MASTER'S THESIS
Feature-based Visualization of Time-Dependent Vortices
by
A.W. van den Heuvel

Supervisors:  prof. dr. ir. J. J. van Wijk
dr. ir. I. Barosan

Eindhoven, March 2006
Abstract

Vortex feature detection is a recent method to effectively work with the large datasets resulting from time-dependent CFD simulations. By employing feature detection on these datasets reductions in the order of 1:10000 are possible. Our vortex feature detection method detects the coreline of the vortex. Since the core of a vortex has specific features that clearly distinguish it from the rest of the vortex volume, it is a logical choice. Other methods focus on the entire vortex volume.

The current implementation for detecting vortex cores uses the parallel vector operator, as proposed by Roth & Peikert. The operator is defined as $v \parallel w$, which returns for two given $n$-dimensional vector fields $v$ and $w$ all points $x$ for which both $v(x)$ and $w(x)$ are parallel, restricted to non-degenerate solutions. As it turns out, this operator can be used to effectively express a wide range of existing algorithms for detecting vortex cores. Part of our research focused on improving the results of the coreline detection algorithm.

Vortex cores detected for several timesteps can be tracked using an algorithm proposed by Roth & Peikert, again based on the use of the parallel vector operator. We present the rationale and design of an improved stream surface generation algorithm based on their original tracking algorithm.

Finally, our research includes the design and implementation of a general flow visualization primitive based on streamlines, that can be used to effectively visualize both global flow properties and vortex flow properties, based on the detected vortex coreline features.
Acknowledgements

I would like to thank my supervisor prof. dr. ir. Jack van Wijk for his helpful comments, valuable insights, and continuing support.

Also, I would like to thank my other supervisor, dr. ir. Ion Barosan, for his suggestions and never ending enthusiasm. He taught me some valuable lessons, both to the subject matter at hand and to matters regarding life in general.

Finally, I thank my parents for both their support and patience throughout the course of this project.
## Contents

Abstract .................................................................. 3

Acknowledgements .................................................. 5

1 Introduction .......................................................... 9
   1.1 Research question .............................................. 9
   1.2 Overview ...................................................... 9

2 Feature-based Flow Visualization ........................ 11
   2.1 Introduction ................................................... 11
   2.1.1 Feature-based visualization ............................. 11
   2.2 Feature dimensionality ...................................... 12
       2.2.1 Point features .......................................... 12
       2.2.2 Visualization of point features ...................... 13
       2.2.3 Line features ........................................... 13
       2.2.4 Visualization of line features ...................... 13
       2.2.5 Surface features ....................................... 14
       2.2.6 Visualization of surface features .................. 14
       2.2.7 Volume features ........................................ 14
       2.2.8 Visualization of volume features .................. 15
   2.3 Feature tracking .............................................. 15

3 Vortex Detection .................................................. 17
   3.1 Vortex definitions ............................................ 17
   3.2 Vortex models ............................................... 18
       3.2.1 Solid body rotation .................................... 18
       3.2.2 Potential vortex ........................................ 19
       3.2.3 Rankine vortex ........................................ 19
   3.3 Vortex properties ............................................ 20
   3.4 Vortex detection .............................................. 21
       3.4.1 State of the art .......................................... 21
   3.5 Parallel vector operator ..................................... 23
3.5.1 Definition .......................................................... 23
3.5.2 Banks & Singer’s method using the parallel vector operator ... 24
3.6 Implementation ........................................................ 26
3.7 Improved corepoint detection ......................................... 28

4 Vortex Tracking ......................................................... 31
4.1 Tracking theory ....................................................... 31
4.2 State of the art ......................................................... 32
4.3 Vortex tracking using the parallel vector operator ............... 33
   4.3.1 Bauer & Peikert’s tracking algorithm ......................... 33
4.4 Optimized tracking algorithm .................................... 35
   4.4.1 Streamsurface boundary propagation ......................... 38

5 Vortex Visualization .................................................. 47
5.1 Streamlines and pathlines ........................................ 47
5.2 Stream arrow rationale .......................................... 48
5.3 Stream arrow parametrization .................................... 49
5.4 Stream arrow construction ....................................... 50
5.5 Slice parametrization .............................................. 55
   5.5.1 Lobed circular slice ........................................... 55
   5.5.2 Rectangular slice ............................................. 57
5.6 Vortex visualization using stream arrows ....................... 58

6 VortexView .............................................................. 65
6.1 VortexView use case scenarios .................................. 65
   6.1.1 A single timestep use-case scenario ......................... 65
   6.1.2 A multiple timestep use-case scenario ..................... 71
6.2 Implementation of VortexView .................................. 73

7 Conclusions ............................................................. 75
7.1 Future work ......................................................... 75

References ................................................................. 81

A Vector Calculus ........................................................ 83
A.1 Notation ............................................................. 83
A.2 Definitions .......................................................... 84
A.3 Scalar and Vector Field Operations ............................... 84
   A.3.1 Gradient ....................................................... 84
   A.3.2 Vorticity ...................................................... 87
   A.3.3 Helicity ....................................................... 89
Chapter 1

Introduction

Fluid dynamics is an important research direction in physics. It plays a major role in areas like aircraft and ship design. Of particular interest are the vortical structures present in most fluid flows. For example, engineers working on the design of combustion engines are interested in the presence of vortices within the flow of fuel through the engine, since they may seriously impair the engine.

Similarly, engineers designing the rotors for large sea vessels aim to minimize vortex creation at the rotor’s tips, to prevent damage to the rotors.

As a final example, the SMARTER research project at the Eindhoven University of Technology (TU/e) conducts research on the simulation of anisotropic flow for transport processes. One typical application is to study how vortices influence the mixing of material particles in chemical reactors. The visualization method and accompanying application was specifically designed and developed to aid their research.

The SMARTER project is funded by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek institute (NWO) in the Computational Science programme with research grant number 635.000.002.

1.1 Research question

The members of the SMARTER project expressed the need for a software tool enabling the automatic detection and subsequent visualization of vortices within time-dependent Computational Fluid Dynamics (CFD) data resulting from flow simulations. Our research is directed at detecting the vortices in the flow, and developing a time-efficient method to track the vortices over time. Furthermore, a novel method for visualizing these vortical structures is requested. Ideally, using the novel visualization method, features of the global flow data can be visualized as well.

1.2 Overview

In Chapter 2 we give a brief introduction on the field of feature-based visualization and feature-based flow visualization in particular. We discuss the current state of the art of the field, subdividing each method based on the feature(s) they visualize. Chapter 3 discusses the visualization of vortices. Several vortex models are discussed, as well as different methods to detect vortices within a flow field. The current state of the art of
vortex visualization is presented, with a discussion of the advantages and disadvantages of each method. After that, Chapter 4 handles the tracking of vortices over time. An existing tracking algorithm is described, as well as an optimized and adapted version of the algorithm. In Chapter 5, we present a novel graphics primitive called a stream arrow for the feature-based visualization of vortices within flow fields. Next, in Chapter 6, several use cases in relation with our vortex visualization tool, VortexView, are discussed, as well as a short overview of the implementation. Finally, in Chapter 7, our conclusions are presented, and some suggestions for future research directions are given.

Appendix A contains an introduction to the mathematics of vector fields for readers not familiar with vector fields and operators defined on vector fields like the gradient and Jacobian operator. Basic flow quantities like helicity and vorticity are discussed as well.
Chapter 2

Feature-based Flow Visualization

2.1 Introduction

Flow visualization is a sub-field of scientific visualization, which in turn is a sub-field of visualization in computer graphics. Scientific visualization deals with continuous data, and is generally referred to as the process of mapping continuous data on graphics primitives. For practical purposes, the continuous data is generally sampled on a discrete set of locations, called nodes, which together usually form a grid. Flow visualization is concerned with visualizing the vector fields resulting from, for example, flow experiments or Computational Fluid Dynamics (CFD) data obtained from flow simulations.

Flow field visualization techniques can be subdivided into three categories [15]: global, geometric, and feature-based visualization. Global visualization is a qualitative visualization of a complete dataset at a low level of abstraction. There is a direct mapping from the data to the visual representation without any complex extraction or conversion step. Examples include the direct visualization of scalar fields using color mapping or volume rendering, or for vector fields, global arrow plots and texture-based visualizations. Geometric visualization extracts and visualizes geometric objects from the data of which the shape is directly related to the data. Examples of geometric visualizations include iso-curves and iso-surfaces for scalar fields, and local arrow plots, stream lines, and stream surfaces for vector fields. Feature-based visualization assumes that a subset of the original dataset is of special interest. It focuses on visualizing these interesting aspects, while the rest of the dataset is not visualized, or not visualized in much detail to avoid visual clutter. Laramee [32] distinguishes yet another category, texture-based visualization, which encompasses all texture-based flow visualization techniques.

2.1.1 Feature-based visualization

Feature-based visualization is mainly used for visualizing large datasets resulting from, e.g., CFD simulations. The resulting datasets contain only a limited amount of relevant information. Feature-based visualization aims to extract the relevant information, and subsequently visualize it. The main advantage of feature-based visualization is that its data representation is much more compact than the original data set. Data reductions in the order of 1:10000 are possible [20].

For example, assume we have a $100 \times 100 \times 100$ dimensional grid, with for every grid point a scalar and vector attribute value representing pressure and velocity respectively.
Furthermore, we make the conservative assumption that 20 vortices are present in the flow, extending over 20 grid cells. Then, after vortex detection, we only need to consider 400 points instead of 1000000, a data reduction of 1:2500.

Taking into account the large data sets resulting from CFD research (this is especially true for time-dependent simulations), being able to size down these datasets significantly is an important advantage. Once all the features from a time-dependent data set have been extracted, an interactive visualization becomes more feasible.

By integrating the concepts of the application area for which the visualization is being developed, a clearer, more meaningful visualization is possible, since only the features the end-user is interested in are visualized.

### 2.2 Feature dimensionality

First, let us give a formal definition of a feature.

**Definition 1:**
Assume we have a function \( f : D \to S \), with \( D \) and \( S \) a certain domain, e.g., \( \mathbb{R} \) or \( \mathbb{R}^2 \times T \). Furthermore, assume we have defined a condition \( C \) on \( D \) and \( f \), where \( C(x, f(x)) \), with \( x \in D \), implies that \( C \) holds for \( x \) and \( f(x) \). Now, a feature is defined as a connected set \( F \subseteq D \), with \( F = (\cup x : x \in D : C(x, f(x)) \).

Definition 1 implies:

**Corollary** All the points \( x \) not belonging to a feature are defined by the set \( S = D \setminus F \). More formally, \( S = (\cup x : x \in D : \neg C(x, f(x)) \).

Examples of features within a flow dataset are critical points, vortex corelines, shock waves and so forth. Typically, features are distinguished from each other based on their dimensionality. Note that we distinguish here between the spatial dimensionality of the visualization domain, and the dimensionality of the features.

