] Sanna, A., Montrucchio, B., Montuschi, P.: A survey on visualization of vector fields by texture-based methods, Research Developments in Pattern Recognition, 1(1), 2000.

[46] Scheuermann, G., Frey, J., Hagen, H., Hamann, B., Jeremic, B., Kenneth, I. J.: Visualization of Seismic Soils Structure Interaction Simulations, Proceedings of IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001), ACTA Press, pp. 778-83, 2001.

[47] Schroeder, V., Volpe, C. R., Lorensen, W. E.: The Stream Polygon: A Technique for 3D Vector Field Visualization, Visualization '91, pp. 126-132, 1991.

[48] Schulz, M., Reck, F., Bartelheimer, W., Ertl., T.: Interactive visualization of fluid dynamics simulations in locally refined cartesian grids, Proc. Visualization'99, pp. 413-416, 1999.

[49] Shen, H., Johnson, C. R.: Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids, 6th IEEE Visualization Conference, 1995.

[50] Shen, H., Kao, D.L.: A new line integral convolution algorithm for visualizing time-varying flow fields, IEEE Transactions on Visualization and Computer Graphics 4:2, pp. 98-108, 1998.

[51] Shen, H., Kao, D.L., Chiang, L., Kuswik, A.: GLIC: An Interactive Software Tool for Visualizing Surface Flows, 37th AIAA Aerospace Sciences Meeting and Exhibit, 1999.

[52] Spears, W.M.: An Overview of Multidimensional Visualization Techniques, GECCO, 1999.

[53] Stalling, D., Hege, H. C.: Fast and Resolution Independent Line Integral Convolution, SIGGRAPH 95 Conference Proceedings, pp. 249-256, 1995.

[54] Stalling, D., Zockler, M., Hege, H. C.: Fast Display of Illuminated Field Lines, IEEE Transactions on Visualization and Computer Graphics 3:2, pp. 118-128, 1997.

[55] Thurmer G., Wuthrich A.: Computing vertex normals from polygonal facets. Journal of Graphics Tools, 3(1):43-46 1998.

[56] Treinish, L. A.: Multi-resolution visualization techniques for nested weather models, Proc. Visualization'00, pp. 513-516, 2000.

[57] Turk, G., Banks, D.: Image-Guided Streamline Placement, SIGGRAPH 96 Conference Proceedings, Anual Conference Series, pp. 453-460, 1996.

[58] Ueng, S. K., Sikorski, C., Ma, K. L.: Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids, IEEE TVCG, 2(2), pp. 100-110, 1996.

[59] Verma, V., Kao, D., Pang, A.: A flow-guided streamline seeding strategy, Proc. Visualization'00, pp. 163-170, 2000.

[60] Verma, V., Kao, D., Pang, A.: PLIC: Bridging the Gap Between Streamlines and LIC, Proceedings of IEEE Visualization '99, San Francisco, October 1999.

[61] van Wijk, J. J.: Spot noise - texture synthesis for data visualization, Proceedings of Siggraph'91, pp. 309-318, 1991.

[62] Wunsche, B.: Visualization of Tensor Fields in Bioengineering, http://www.cs.auckland.ac.nz/~burkhard/PhD/introduction.html.

[63] Zockler, M., Stalling, D., Hege, H. C.: Interactive Visualization of 3D-Vector Fields Using Illuminated Stream Lines, Proc. Visualization'96, pp. 107-113, 1996.

[64] Žára, J., Beneš, B., Felkel, P.: Moderní pocítacová grafika, Computer Press, Praha, pp. 353-355, 1998.

PUBLICATIONS

[i] Jirka, T., Skala, V.: Isosurface Vertex Normal Computation, submitted to Szczyrk 2003, 2003.

[ii] Jirka, T.: Metody extrakce isoploch pro tetrahedronové síte a zobrazování neskalárních velicin, diplomová práce + dodatek k DP, Fakulta aplikovaných ved – Západoceská univerzita v Plzni, 2001.

[iii] Jirka, T., Skala, V.: Gradient Estimation and Vertex Normal Computation, Technical Report, University of West Bohemia in Pilsen, 2002.

[iv] Jirka, T., Skala, V.: Gradient Vector Estimation, ICCVG 2002, 2002.

STAYS AND CONFERENCES

Stays

14.2.2001-14.6.2001 University of Girona, Spain

Conferences

25.9.2002-29.9.2002 ICCVG 2002, Zakopane, Poland