#### 2.2.1 Point features

Point features are described by a 0D location in the data set. An example of a point feature is a **critical point**, i.e., a point \( x \) where \( v(x) = 0 \). Translating this to the notation of definition 1, we obtain for \( f: f(x) = v(x) \), and the condition \( C: C(x, y) := y = 0 \). Critical points may be further subdivided in saddle points, center points etc. Arbitrary point features can be defined based on certain conditions that need to hold for combinations of properties of points in a flow data set. Typical point properties include:

**Vorticity:** Vorticity is used to denote the curl of the velocity in a flow field, i.e., the circulation per unit area at a point in a flow field. The vorticity is denoted by a vector quantity, pointing along the axis of rotation. A more thorough explanation is given in Appendix A.

**Helicity:** Helicity is a scalar value calculated by taking the dot product of the normalized velocity and vorticity. Helicity mainly results from shear effects in the flow. Therefore, helicity is usually highest near boundary walls of a flow.
Pressure: The pressure within a vortex is highest at the boundaries of the vortex, and decreases towards the center of the vortex.

Velocity: A vector indicating the direction and magnitude of velocity at a specific point in the flow field.

2.2.2 Visualization of point features

Several icons have been used to visualize individual point features. A short list is given below.

Sphere: a trivial example of an icon to visualize a point feature is a sphere with its center at the location of the point of interest. The color of the sphere may depend on the point feature that needs to be visualized.

Arrow: another basic example of a point feature icon is an arrow, used to depict local properties of the flow at the point of interest.

Flow probe point: first proposed by De Leeuw and Van Wijk [14], the flow probe point icon extends the arrow icon by combining the local values of velocity, torsion, curvature, acceleration, shear, and convergence in one visualization.

2.2.3 Line features

Line features are features located on a 1D connected set of points in the domain. Some examples of line features are:

Vortex core: there is no general formal definition of what actually constitutes the vortex core. Roth [34] describes the vortex core line as 'the center line of the swirling flow structure, around which the flow spirals'.

Stream line: a stream line is a line that is tangential to the instantaneous velocity direction in a flow. Determining either the start- or endpoint of the streamline such that most insight into the flow features is obtained is difficult. One solution is to let the user interactively pick a starting point from which the streamline should emanate. Turk and Banks [42] proposed an automatic seed point selection method that results in a picture with a uniform streamline distribution.

Path line: a path line is the line that an imaginary particle would make if it followed the flow of the fluid it was released in.

Streak line: a streak line is the line that is formed by constantly releasing particles at a certain point in the flow.

2.2.4 Visualization of line features

Several icons can be used to visualize certain line features:

Twisted stream tube: the formal definition of a stream tube is: the surface formed by all streamlines passing through a closed contour [7]. Darmofal [13] describes a method to extend stream tubes to let the visualization convey rotation and divergence effects as well.
Stream ribbon: a stream ribbon is a visualization of a stream line by giving it a certain width and rotation based on the local velocity and/or vorticity [13]. Darmofal considers stream ribbons to be less effective at conveying the twist information than stream tubes.

Animated particles: Ragas [28] describes a time-dependent visualization method for stream lines that is also able to convey velocity information. A glyph-like object is animated along the stream line with a velocity equal to the flow velocity at the stream line. A static version of the method renders glyphs along the stream line using an offset between two glyphs based on the velocity at the stream line.

All the surfaces of the objects described above may be coloured by a scalar to denote any one of the vortex properties as described in Section 2.2.1.

2.2.5 Surface features

The location of surface features is described by a 2D set of points. Examples of surface features are:

Vortex volume surface: the set of points defining the boundary of vortex volume. The vortex volume may be defined using several different methods. Roth [34] discusses several methods, e.g., based on a threshold on pressure, helicity and vorticity magnitude. His conclusion is that none of the discussed methods was able to reliably detect all vortices in typical turbomachinery flow data. Banks & Singer [5] suggest to use the pressure to obtain a good visualization of the vortex volume surface. They make use of a previously detected vortex core to construct the surface.

2.2.6 Visualization of surface features

Stream surface: a stream surface is defined by a set of stream lines spawned from a line [17, 18]. Extra stream lines need to be inserted at points where the flow locally diverges, e.g., when encountering an obstacle like a turbine blade.

Surface particles: surface particles are particles with a given normal that are carried along with the flow. They can be used to approximate stream surfaces [43].

As with the line feature objects, surface objects may be coloured by a scalar to denote any of the vortex properties as described in Section 2.2.1. Furthermore, texture methods may be used to further enhance the visualization of the surface object, e.g., using Line Integral Convolution (LIC) [11] or IBFV [44] a noise image smoothed along streamlines is generated that can be used either as a static or animated texture.

2.2.7 Volume features

Volume or region features are described by a 3D set of points. Examples of volume features are:

Vortex volume: The set of points defining the vortex volume. Exactly what definition is used to define whether a point actually belongs to a vortex or not should is open for discussion (see 2.2.5).
2.2.8 Visualization of volume features

**Volume rendering:** Crawfis [26] describes a volume rendering method based on tetrahedra to efficiently render flow volumes.

**Texture methods:** LIC methods have been applied to render 3D flow fields [19]. The major drawback is that the resulting image is difficult to interpret, due to the denseness of final visualization.

2.3 Feature tracking

As previously discussed in Section 2.1, feature tracking is used to determine matching sets of features across several different datasets. Feature tracking is particularly useful when visualizing time-dependent vortices, since we can use it to detect certain events based on the transition of features. For example, one feature may break up in several parts, like one vortex splitting in two. Another example is two features merging, the opposite process. Furthermore, features may appear or disappear. A feature tracking graph can be used to give a schematic overview of these events over time, with symbols indicating the type of event [31].

Extensive research on feature tracking was done by Reinders and Silver et al. [31, 29, 37, 38].
Chapter 3

Vortex Detection

In the previous chapter we discussed the relevance of visualizing features within a flow field. As mentioned in Chapter 1, vortices are considered to be the most important structures within flow fields. Therefore, it is a logical choice to concentrate on locating and visualizing vortex features when visualizing a flow field.

3.1 Vortex definitions

Unfortunately, there is no unique definition for a vortex. Intuitively, when we talk about a vortex in a flow field, we refer to a certain region within the flow field that exhibits a swirling motion around some central region. Several authors provide definitions for a vortex. For example, a classic definition of a vortex is the one given by Lugt in his book ‘Vortex Flow in Nature and Technology’:

“A vortex is the rotating motion of a multitude of material particles around a common center.” [23]

Another definition of a vortex by Robinson frequently referred to is:

“A vortex exists when instantaneous streamlines mapped onto a plane normal to the vortex core exhibit a roughly circular or spiral pattern, when viewed from a reference frame moving with the center of the vortex core.” [33]

Finally, Roth in his work uses the following definition for a vortex:

“A vortex constitutes a rotating flow around a common center having some minimal size.” [34]

The definitions by Lugt and Roth are too general and vague, whereas the definition by Robinson is not a proper definition in the strict sense, since it is self-referential. Furthermore, no insight is provided in what ‘the center of the vortex core’ constitutes. We abstain from giving an informal definition for a vortex here. Instead, we base our work on the formal conditions general vortices adhere to, as discussed later on in this chapter.
3.2 Vortex models

Several mathematical models exist to describe the velocity distribution in the simplest plane vortex type, the circular vortex. The basic vortex models that we describe here can all be expressed in terms of a linear flow model. A linear flow model is described by a linear vector field, which is defined in Appendix A. Here, we define a linear vector field in 3D space.

**Definition 2 (Linear Vector Field):**
A linear vector field \( \mathbf{v} \) on the 3-dimensional Euclidean space is a real-valued 3-component vector function of 3 spatial variables \( \mathbf{v} : \mathbb{R}^3 \to \mathbb{R}^3 \) with \( \mathbf{v}(x) = \mathbf{v}_0 + Mx \), for a constant vector \( \mathbf{v}_0 \), and a constant matrix \( M \).

Thus, for a given position \( x \) in 3-dimensional space, \( \mathbf{v} \) defines a vector \( \mathbf{v}(x) \). The matrix \( M \) is also the Jacobian of the linear vector field defined by \( \mathbf{v} \). Please refer to Appendix A for an explanation of the Jacobian.

### 3.2.1 Solid body rotation

Solid body rotation is the simplest example of a vortex flow model in a plane. As an example of a solid body rotational flow consider a cylindrical tank filled with a viscous fluid that is steadily rotated. After a certain period of time, the fluid will rotate like a solid body because of its adherence to the walls of the tank.

For a more formal discussion, we assume that the rotation is in the \( x-y \) plane. Then, the Jacobian \( M \) has the form

\[
M = \begin{pmatrix}
0 & -\omega & 0 \\
\omega & 0 & 0 \\
0 & 0 & \gamma
\end{pmatrix},
\]

and describes a rotation about the origin (assuming \( \mathbf{v}_0 = 0 \)) with rotational speed \( \omega \). In this case, \( \gamma \) denotes the constant velocity in the \( z \)-direction. The streamlines of the corresponding flow field are spirals in case \( \gamma \neq 0 \). Such a flow field is usually referred to as a helical flow field. Figure 3.1 depicts the velocity and vorticity distribution for a solid body rotation in the \( x-y \) plane. In the figure, \( r \) is the distance from the center of rotation and is given by \( r = \sqrt{x^2 + y^2} \), whereas the arrows represent the velocity vectors at \( (r, 0, 0) \).

Note that in a flow exhibiting a solid body rotation, all fluid particles are at rest for an observer rotating with the fluid. The angular speed \( \omega \), defined as \( \omega = \frac{d\Theta}{dt} \), i.e., the angular position \( \Theta \) as a function of the time is constant for a solid body rotation as can be seen from Equation 3.1. The angular speed is a scalar measure of the rotational rate, and equals the magnitude of the angular velocity vector. For a solid body rotation, the angular velocity at a certain point \( x \) in the flow is equal to the vorticity at \( x \) [23]. Thus, vorticity is constant for a solid body rotation.
3.2.2 Potential vortex

For a potential or irrotational vortex, the speed of the flow is inversely proportional with respect to the distance $r$ from the center of the flow. The Jacobian $M$ has the form

$$M = \begin{pmatrix} 0 & -\frac{\omega}{r} & 0 \\ \frac{\omega}{r} & 0 & 0 \\ 0 & 0 & \phi \end{pmatrix}.$$  \hfill (3.2)

Again, we assume that the $z$-axis is the rotation axis. The distance $r$ for a point $x$ from the rotation axis is given by $r = \sqrt{x^2 + y^2}$.

For a potential vortex, the velocity magnitude diminishes to zero for increasing $r$. In contrast, for a solid body rotation the velocity magnitude will go to infinity when $r$ goes to infinity. On the other hand, for the potential vortex model the velocity magnitude will approach infinity when $r$ approaches zero.

Figure 3.2 depicts the velocity distribution for a potential vortex in the $x$-$y$ plane. The vorticity distribution is not shown, as it is zero for every point belonging to a potential vortex [7].

3.2.3 Rankine vortex

The solid body rotation and potential vortex models do not translate particularly well to vortices that occur in nature. For a solid body rotation, the velocity goes to infinity for points for which the distance to the axis of rotation goes to infinity. On the other hand, for a potential vortex, the velocity goes to infinity for points whose distance to
the axis of rotation approaches zero. More elaborate vortex models have been proposed to eliminate these problems. One of them is the Rankine vortex model, named after William Rankine (1820-1872), who first proposed it.

Figure 3.3 depicts the velocity distribution for a Rankine vortex in the \(x\)-\(y\) plane. The Rankine vortex model is a combination of a solid body rotation and a potential vortex. Consider a Rankine vortex for which the axis of rotation coincides with the \(z\)-axis. Then, all points \(p\) in the \(x\)-\(y\) plane for which \(\gamma(p) = \sqrt{p_x^2 + p_y^2} \leq R\) holds are considered to be part of a solid body rotation, where \(R\) is considered to be the average radius of the vortex. Thus, for all points \(p\), with \(\gamma(p) < R\), the Jacobian from Equation 3.1 is used to define the linear flow field, whereas for all points \(q\), with \(\gamma(q) > R\), the Jacobian from Equation 3.2 is employed.

The vorticity distribution of a Rankine vortex is constant and not equal to zero for all points \(p\) with \(\gamma(p) \leq R\). For points \(q\), with \(\gamma(q) > R\), the vorticity is zero.

![Velocity and vorticity distribution for a Rankine vortex.](image)

**Figure 3.3:** Velocity and vorticity distribution for a Rankine vortex.

### 3.3 Vortex properties

It is important to make the distinction between vortex properties and vortex features here. As discussed in Chapter 2, features are made up of a connected set of points that adhere to a certain condition. In contrast, vortex properties are either measurable or derivable physical quantities associated with the vortex as a whole, or with certain points or regions considered to be part of the vortex. Vortex properties can be derived from vortex features, whereas the converse is usually not possible.

Measurable properties include, amongst others, the velocity, pressure, or temperature at a certain point in the vortex region. An apparent derivable property is the position of a vortex, usually obtained by calculating the gravity point of the set of points making up the vortex region. In case of a time-dependent visualization, by calculating the position of a vortex for every time step, we can calculate the velocity of the vortex as a whole.

Another important vortex property is the vortex strength. No unique definition for the vortex strength exists, and usually the rotation strength is used as a measure of the vortex strength. For every point \(x\) in a flow, the rotation strength can be calculated and indicates how fast the flow locally rotates around \(x\). \(M = \nabla v(x)\) denotes the velocity gradient tensor at \(x\) for a certain flow. Now, the rotation strength for \(x\) is determined by the absolute value of the imaginary part of the complex eigenvalues of \(M\) [12, 36]. Roth [34] uses a similar approach to determine the vortex strength along a certain point on the coreline. He projects the local flow at a point \(x\) along the coreline on a plane \(P\)
with normal \( v(x) \) through point \( x \), as illustrated in figure 3.5. In practice, \(|v(x)| \neq 0\) holds, so that \( P \) is always properly defined. In a theoretical situation where \(|v(x)| = 0\) holds, the coreline tangent can be used as the normal for \( P \). The vortex strength at \( x \) is then determined by the absolute value of the imaginary part of the eigenvalue of the projected 2D flow in \( P \).

**Figure 3.4:** \( C \) is a coreline through \( x \), with \( P \) a plane centered at \( x \) with normal \( v(x) \). A hypothetical streamline of the local flow projected at \( P \) is illustrated by the gray spiraling arrow. Note that, in general, the plane normal \( v(x) \) and the coreline tangent at \( x \) are not parallel.

Using the projected 2D flow at a certain point \( x \) along the coreline we can determine another vortex property; whether the vortex is attracting or repelling. For an attracting vortex, the fluid particles are moving towards the coreline of the vortex, whereas for a repelling vortex they are moving away from the coreline of the vortex. We know that for the projected velocity on plane \( P \) denoted by \( v_P \), \(|v_P(x)| = 0\) holds, since we used \( v(x) \) as the normal vector for \( P \). Thus, \( x \) is a critical point for the projected flow on \( P \), allowing a critical point analysis to be performed on velocity gradient of the projected flow at \( x \). Abraham [3] made a classification of critical points based on the eigenvalues of the velocity gradient, along with the flow configuration around each kind of critical point, as depicted in figure 3.5. For example, for a point \( x \) somewhere along the coreline of an attracting vortex, the eigenvalues of the velocity gradient of the projected flow at \( x \) will have non-zero imaginary parts, and negative real parts.

### 3.4 Vortex detection

As previously discussed in Section 3.1, no unique definition for a vortex exists. This is a major problem, seriously hindering the development of a ‘perfect’ vortex detection method. An ideal vortex detection method does not result in any false positives, and guarantees that no vortices in the flow remain undetected.

#### 3.4.1 State of the art

Every detection method assumes a certain definition for a vortex, and either focuses on the line, surface, or volume features of a vortex, as previously discussed in Chapter 2.

An example of a method focusing on a line-type feature of a vortex is Banks & Singer’s predictor-corrector method [5]. Their method presented in 1994 was the first method able to construct vortex cores as line-type features. They assume that the vortex core approximates a vorticity line, i.e., a streamline of the vorticity field, and that the vortices are sustained by pressure gradients. The algorithm first determines the starting point of a coreline by looking at grid points with low pressure and high vorticity magnitude.
From each starting point, a coreline is constructed by tracing the vorticity lines, while correcting the prediction in the direction of the minimum of the local pressure field perpendicular to the core.

Detection methods looking at volume or region-type features of a vortex define a vortex as a region within the flow that adheres to a certain condition. Straightforward examples of region-type detection methods focus on basic properties of a vortex. For example, Zabusky et al. [25] focus on regions within the flow with high vorticity. They define the vortex as a region of high vorticity, using vorticity as the definition of rotation. Another straightforward approach would be to set a threshold on the pressure in the flow field, assuming that the centripetal force causing a rotating motion is caused by a pressure minimum at the center of the vortex. The drawback of these approaches is that they both require an arbitrary threshold. Furthermore, in some cases they fail to uniquely identify vortical structures, as a region of high vorticity is also encountered near the boundaries of a flow due to the shear encountered there.

Jeong and Hussain proposed a more practical region-type definition for a vortex, known as the \( \lambda_2 \)-definition, which is widely used. They decompose the velocity gradient tensor \( J \) into its symmetric part, the rate-of-deformation or strain tensor \( S \), and its antisymmetric part, the spin tensor \( \Omega \). Then, the three eigenvalues \( \lambda_1, \lambda_2, \) and \( \lambda_3 \) of \( M = S^2 + \Omega^2 \) are determined, and ordered according to their value, such that \( \lambda_1 \leq \lambda_2 \leq \lambda_3 \). According to their definition a vortex is a region in space for which the matrix \( M \) has two negative eigenvalues. Note that \( M \) only has real eigenvalues, since it is symmetric. The surface of the vortex is defined by the set of points for which \( \lambda_1 \) is negative, and \( \lambda_2 \) is zero.

The most important problem associated with all region-type vortex detection methods is that they are particularly poor at isolating vortices. When two vortices are close together in the flow, they will usually show up as one vortex after detection, even if their sense of rotation is different. This makes vortex tracking (see Chapter 4) in particular very difficult, although it has been done, with limited success [37].
All vortex detection methods rely on certain heuristics, a direct consequence of the lack of a proper definition for a vortex. Some of these heuristics only apply to certain vortex types. For example, Banks & Singer explicitly state: “The method was designed for elongated vortices . . . rather than broad vortex sheets”. Furthermore, some of these heuristics are also valid for non-vortical structures in the flow field. This causes detection methods to suffer from several problems. First of all, false-positives may be detected, i.e., some of the detected features may not belong to a vortical structure. Secondly, some vortices present in the flow may remain undetected.

An excellent overview of vortex detection methods and a more detailed discussion of the detection methods mentioned above is given by Jiang & Thompson [24].

3.5 Parallel vector operator

Numerous vortex detection methods exist besides the vortex detection methods briefly mentioned in the previous paragraph. In 2000, Roth [34] uncovered a mathematical basis underlying most line-type vortex detection techniques. He proposed a new mathematical operator on a pair of vector fields, the so called parallel vector operator. As it turns out, most line-type vortex detection techniques can be expressed in terms of the parallel vector operator.

3.5.1 Definition

The parallel vector operator ($\parallel$) is applied to two vector fields $v$ and $w$, and results in a set of points $S$ for which the two vector fields are parallel. Let $v$ and $w$ be two $n$-dimensional vector fields, then $S$ is given by

$$S = \{x \in \mathbb{R}^n : v(x) = 0\} \cup \{x \in \mathbb{R}^n : \exists \lambda, w(x) = \lambda v(x)\}. \quad (3.3)$$

Alternatively, when $n = 2$ or $n = 3$, $S$ may be expressed as

$$S = \{x \in \mathbb{R}^n : v(x) \times w(x) = 0\}. \quad (3.4)$$

We can now give a definition of the parallel vector operator.

**Definition 3 (Parallel vector operator):**

For two $n$-dimensional vector fields $v$ and $w$, we denote by $v \parallel w$ the operator which returns the set $S$ (Equation 3.3) restricted to the non-degenerate solutions, i.e., excluding isolated points and points which have an $(n-1)$-dimensional neighborhood in $S$.

Typically, when expressing a vortex detection method in terms of the parallel vector operator, additional conditions are needed for the points in $S$ before they are considered to be part of a vortex core. These conditions vary based on the detection method implemented using the parallel vector operator.

In Roth’s work on the parallel vector operator several classic vortex detection methods are discussed and expressed in terms of the parallel vector operator. We implemented
several of these methods, including Banks & Singer’s predictor corrector method, as discussed above, and Sujudi & Haimes’ eigenvector method [40]. Besides placing classic vortex detection methods in the context of the parallel vector operator, Roth introduced a novel vortex detection method that we implemented as well.

To give an indication of how these individual vortex detection methods translate to the parallel vector operator, we briefly discuss Banks & Singer’s predictor corrector method in the next section, and show how it can be expressed in terms of the parallel vector operator.

### 3.5.2 Banks & Singer’s method using the parallel vector operator

Banks & Singer’s predictor-corrector method was the first method to construct vortex corelines as line-type features, and was first presented in 1994 [5]. From a predetermined starting point their algorithm steps along the vorticity vector and corrects each step along that vector in the direction of the pressure minimum on a plane perpendicular to the vorticity vector. Their algorithm is outlined in Listing 3.1. A schematic overview of the steps involved in Banks & Singer’s predictor-corrector algorithm is given in Figure 3.6.

![Figure 3.6](image)

**Figure 3.6:** Schematic overview of the steps involved in Banks & Singer’s predictor-corrector algorithm. Based on an illustration by Banks & Singer [5]. (a) the vorticity vector $\nu_i$ at $p_i$; (b) the predicted point $p'_{i+1}$ pointed to by the vorticity vector $\nu_i$; (c) the computed vorticity vector $\nu_{i+1}$ at $p'_{i+1}$; (d) the plane $P$ with normal $\nu_{i+1}$, centered at the local pressure minimum on $P$, which equals the corrected point $p_{i+1}$. The gray lines are pressure isolines. Note that the magnitude of the vector $p_{i+1} - p'_{i+1}$ may not exceed a certain threshold value, otherwise, the algorithm is terminated.

The two fundamental assumptions behind the predictor-corrector method are that a vortex coreline can be approximated by a vorticity line, i.e., a streamline of the vorticity field, and that the local pressure assumes a minimum at the vortex coreline. The algorithm terminates in case the prediction step differs too much from the local pressure minimum, i.e., the corrector step is too large. To start the algorithm, a collection of seed points needs to be determined. Banks & Singer suggest to restrict the collection of seed points to the points in the grid with low pressure and a high vorticity magnitude. Post-processing is required to prevent redundancy in the resulting coreline collection, as multiple seed points may result in the same coreline.
for each seed point p
    currentPoint = p;

while true do
    /* Calculate the vorticity vector at the current point. */
    vorticity = CalculateVorticity( currentPoint );

    /* Step in the direction of the vorticity vector. */
    nextPoint = currentPoint + vorticity;

    /* Calculate the vorticity vector at the next point. */
    nextVorticity = CalculateVorticity( nextPoint );

    /* Construct a plane centered at the predicted point, with
    * the normal equal to the vorticity vector at the next
    * point. */
    plane = ConstructPlane( nextPoint, nextVorticity );

    /* Determine the location of the pressure minimum on the
    * plane. */
    pressureMinimum = FindPressureMinimum( plane );

    /* In case the distance between the predicted point and the
    * location of the pressure minimum is smaller than a
    * certain threshold, take the location of the minimum as
    * the corrected point on the core and continue. */
    if | pressureMinimum - nextPoint | < threshold then
        currentPoint = nextPoint
    else
        break
    fi
od

Listing 3.1: Pseudo code for Banks & Singer’s predictor-corrector method.

The underlying assumption of Banks & Singer’s algorithm is that the direction of the vorticity vector has little variation compared to the variation of pressure. Thus, using this assumption, the predictor-corrector algorithm results in points where the vorticity vector is approximately parallel to the pressure gradient. In terms of the parallel vector operator defined in Definition 3, the corelines returned by Banks & Singer’s algorithm consist of points for which

$$\nabla \times v \parallel \nabla p$$

holds. As mentioned in Section 3.5.1, usually the parallel vector condition is not sufficient. That statement is true for this case as well, since the corepoints found using Equation 3.5 may also have a maximum pressure in the plane perpendicular to the vorticity. Therefore, we need to add the additional condition that the corepoints returned are located at a local pressure minimum.
3.6 Implementation

The parallel vector operator is implemented as an iterative method on a set of faces $F_{c_{i,j,k}}$ per grid cell $c_{i,j,k}$. Our implementation handles any one of the following grid types, in increasing order of generality:

**Uniform grid:** a uniform grid is described by three numbers $x_n$, $y_n$, and $z_n$ indicating the number of sample points along the $x$, $y$, and $z$ axis respectively. The distance between consecutive sample points along an axis is constant.

**Rectilinear grid:** differs from a uniform grid in the sense that the distance between the sample points along an axis is not constant. The topology is defined implicitly by specifying the grid dimensions, whereas the geometry is specified by a list of separate $x$, $y$, and $z$ coordinates.

**Structured grid:** a structured grid has regular topology and irregular geometry. The topology of the grid is defined implicitly by specifying the grid dimensions. The geometry is represented by an array of point coordinates. The grid may be warped into any configuration as long as the individual cells do not overlap or self intersect.

Note that when we process a grid cell, we only need to consider a subset of all the cells' faces, because every grid cell shares a certain amount of faces with its neighboring cells. Figure 3.7 illustrates an $8 \times 8 \times 8$ dimensional rectilinear grid along with two grid cells $c_{6,7,0}$ and $c_{6,1,0}$, with the faces that are in $F_{c_{6,7,0}}$ and $F_{c_{6,1,0}}$ respectively highlighted.

![Figure 3.7](image)

*Figure 3.7:* (a) grid cell $(6, 7, 0)$, with the shaded faces belonging to the set of faces $F_{c_{6,7,0}}$ that are considered during the detection process. Note that cell $(6, 7, 0)$ is a boundary cell. To cover all the faces of the cells in the grid, the right faces of the cells at the right boundary of the grid need to be processed as well. Similarly, the top and back faces of the top and back boundary cells need to be in $F$ as well. Therefore, the top face of grid cell $(6, 7, 0)$ is part of $F_{c_{6,7,0}}$ as well; (b) grid cell $(6, 1, 0)$. Again, the shaded faces belong to the set of faces $F_{c_{6,1,0}}$. For every cell $(i, j, k)$ in a $I \times J \times K$ dimensional grid, with $0 \leq i < I - 1 \wedge 0 \leq j < J - 1 \wedge 0 \leq k < K - 1$, the set of faces $F_{c_{i,j,k}}$ is made up of the left, front, and bottom cell face as depicted in this subfigure.
The algorithm detecting the points part of the set from Equation 3.3 then proceeds as follows. Assume we are looking for corepoints in a 3D grid. For every face $\psi \in F$ we try to find zeros for the function $h: [0,1]^2 \rightarrow \mathbb{R}^3$, defined over a grid face and given by

$$h(\sigma_i, \sigma_j) = v(\sigma_i, \sigma_j) \times w(\sigma_i, \sigma_j).$$

(3.6)

Here, $\sigma_i$ and $\sigma_j$ are the parameters describing positions on the grid face. The vector fields $v$ and $w$ vary, depending on the detection method, and are defined on the four corner points $\psi$. We use bilinear interpolation to determine $v(\sigma_i, \sigma_j)$ and $w(\sigma_i, \sigma_j)$. Figure 3.8 depicts an example graph resulting from $|h(\sigma_i, \sigma_j)|$ defined over a cell’s bottom face. The graph in Figure 3.8 is a contrived example. In practice, since $|h(\sigma_i, \sigma_j)|$ is interpolated using the four vectors defined at the cell face corners, the graph will be less fluctuating.

Figure 3.8: An example graph of $|h(\sigma_i, \sigma_j)|$ defined over the gray-shaded cell face.

We use the Newton-Raphson method to find a zero for $h$ defined over $\psi$. For the 1D case, the $n$-th Newton-Raphson step in the direction of the root for a function $f: \mathbb{R} \rightarrow \mathbb{R}$ is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$ 

(3.7)

Thus, the correction for $x$, $\Delta x$ is a solution of

$$f'(x_n) \Delta x = -f(x_n).$$

(3.8)

The partial derivative of $h$ with respect to $\sigma_i$ is given by:

$$\frac{\partial h}{\partial \sigma_i} = \frac{\partial (v \times w)}{\partial \sigma_i} = \frac{\partial}{\partial \sigma_i} \left( \begin{array}{c} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{array} \right)$$

$$= \frac{\partial v}{\partial \sigma_i} \times w + v \times \frac{\partial w}{\partial \sigma_i}.$$ 

(3.9)

Here, we used $v_x$ as a shorthand notation for $v_x(\sigma_i, \sigma_j)$. The two partial derivatives of $h$ form a $3 \times 2$ matrix that we call $J_h$. Using Equation 3.8, the correction vector $\Delta \sigma$ is a
solution of

\[ J_h \Delta \sigma = -h. \]  \hspace{1cm} (3.10)

This system is over-determined, as we have only two unknowns, \( \Delta \sigma_i \) and \( \Delta \sigma_j \), and three components of \( h \). Equation 3.10 is a linear least squares problem, for which the following normal equation gives the least squares solution

\[ J_h^T J_h \Delta \sigma = -J_h^T h. \]  \hspace{1cm} (3.11)

Since it is only a \( 2 \times 2 \) system, it is reasonable to calculate the solution to 3.11 analytically, resulting in

\[ \Delta \sigma = - (J_h^T J_h)^{-1} J_h^T h \]  \hspace{1cm} (3.12)

for the correction of \( \sigma \) for the Newton-Raphson step.

Initially we use \( \sigma_i = \sigma_j = 0.5 \), i.e., the center of the face under consideration as the start point of the Newton-Raphson iteration. The Newton-Raphson iteration terminates when either a root of \( h \) is found, or the number of iterations exceeds a certain threshold \( T \). For a point \((\sigma_i, \sigma_j)\) to be considered a root of \( h \), \( 0 \leq |h(\sigma_i, \sigma_j)| \leq \varepsilon \) needs to hold, with \( \varepsilon \) a sufficiently small value. In addition to checking for convergence of the magnitude of \( h \) towards zero, we also check for convergence of \( |\Delta \sigma| \) towards zero. In case convergence of the correction step is observed, we calculate \( |h(\sigma_i, \sigma_j)| \), and accept it as a parallel point in case \( 0 \leq |h(\sigma_i, \sigma_j)| \leq \xi \) holds, with \( \varepsilon < \xi \), and \( \xi \) sufficiently small.

### 3.7 Improved corepoint detection

A significant drawback of the parallel vector method as discussed in section 3.6 stems from the use of the Newton-Raphson root-finding method. The choice of the initial start point for the Newton-Raphson iteration may influence the outcome of the algorithm. Given a face containing a corepoint \( c \), the algorithm does not guarantee the detection of \( c \). Whether \( c \) is successfully detected or not depends on the initial choice for \( \sigma_i \) and \( \sigma_j \).

Assume we try to find the root of a function \( f(x) \). Equation 3.13 defines the relation between the first and second derivative at the root of the function \( f \) and the error at step \( n + 1 \) of the iteration process [4];

\[ \epsilon_{n+1} \approx \frac{f''(s)}{2f'(s)} \epsilon_n^2. \]  \hspace{1cm} (3.13)

Now assume that the first derivative \( |f'(x)| \) is very small near a root \( r \) of \( f(x) \). Geometrically, a small \( |f'(x)| \) means that the tangent of \( f(x) \) near \( r \) almost coincides with the \( x \)-axis, so that the Newton-Raphson method may diverge from the root instead of converging to it. Using Equation 3.13, we can see that a small \( |f'(x)| \) near \( r \) may cause the error to become greater than 1. Similarly, using Equation 3.13, in case the second derivative \( |f''(x)| \) is sufficiently large near the root divergence may occur as well.

Using the values of \( |f'(x)| \) and \( |f''(x)| \) Adams [4] derives error bounds for an initial starting point that guarantee convergence for a function \( f \), provided that \( f \) has a root.
and the initial starting point is sufficiently close to that root. In practice this method is rarely used, since it is generally more efficient to determine whether the Newton-Raphson method converges to a certain point after a set number of iterations. In case no convergence is observed, the iteration is aborted. Our implementation employs the latter approach.

To alleviate for the negative impact the choice of the initial values for $\sigma_i$ and $\sigma_j$ may have on the final result of the vortex detection algorithm, we decided to generate a rectangular $n \times n$ grid of $N = n^2$ candidate starting points for every face, as depicted in Figure 3.9.

![Figure 3.9: A $4 \times 4$ starting point grid on top of a cell face.](image)

Now, the detection algorithm proceeds by picking the $i$-th point from the grid of possible starting points, with $i < N - 1$. In case no corepoint is found for starting point $i$, point $i + 1$ is chosen, and the detection algorithm starts again. This process is repeated until all $N$ starting points have been considered, or a corepoint is found. Using this extension to the original parallel vector algorithm, better detection results were obtained.
The graphs in Figure 3.10 illustrate the relation of the number of points in the start point grid with respect to the number of vortices detected and the amount of time required to complete the detection process. Several runs with different datasets were performed, and the results were averaged. Note that in graph (a), for the default detection method where the starting point grid consists of only one point, the vortex index number is set to 100. Thus, for a starting point grid consisting of 4 points, the number of detected vortices increases by an average 7%. The tests were performed on an AMD XP 2600+ machine using $121 \times 49 \times 96$ dimensional flow datasets.

Note that although the amount of extra candidate corepoints is relatively small compared to the total number of candidate corepoints found, using the starting point grids proves to be beneficial. The reason is that finding one previously missed corepoint may result in one complete extra vortex ending up in the final visualization. For example, in case a user is trying to detect vortices containing at least 15 corepoints, failing to detect the middle corepoint of a vortex core containing 29 corepoints due to a problematic starting point will result in the complete vortex to be ignored.
Chapter 4

Vortex Tracking

The feature detection method described in Chapter 3 enables the detection of features for a dataset corresponding with a certain time step $t_i$. We can apply the same detection algorithm for other time steps, and build an animation by combining the separate visualizations for every time step. However, the resulting animation will be unstable, and will probably contain considerable noise in the form of features present for only a small range of time steps.

4.1 Tracking theory

To enable the implementation of a convincing time-dependent visualization and a notion of motion and evolution of the detected features over time, we need to determine the correspondence between features in successive time steps that actually represent the same objects. This is called the correspondence problem. Here we define in abstract terms a mapping $M$ from a feature detected in time step $t_i$ to one or more features detected in time step $t_{i+1}$ as

$$ M : \mathcal{F} \rightarrow \mathcal{F}^P $$

where $\mathcal{F}$ denotes the set of features. Assume we have a function $\phi : \mathbb{R} \rightarrow \mathcal{F}$, returning the set of detected features for a certain time step $t_i \in \mathbb{R}$. Then, for a certain feature $\eta \in \phi(t_i)$, $M(\eta)$ returns a set of features from time step $t_{i+1}$ evolving from $\eta$.

Using this mapping $M$, we can prevent the popping effect caused by features which are only present for small time ranges, since from $M$ we are able to derive the lifetime of features. Furthermore, we can detect certain events during the lifetime of a feature. The events that may occur were first described by Samtaney et al. [35] and are: continuation, creation, dissipation, bifurcation, and amalgamation.

For continuation, a feature detected in time step $t_i$ continues to the next time step $t_{i+1}$ as one single feature. Thus, for a feature $\eta \in \phi(t_i)$, continuation occurs when $|M(\eta)| = 1$. The properties associated with $\eta$ may of course differ for the two time steps.

In case of creation, a new feature appears for a certain time step $t_i$ that can not be matched with a feature from a previous time step. In terms of $M$, this translates to a feature $\eta \in \phi(t_i)$ for which $(\forall f : f \in \phi(t_{i-1}) : \eta \notin M(f))$ holds. On the other hand, dissipation occurs when a feature disappears after a certain time step, or alternatively,
when for $\eta \in \phi(t_i)$, $M(\eta) = \emptyset$ holds.

For bifurcation, a feature breaks up in two or more features in the next time step, whereas for amalgamation, two or more features merge to one feature in the next time step. Thus, a bifurcation event occurred for a feature $\eta \in \phi(t_i)$ when $|M(\eta)| \geq 2$ holds. An amalgamation event on the other hand occurred when $(\exists f, g : f, g \in \phi(t_{i-1}) : \eta \in M(f) \land \eta \in M(g))$.

Reinders et al. [30] added four additional events to this list, entry and exit events, occurring when features move beyond the boundaries of a dataset, and loop and junction events corresponding with changes in the topology for certain graph-type features.

Note that in practice, the previously described conditions for the events described by Samtaney may not be sufficient, requiring additional conditions to hold to compensate for noisy or rough datasets. For example, it is unlikely for a feature to bifurcate between time step $t_i$ and $t_{i+1}$, and subsequently amalgamate between $t_{i+1}$ and $t_{i+2}$. All of the events we just described directly apply to the feature we are interested in: the vortex core lines.

### 4.2 State of the art

Several approaches exist to solve the correspondence problem when tracking features over time. The first approach implicitly solves the correspondence problem by extracting features from the spatio-temporal domain, usually by linearly interpolating between discrete time steps. Alternatively, feature tracking algorithms have been described operating solely in the space domain, where features between different time steps are matched based on region or attribute correspondence.

Region correspondence methods compare the regions of interest obtained from the feature extraction process between successive time steps. Corresponding regions can be found by determining the spatially overlapping regions extracted from four-dimensional time-dependent datasets using isosurfaces. This method however is error-prone, as corresponding regions are not required to spatially overlap between successive time steps. Furthermore, an overlapping region is not guaranteed to belong to the same feature. Silver and Wang [39, 37] employ this method for feature tracking, and prevent the previously mentioned associated problems by checking the volumes of the corresponding features, and taking the best match.

Another method of feature tracking focuses on the feature’s attributes, and is called attribute correspondence. For this method, tracking proceeds by matching features with similar attributes like position, size, volume, diameter, or orientation between successive time steps. For example, Samtaney et al. [35] create correspondence criteria based on user-provided tolerances for certain attribute values.

Tricoche et al. [41] proposed an algorithm that falls in the first category of algorithms implicitly solving the correspondence problem by operating in the spatio-temporal domain. Their algorithm enables tracking of two-dimensional vector field topologies by interpolating in the three-dimensional spatio-temporal domain.
4.3 Vortex tracking using the parallel vector operator

Our implementation of a feature tracking algorithm is loosely based on a tracking algorithm first proposed by Bauer & Peikert [8]. It is an optimized version of their tracking algorithm, and differs in the way the final streamsurface triangulation is performed. We first describe the original tracking algorithm of Bauer & Peikert, after which a discussion of our version of the algorithm follows.

4.3.1 Bauer & Peikert’s tracking algorithm

Bauer & Peikert’s tracking algorithm is based on the parallel vector operator, and operates in the spatio-temporal domain. The 3D space is extended with the tracking dimension, which in this case is the time dimension. To understand the reasoning behind this scheme, suppose we aim to track 0D features in a 1D space, as illustrated in Figure 4.1. The cells in 1D space, indicated by the line segments along the space axis, are extended with the time dimension, resulting in 2D space-time cells. Furthermore, assume we have a feature detection algorithm operating on 1D line segments, used to detect the features in time step $t = 0$ and $t = 1$. Tracking now proceeds by applying the tracking algorithm on the 1D space-time line segments, resulting in the features with time location $\tau$, with $0 < \tau < 1$. Projecting a feature $g$ detected along a space-time line segment with time location $\tau$ on our original 1D space results in the location of $g$ in our original 1D space.

For example, in Figure 4.1, projecting the features detected along the space-time line segments results in the position of feature $g$ at $t = 0.5$ and $t = 0.85$, resulting in $x = 1$ and $x = 2$ respectively. Subsequently, using heuristics, $g$ is connected with features sharing the same space-time cell.

![Figure 4.1](image_url): Tracking of 0D features in 1D space. The line segments along the space axis illustrate the space cells, whereas the squares in the graph illustrate 2D space-time cells. The dashed lines indicate the boundaries between the space-time cells. Image based on Figure 3 in ‘Vortex Tracking in Scale Space’ [8].

In general, $n$D features result in $(n + 1)$D structures as a result of extending the original $m$D space. For example, in Figure 4.1, tracking the 0D point features results in 1D lines. The vortex coreline features we are focusing on are found in a 3D flow dataset with regular hexahedral grid cells, i.e., cubes. Extending a 3D cube with an extra dimension results in a 4D hypercube or tesseract, as depicted in Figure 4.2. The 1D corelines result in 2D streamsurfaces.

When looking at the hypercube resulting from lifting a 3D grid cell, it seems that the grid cell at $t = 0$ is smaller than the grid cell of time step $t = 1$, but this is, of course, not the case. The configuration of the eight cubes as depicted in Figure 4.2 is only one
CHAPTER 4. VORTEX TRACKING

Figure 4.2: A 4D hypercube or tesseract. The hypercube consists of 8 3D cubes, 24 faces, 32 edges, and 16 vertices. One cube is located in the center of another surrounding cube. The remaining six cubes are located between each corresponding face of the inner and outer cube. The 4D hypercube that results from lifting a 3D hexahedral grid cell. The inner cube represents the grid cell at time step $i$, whereas the outer cube represents the grid cell at time step $i + 1$. The time axes are the edges connecting both cubes indicated by arrows.

of many possible configurations, and only serves to picture the fourth axis, i.e., the time axis.

The tracking algorithm of Bauer & Peikert now proceeds as follows; given two vector fields $v(x, y, z, t)$ and $w(x, y, z, t)$ defined for each vertex of the hypercube resulting from lifting a grid cell, find the set of points where the two vector fields are parallel by applying the detection algorithm from Chapter 3 for all the faces of the eight boundary cubes. Subsequently, for each boundary cube the parallel points detected on each face of the boundary cube are connected using the same heuristics as defined in Chapter 3. Then, the line segments are projected back to 3D space, and the time dimension is stored as a vertex attribute. Finally, for each grid cell, the projected line segments forming polygons are triangulated resulting in a small patch of the streamsurface. The algorithm is outlined in Listing 4.1, and an example scenario is depicted in Figure 4.3. Note that the corepoints found on the space-time faces of the six space-time cubes are basically intersections of the moving coreline with one of the eight grid cell edges.

Figure 4.3: (a) a grid cell at time step $i$ for which a corepoint was found on the front and right face, together forming one segment of a coreline; (b) the resulting hypercube after lifting the grid cell from time step $i$ to $i + 1$. One space-time corepoint was found and connected with the corepoints from time step $i$; (c) the result after projecting the two space-time line segments back to 3D space.
ReadDataSet( 0 );
for i = 1 to numberOfDataSets − 1
    ReadDataSet( i );
    for each grid cell
        ConstructHyperCube();
        for each boundary cube
            /* Detect a coreline in the boundary cube, by
            * applying the algorithm from Chapter 4 for each
            * face of the cube. */
            DetectCoreline();
            /* Project coreline from 4D to 3D. */
            ProjectCoreline();
        next boundary cube
        /* Triangulate the polygon made up of all projected
        * line segments to form a streamsurface patch. */
        Triangulate();
    next grid cell
    ReleaseDataSet( i − 1 );
next i

Listing 4.1: Pseudo code for Bauer & Peikert’s tracking algorithm

4.4 Optimized tracking algorithm

A drawback of Bauer & Peikert’s tracking algorithm is that the complete vortex detection process is executed for all the faces of all of the eight boundary cubes, significantly increasing the amount of time required for detecting a streamsurface. By taking advantage of the topology of a typical grid we can limit the amount of boundary cubes that need to be considered per grid cell to three, since two neighboring grid cells share equivalent boundary cubes.

For the following examples, assume we have a \( I \times J \times K \) rectilinear grid. Figure 4.4 illustrates two neighboring hypercubes \( A \) and \( B \) with grid coordinates \( (i, j, k) \) and \( (i + 1, j, k) \) respectively, with \( 0 \leq i < I − 1, 0 \leq j < J, \) and \( 0 \leq k < K \). The highlighted boundary cubes result from lifting the cell face shared by both grid cells, resulting in equivalent boundary cubes.

Now assume we start the detection process for the lower left grid cell at \( (0,0,0) \). Processing grid cell \( (p, q, r) \), with \( 0 \leq p < I − 1, 0 \leq q < J − 1, \) and \( 0 \leq r < K − 1 \) then only requires the left, front, and bottom boundary cubes to be considered. The right boundary cube is considered when processing cell \( (p + 1, q, r) \), the top boundary cube is considered when processing cell \( (p, q + 1, r) \), and finally the back boundary cube is considered when processing cell \( (p, q, r + I \cdot J) \). Note that for a cell \( (i, j, k) \), with
\( i = I - 1 \lor j = J - 1 \lor k = K - 1 \), the boundary cubes resulting from lifting non-shared cell faces need to be considered as well to cover the complete extent of the grid.

This still requires separate vortex detection steps for the boundary cube grid cells. As previously discussed in Chapter 3, each grid cell requires three faces to be considered resulting in a total of nine additional faces that need to be considered per grid cell for Bauer & Peikert’s algorithm.

This number can be further reduced by looking at the faces shared by the three boundary cubes. In Figure 4.5, the faces shared by the three boundary cubes are illustrated. For a hypercube that results from lifting grid cell \((p, q, r)\), with \(0 \leq p < I - 1\), \(0 \leq q < J - 1\), and \(0 \leq r < K - 1\), it is sufficient to only consider the three shaded space-time faces. Processing the neighboring boundary cubes will then result in complete coverage of all space-time faces. For example, assume we are considering a hypercube that results from lifting grid cell \((p, q, r)\), with \(0 \leq p < I - 1\), \(0 \leq q < J\), and \(0 \leq r < K\). The space-time face \(F\) depicted in Figure 4.5 is ignored when applying the tracking algorithm to grid cell \((p, q, r)\). Instead, it is processed as space-time face \(G\) for the boundary cube that results from lifting grid cell \((p + 1, q, r)\). The same rationale can be applied for all other non-shaded space-time faces. Again, for a grid cell \((i, j, k)\) with \(i = I - 1 \lor j = J - 1 \lor k = K - 1\), all non-shared space-time faces need to be considered as well to cover the complete space-time dimension of the lifted grid.

**Figure 4.4:** Two neighboring hypercubes, for which the highlighted boundary cubes are equivalent.

**Figure 4.5:** (a) shows the three boundary cubes typically considered when processing a grid cell during the tracking process, with the faces shared by these three boundary cubes shaded gray; (b) shows the space-time cell formed by the three shared boundary cube faces.
The resulting space-time cell as illustrated in Figure 4.5 consists of the same number of faces for the non-degenerate space cell from Chapter 3. Therefore, by choosing a proper vertex ordering for every individual face, we can seamlessly apply the corepoint detection algorithm from Chapter 3 to a space-time cell.

In conclusion, this approach requires the corepoint detection algorithm to be applied for three additional faces per grid cell. By combining these three faces in a space-time cell, the vortex detection algorithm described in Chapter 3 can be applied. In total, for a grid cell \((p, q, r)\) with \(0 \leq p < I \land 0 \leq q < J \land 0 \leq r < K\), our approach requires the corepoint detection algorithm to be applied to nine faces, whereas Bauer & Peikert’s algorithm requires the algorithm to be applied to 15 faces, assuming shared boundary cubes are only processed once. The number of faces processed per grid cell largely determines the total amount of time required for the entire tracking process, as it is by far the most expensive operation. Therefore, since the amount of faces that need to be considered has been decreased by \(40\%\), it is reasonable to argue that the total time required for the tracking process is decreased by approximately \(40\%\).

\[\text{Figure 4.6: Example situation where Bauer & Peikert’s algorithm might result in an incorrect streamsurface patch.} \]

(a) initially, we have a grid cell at time step \(i\) containing two coreline segments, \(C\) and \(D\). Both are moving in the same direction, indicated by the arrow; (b) at the next time step, \(i + 1\), \(D\) has left the grid cell under consideration, while \(C\) is roughly at the same position as \(D\) was at time step \(i\); (c), (d) show the process of tracing \(C\) and the resulting streamsurface patch assuming \(D\) is not present; (e), (f) show the process of tracing \(D\) and the resulting streamsurface patch assuming \(C\) is not present; (g) illustrates the space-time corelines created by connecting the corepoints in every boundary cube when both coreline segments \(C\) and \(D\) are considered. Note that for the front boundary hypercube incorrect connections were created; (h) shows the final incorrect streamsurface patches after projecting the space-time corelines.

Our approach requires a different streamsurface triangulation method, as our space-time cell datastructure does not allow for corelines to be detected in the boundary cubes. Due to the introduction of the space-time cell, there is no real notion of a boundary cube in the datastructure; it would require an additional post-processing step to reconstruct
them. In the next paragraph, our triangulation method based on streamsurface boundary propagation is discussed.

4.4.1 Streamsurface boundary propagation

Bauer & Peikert employ an ad-hoc triangulation method after they have detected the corelines lying in the space-time domain. As discussed in the previous paragraph, the 4D corelines are projected to 3D space, and the resulting polygons are subsequently triangulated. The drawback of their method is that the heuristics used to connect corepoints to form corelines can cause problems for the corelines detected in the boundary cubes. For example, refer to the scenario sketched in Figure 4.6.

In particular, Figure 4.6 (g) illustrates the ambiguity the coreline connection algorithm is faced with. Based on heuristics, a good choice has to be made to prevent a malformed streamsurface patch from forming after triangulation. Note that theoretically it is possible to define proper heuristics for these kind of situations, but that it requires additional logic to be added to the original coreline detection algorithm, thereby removing the advantage of being able to seamlessly integrate the vortex detection algorithm from Chapter 3.

The cause of the problem arising here is that streamsurface triangulation is reduced to a local grid cell problem, without taking into account global flow information like the time frame or coreline a certain corepoint belongs to.

Our streamsurface triangulation method is taking a more global approach by looking at a complete coreline, and gradually propagating it using candidate space-time corepoints and corepoints belonging to corelines from the next time step. What follows is a general description of our complete tracking algorithm, in particular the streamsurface propagation algorithm and the heuristics used to generate a proper streamsurface.

Our tracking algorithm up to this point is listed in Listing 4.2. We have discussed the whole algorithm up to the streamsurface generation step.

In the following discussion, when we refer to a coreline or a streamsurface, we refer to their mesh representations, consisting of triangles, edges, and vertices. Additionally, the following notation is employed:

- $E(M)$ returns the collection of edges belonging to mesh $M$. The two vertices of an edge $e$ are denoted by $e_0$ and $e_1$. The collection of all edges is written as $E$.
- $V(M)$ returns the collection of vertices belonging to mesh $M$. The time attribute of a vertex $v$ is denoted by $v_t$. The collection of all vertices is written as $V$.
- $T(M)$ returns the collection of triangles belonging to mesh $M$. The collection of all triangles is written as $T$.

Assume we are currently tracking vortices between time step $i$ and $i+1$. Furthermore, assume we have detected $N$ corelines for time step $i$. Now, for every coreline $L_{i,j}$ in time step $i$, with $0 \leq j < N$, we initialize the streamsurface of $L_{i,j}$, designated by $S_{i,j}$, using all the edges from $L_{i,j}$. Thus, initially $E(S_{i,j}) = E(L_{i,j})$ as well as $V(S_{i,j}) = V(L_{i,j})$ holds. Every vertex $v \in V(L_{i,j})$ has an additional time attribute $v_t$ equal to $i$. Before defining the boundary of a streamsurface, we define a function $I : E \times E \rightarrow \mathbb{B}$ as

$$I(e, e') = (e_0 = e'_0 \land e_1 \neq e'_1) \lor (e_0 = e'_1 \land e_1 \neq e'_0). \quad (4.2)$$
ReadDataSet( 0 );

/* Here, we apply the unmodified algorithm from Chapter 4. */
DetectCorepoints( GetDataSet( 0 ) );
DetectCorelines( 0 );

for i = 1 to numberOfDataSets − 1
    ReadDataSet( i );

    /* Here, we apply the unmodified algorithm from Chapter 4. */
    DetectCorepoints( GetDataSet( i ) );

    /* Using the datasets from time step i − 1 and i, construct
     * a grid consisting of space–time cells, and use the
     * algorithm from Chapter 4 to detect the space–time
     * corepoints. */
    ConstructSpaceTimeGrid( i − 1, i );
    DetectCorePoints( GetSpaceTimeGrid( i − 1, i ) );

    /* Generate the streamsurfaces. */
    for each coreline l in time step i
        PropagateCoreline( l );
    next coreline;

    ReleaseDataSet( i − 1 );
next i

Listing 4.2: Pseudo code for the adapted tracking algorithm.

Thus, \( I(e, e') \) indicates whether the two edges \( e \) and \( e' \) are neighboring edges in the sense that they share one and only one vertex, as illustrated in Figure 4.7. Additionally, we define a function \( N_{i,j} : E \rightarrow E \) for a streamsurface \( S_{i,j} \) as

\[
N_{i,j}(e) = \left( \cup e' \in \mathcal{E}(S_{i,j}) : e \neq e' : I(e, e') \right).
\] (4.3)

For an edge \( e \in S_{i,j} \), \( N_{i,j}(e) \) returns the collection of edges sharing a vertex with \( e \).

An edge \( e \in \mathcal{E}(S_{i,j}) \) is part of a triangle in case there are two edges in \( N_{i,j} \) that share a vertex, or formally

\[
T(e) = \left( \exists f, g \in N_{i,j}(e) : f \neq g : I(f, g) \right).
\] (4.4)

By looking at the vertex time attributes of the three edges making up a triangle, we can determine whether an edge \( e \) is ahead in time with respect to another edge \( f \) by adding the vertex time attributes of every edge vertex and comparing the sums. Using Equation 4.3 and 4.4, we define a function \( A_{i,j} : E \rightarrow B \) returning whether for all
triangles an edge \( e \in \mathcal{E}(S_{i,j}) \) is part of, it is the newest edge of all these triangles

\[
A_{i,j}(e) = (\forall f, g : f, g \in N_{i,j}(e) : I(f, g) \land (f_0 + f_1 + g_0 + g_1 \leq 2e_0 + 2e_1)). \tag{4.5}
\]

Then, using Equation 4.5, the boundary of a streamsurface is a collection of edges \( B_{i,j} \), with \( B_{i,j} \subseteq \mathcal{E}(S_{i,j}) \), defined by

\[
B_{i,j} = (\forall : e \subseteq \mathcal{E}(S_{i,j}) : A_{i,j}(e)). \tag{4.6}
\]

Since it is not efficient to calculate the streamsurface boundary using Equation 4.6 every time the streamsurface is altered, we now try to give an incremental version of the streamsurface boundary definition given above.

The streamsurface generation algorithm is based on propagating the streamsurface boundary \( B \) associated with every coreline. Initially, for a coreline \( L_{i,j} \), the associated streamsurface boundary is equal to the coreline itself, thus \( B_{i,j} = \mathcal{E}(L_{i,j}) \). Now, for every edge in the streamsurface boundary, we try to propagate it to a coreline from the next time frame, or a space-time corepoint.

In Figure 4.8, the initial situation is illustrated. The current edge \( e \) under consideration is shown along with the grid cell \((u, v, w)\) it is located in. The grid cell belongs to a rectilinear \( U \times V \times W \) grid, with \( 0 < u < U - 1 \land 0 < v < V - 1 \land 0 < w < W - 1.\) Every corepoint has a cell collection property \( C \), containing the set of cells the corepoint belongs to. In the unlikely case a corepoint is detected at the corner point of a face, it is part of eight different cells. In the current example, the cell collection \( C \) for corepoint \( c_i \) is \( C_{c_i} = \{(u-1, v, w), (u, v, w)\} \). For corepoint \( d_i \), \( C_{d_i} = \{(u, v, w), (u+1, v, w)\} \) holds. We introduce edge \( e \) as the edge formed by connecting corepoints \( c_i \) and \( d_i \). The cell \( e \) belongs to is the intersection of the cell collections of both corepoints making up the edge. Thus, the cell edge \( e \) belongs to is given by \( C_{c_i} \cap C_{d_i} = \{(u, v, w)\} \). In the following discussion, for an edge \( e \), \( e_c \) equals the cell edge \( e \) is located in.

Assume we have a collection \( \mathcal{W} \) of \( N \) space-time points, containing all the corepoints that were detected in the space-time domain. Space-time points are always located at the edges of the grid cells. Therefore, a space-time point usually belongs to four cells. In the unlikely case the space-time point is located at one of the corner points of a grid cell, it is part of eight cells. Every point \( s_i \in \mathcal{W} \), with \( 0 \leq i < N \), has an associated time attribute \( s_{ti} \). A propagation step from time step \( i \) to time step \( i + 1 \) now proceeds as follows. First, the algorithm checks to see whether any space-time corepoint for which \( i < s_{ti} < i + 1 \) holds qualifies as a candidate propagation point. In this case, a candidate
Figure 4.8: (a) coreline $L_{i,j}$ and $L_{i+1,k}$ passing through grid cell $(u, v, w)$ and $(u, v, w - 1)$ respectively, with edge $e$ the edge formed by corepoints $c_i$ and $d_i$ highlighted; (b) the situation after propagating edge $e$. The highlighted edges are part of the streamsurface boundary $B_{i,j}$ associated with streamsurface $S_{i,j}$.

propagation point is defined as a corepoint $p$ for which the function $G : \mathbb{R}^3 \times E \rightarrow B$ holds, given by

$$G(p, g) = e_0 + e_1 < 2p_t \land e_0 \neq p \land e_1 \neq p \land g_c \in C_p. \quad (4.7)$$

In our example, two candidate space-time corepoints $s_0$ and $s_1$ are found. Here, $C_{s_0} = \{(u, v, w), (u, v - 1, w), (u - 1, v, w), (u - 1, v - 1, w)\}$ holds, which implies that $(c_i, d_i)_c = (u, v, w) \in C_{s_0}$ holds as well. Also, both $i < s_0 < i + 1$ and $i < s_1 < i + 1$ hold, which implies that both $c_{it} + d_{it} < 2s_0$ and $c_{it} + d_{it} < 2s_1$ hold, as $c_{it}$ and $d_{it}$ both equal $i$. Obviously, $c_i \neq s_0 \land d_i \neq s_0$ holds as well, and therefore, $G(s_0, e)$ holds. Similarly, $G(s_1, e)$ holds as well. Edge $e$ will now be propagated to the nearest space-time corepoint. Where the distance of a space-time corepoint $s$ to the edge $e$ is calculated by taking the average distance between $s$ and the two points making up edge $e$. Theoretically, it is possible that multiple candidate space-time corepoints remain after this selection step. Later, we will introduce heuristics enabling us to calculate which one of the candidate space-time corepoints will result in a better streamsurface mesh. In this case, edge $e$ is propagated to $s_0$. Propagation is done by introducing two new edges $f$ and $g$, with $f = (c_i, s_0)$ and $g = (d_i, s_0)$. The new streamsurface boundary $B_{i,j}$ is now calculated using Equation 4.8. Note that in case no candidate propagation points were
found for edge \( e \) it is removed from \( B_{i,j} \) as well.

\[
B'_{i,j} = (B_{i,j} \backslash \{e\}) \cup \{f, g\}
\]  

The triangle made up by the edges \( e, f, \) and \( g \) is added to the streamsurface mesh. Every vertex \( v \) of the new triangle is assigned an additional time attribute \( v_t \). For the vertices resulting from corepoints \( c_i \) and \( d_i \), \( c_{it} = i \land d_{it} = i \) holds. The time attribute of the vertex corresponding with corepoint \( s_0 \) equals \( s_0_t \). Note that the predicate \( A_{i,j} \) holds for both edge \( f \) and \( g \). Edge \( e \) and \( g \) are both part of \( N_{i,j}(f) \). Furthermore, \( I(e, g) \) holds. Finally, \( e_0 + e_1 + g_0 + g_0 = 1 \leq f_0 + f_1 = 1 \) holds. Similarly it can be shown that \( A_{i,j}(g) \) is true. Figure 4.8 shows the resulting streamsurface triangle.

In case no candidate space-time corepoint is found, the algorithm searches all the corelines from the next time step \( i + 1 \) for a candidate corepoint that can be connected with the current edge \( e \). An example situation is illustrated in Figure 4.9. Again, the current edge \( e \) under consideration is shown along with the grid cell \((u, v, w)\) it is located in. As in the previous example, the cell collection \( C \) for corepoint \( c_i \) is \( C_{c_i} = \{(u - 1, v, w), (u, v, w)\} \). For corepoint \( d_i \), \( C_{d_i} = \{(u, v, w), (u + 1, v, w)\} \) holds. We introduce edge \( e \) as the edge formed by connecting corepoints \( c_i \) and \( d_i \). Again, function \( G \) as defined in Equation 4.7 is used to determine whether a corepoint qualifies as a candidate propagation point.

In this example, two candidate corepoints \( c_{i+1} \) and \( d_{i+1} \) are found. Edge \( e \) will now be propagated to either corepoint \( c_{i+1} \) or \( d_{i+1} \). Assume that edge \( e \) is propagated to corepoint \( c_{i+1} \). As mentioned previously, later in this section we introduce heuristics enabling us to determine the candidate point resulting in the best looking final mesh.
Propagation is done by introducing two new edges $f$ and $g$, with $f = (c_i, c_{i+1})$ and $g = (d_i, c_{i+1})$. The new streamsurface boundary $B_{i,j}$ is now calculated using Equation 4.8. Note that in case no candidate propagation points were found for edge $e$ it is removed from $B_{i,j}$ as well.

The triangle made up by the edges $e$, $f$, and $g$ is added to the streamsurface mesh. Every vertex $v$ of the new triangle is assigned an additional time attribute $v_t$. For the vertices resulting from corepoints $c_i$ and $d_i$, $c_i t = i$ and $d_i t = i$ holds. The time attribute of the vertex corresponding with corepoint $c_i + 1$ equals $i + 1$. Note that the predicate $A_{i,j}$ holds for both edge $f$ and $g$. Edge $e$ and $g$ are both part of $N_{i,j}(f)$. Furthermore, $I(e, g)$ holds. Finally, $e_0 + e_1 + g_0 + g_1 = 4i + 2 \leq 2(f_0 + f_1) = 4i + 4$ holds. Similarly it can be shown that $A_{i,j}(g)$ is true. Figure 4.9 shows the resulting streamsurface triangle.

**Figure 4.10:** (a) the two grid cells $(u-1, v, w)$ and $(u, v, w)$ before propagating edge $f = (m_1, b_1)$. Note that the highlighted edges are part of the streamsurface boundary $B_{i,l}$; (b) the situation after propagating edge $f$ to $b_2$.

During the propagation of an edge $e$, it is possible that multiple corepoints or space-time corepoints are found sharing the same cell with $e$. In some circumstances, an arbitrary choice from one of the candidate points will result in a malformed streamsurface. Consider the situation sketched in Figure 4.10. We are propagating the streamsurface boundary $B_{i,j}$, and are currently considering edge $f = (m_1, b_1)$, lying on the face shared by two cells $(u - 1, v, w)$ and $(u, v, w)$, with $0 < u < U \land 0 \leq v < V \land 0 \leq w < W$. Using Equation 4.7, we have four candidate points edge $f$ may be propagated to, namely $m_0$, $b_0$, $b_2$, and $m_2$. Choosing $m_0$ as the point edge $f$ is propagated is obviously a bad choice, as the resulting triangle $(m_0, m_1, b_0)$ is already part of the streamsurface resulting in a redundant triangle. Furthermore, edge $f$ will be removed from $B_{i,l}$ maybe causing the streamsurface to stop from being propagated any further. Similarly, $b_0$ is not a good choice either.
This situation can be resolved by removing \( m_0 \) and \( b_0 \) from the candidate point list by observing that both \( m_0 \) and \( b_0 \) do not lie ‘in front’ of the streamsurface viewed from edge \( f \). To determine whether a point lies in front of a streamsurface with respect to a certain edge \( e \) we rewrite function \( A \) to a function \( A : \mathbb{E} \times \mathbb{R}^3 \times \mathbb{R}^3 \). For a given point \( p \), \( A(e, n, p) \) determines whether \( p \) is in front of an edge \( e \) part of a triangle with normal \( n \). The function \( A \) is given by

\[
A(e, n, p) = ((e_1 - e_0) \times n) \cdot (p - e_1) < 0. \tag{4.9}
\]

Using Equation 4.9, we can now rewrite \( G \) given by Equation 4.7 to narrow down the number of candidate points for a streamsurface boundary edge \( e \), assuming that \( e \) is already part of a streamsurface triangle with normal \( n \) as follows

\[
G'(p, e, n) = G(p, e) \land A(e, n, p). \tag{4.10}
\]

Note that for the initial situation where \( B_{i,j} = \mathcal{E}(\mathcal{L}_{i,j}) \) holds, \( G' \) can not be applied to any one of the edges in the streamsurface boundary, since no edge is part of a streamsurface triangle yet. Therefore, for these edges \( G \) is used to determine candidate propagation points.

Now, using \( A \), we can eliminate both \( m_0 \) and \( b_0 \) from the list of candidate propagation points. Assume that we now choose \( b_2 \) as the point to propagate edge \( f \) to. The resulting situation is depicted in Figure 4.10 (b). Note that the highlighted edges indicate the edges belonging to \( B_{i,j} \). Edge \( f \) has been removed from \( B_{i,j} \), whereas edge \((m_1, b_2)\) has been added to \( B_{i,j} \). Another problem now arises upon propagation of edge \( g = (b_1, b_2) \). Every triangle resulting from the propagation of \( g \) is either invalid or redundant. In fact, it is not necessary to propagate \( g \), and it should have been removed from \( B_{i,j} \) after propagation of edge \( f \). Therefore, we need to revise the streamsurface boundary update rule given by Equation 4.8 in such a way that edge \( g \) will not be added. Assuming that edge \( e' \) is being propagated resulting in two new edges \( f' \) and \( g' \) we can rewrite Equation 4.8 to

\[
B'_{i,j} = B_{i,j} \setminus \{(e') \cup \{f', g'\} \cap B_{i,j}\}) \cup \{(f', g') \setminus B_{i,j}\}. \tag{4.11}
\]

Thus, we update the streamsurface boundary by removing the edge we just propagated and by removing the edges that were already part of the streamsurface boundary. This is the final incremental definition of a streamsurface boundary, given that initially \( B_{i,j} = \mathcal{E}(\mathcal{L}_{i,j}) \) holds.

Returning to the example, using Equation 4.10 we can eliminate both \( b_0 \) and \( m_0 \) from the list of candidate propagation points. Now, two candidate propagation points \( b_2 \) and \( m_2 \) remain. As previously mentioned, we have implemented a heuristic that chooses a candidate corepoint such that the final mesh quality is optimized. A high quality triangle mesh ideally consists of equilateral triangles for which every vertex angle equals \( \frac{\pi}{3} \) rad. The equiangle skewness is a value between 0 and 1 that can be calculated for every triangle \( T \), and that indicates how well \( T \) approaches the ideal equilateral triangle form. For a triangle \( T \) with vertex angles \( \alpha, \beta, \) and \( \gamma \), where \( \alpha \) and \( \gamma \) are equal to the minimum and maximum vertex angle respectively, the equiangle skewness is given by

\[
\max\left(\frac{3\gamma - \pi}{2\pi}, 1 - \frac{3\alpha}{\pi}\right). \tag{4.12}
\]
Thus, for an equilateral triangle, the equiangle skewness equals 0, whereas for a triangle where the maximum vertex angle approaches $\pi$, the equiangle skewness approaches 1. Therefore, when determining the optimal candidate corepoint $o$, we choose $o$ in such a way that the resulting triangle after propagation has a minimal associated equiangle skewness.

![Figure 4.11: The situation sketched in Figure 4.6 re-evaluated using our streamsurface propagation algorithm. We will track the streamsurface patches generated by propagating coreline segment $C$. (a) Initially, we have a grid cell at time step $i$ containing two coreline segments, $C$ and $D$. Both are moving in the same direction, indicated by the arrow. The algorithm first evaluates all candidate space-time points part of the grid cell.; (b) Three candidate space-time corepoints have been detected, and are labeled $s_0$, $s_1$, and $s_2$ respectively; (c) $C$ is propagated to the nearest space-time corepoint $s_1$ resulting in the shaded streamsurface patch; (d) Now, assuming that the space-time corepoint $s_2$ lies closer to the edge $(c_0, s_1)$ than $s_0$, it is still discarded as most likely $s_{2t} < s_{1t}$ holds. The boundary may not be propagated backwards in time, and therefore, edge $(c_0, s_1)$ is propagated to $s_0$. The remaining steps are trivial, and the final result is displayed in sub-figure (e).](image)

The final algorithm corresponding with the PropagateCoreline call from Listing 4.2 is printed in Listing 4.3. The predicate $G'$ is included as function call $G$. The equiangle triangle quality measure is implemented by the function OptimalCandidate. Note that depending on the sampling size of the dataset grid, the resulting streamsurface mesh can be quite coarse and jagged. We resolved this by applying a standard mesh subdivision scheme; Loop’s subdivision [22]. The mesh is smoothed by recursively breaking up each triangle in four new triangles, while at the same time adjusting the position of every vertex based on the location of all adjacent vertices. Note that Loop’s subdivision scheme was slightly altered; the location of all corepoints remain unchanged to prevent the detected corelines from being altered.

We now apply our algorithm to the situation sketched in Figure 4.6 to see that our algorithm results in a correct streamsurface. The steps involved are briefly outlined in Figure 4.11.
/* Initialize the streamsurface boundary from the current coreline \( L \).*
B = Edges( L );

for each edge \( e \) in \( B \) do
  /* Search the space–time points and all other corelines
   * for candidate propagation points. */
  candidates = {};
  for each space–time point do
    if \( G( p ) \) then
      candidates = candidates + \( p \);
    fi
  next space–time point;

  for each coreline \( l \) do
    for each point \( p \) in \( l \) do
      if \( G( p ) \) then
        candidates = candidates + \( p \);
      fi
    next point;
    next coreline;

  /* In case we found a candidate point, create a
   * streamsurface triangle. */
  if size( candidates ) > 0 then
    /* Using the equiangle skewness criteria, determine
     * the optimal propagation point. */
    candidate = OptimalCandidate( candidates );

    /* Create a triangle consisting of the current
     * edge points and the new candidate point. */
    AddTriangle( e[0], e[1], candidate );

    /* Remove the current edge from the boundary. */
    B = B \( - \) {\( e \));

  /* Add the newly created edges to the boundary if
   * they were not already part of it. In case they
   * were, remove them from the boundary. */
  for \( i = 0 \) upto 1 do
    if \( ( e[i], \text{candidate} ) \) in \( B \) then
      B = B \( + \) {(\( e[i], \text{candidate} )};
    else
      B = B \( - \) {(\( e[i], \text{candidate} )};
    fi
  od
  fi
next edge

Listing 4.3: Pseudo code for the coreline propagation method.
Chapter 5

Vortex Visualization

In this chapter a flow visualization primitive, the stream arrow, is introduced. First, we present a formal introduction to streamlines since they play an important role in the construction of a stream arrow. Subsequently, a rationale for the introduction of stream arrows is given. Then, the construction and mathematical foundation behind the stream arrow is discussed. Finally, the stream arrow primitive is discussed in a vortex visualization context.

5.1 Streamlines and pathlines

Given a flow defined by a three-dimensional time-dependent velocity vector field, releasing a particle $A$ at a certain location in the flow results in a path $P$. For a definition of a time-dependent vector field, please refer to Appendix A. In case the flow field changes over time, i.e., the flow is unsteady, the path traveled by $A$ differs depending on whether we trace $A$ through the flow field for a given time step or over a period of time. In the former case, the path $P$ is called a streamline, whereas in the latter case, $P$ is called a pathline or particle trace.

**Definition 4 (Streamline):**
Suppose $V : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ defines a three-dimensional time-dependent flow field. Then, a streamline through $V$ for time step $i$ with startpoint $p_0$ is the path defined by the parametric function $r : \mathbb{R} \to \mathbb{R}^3$, for which

$$\frac{dr}{d\tau} = V(r(\tau), i), \quad r(0) = p_0$$

(5.1)

holds.

Equation 5.1 can be solved numerically by integrating $V(r(\tau), i)$ starting from $p_0$ using an integration method like Runge-Kutta [27]. Our application uses either a second- or fourth-order Runge-Kutta implementation provided by VTK.

Informally, a pathline is the line followed by a particle in the flow over time. A formal definition of a pathline is given in Definition 5. As previously discussed, the difference between a streamline and a pathline is that a pathline is defined over a certain time range, whereas a streamline is defined for a specific time step. The streamlines of a time-dependent or unsteady flow differ for every time step. On the other hand, in case of a time-independent or steady flow, streamlines and pathlines coincide.
Figure 5.1: Illustration of the relation between streamlines and pathlines. The flow depicted is a 2D vortex flow spiraling outward from the center. Note that the center of the vortex is gradually moving along the $x$-direction with respect to the viewer; (a) a streamline at three subsequent time steps, with $P$ a particle moving through the flow implicitly defining the streamline; (b) the pathline that results from tracing $P$ from $t = 0$ to $t = 2$ at $t = 2$.

**Definition 5 (Pathline):**
Suppose $V : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$ defines a three-dimensional time-dependent flow field. Then, a pathline through $V$ is the path defined by the parametric function $p : \mathbb{R} \rightarrow \mathbb{R}^3$, for which

$$\frac{dp}{dt} = V(p(t), t)$$

holds.

In Figure 5.1, the difference between streamlines and pathlines is further illustrated.

### 5.2 Stream arrow rationale

A good visualization tries to maximize the amount of information exposed to the observer pertaining to the subject at hand, without hindering the process of extracting information from the visualization due to, for example, visual clutter. Occlusion and the associated visual clutter is the main problem associated with flow visualization and vortex visualization in particular, ruling out the use of a dense visual representation of the dataset. Thus, we need to strike a balance between minimizing the amount of visualization primitives used, and at the same time maximizing the amount of information exposed.
Most sparse 3D flow visualization techniques use either iconic techniques or line primitives like streamlines or stream ribbons. Our approach is to combine the line primitives discussed in Chapter 2 in one visualization widget, the stream arrow, which can subsequently be applied to construct a sparse 3D flow visualization. This enables us to make use of all the information cues introduced by each line primitive separately, and at the same time adding extra information cues specific for a stream arrow.

5.3 Stream arrow parametrization

A stream arrow is a three-dimensional arrow widget originating from a certain point in the flow. All the parameters that can be used to influence the appearance of a basic two dimensional arrow can be applied to a stream arrow as well. Figure 5.2 shows parameters that can be applied to a two-dimensional arrow to influence its appearance.

The parameters $\alpha$, $\beta$, $\omega$, $l_a$, and $l_b$ form the minimum set of parameters required to properly specify an arrow. The parameter $\omega$ is not depicted in the figure, and is the fraction $\frac{w_b}{w_a}$, thus indicating the width of the arrow base $w_b$ relative to the width of the arrow head $w_a$.

In Figure 5.2, the width of the arrow base $w_b$ and arrow head $w_a$ can be derived from the other parameters. Note that although it might seem more intuitive to let the user directly control the width of the arrow base and the arrow head, this leads to problems in practice. The length of every stream arrow in the flow differs, and therefore, the width of the arrow base and arrow head differ for every stream arrow, since we aim to keep the proportions of every stream arrow the same. Thus, letting the user directly influence the arrow base and head width would require a width setting that somehow depends on the length of every stream arrow based on, again, a certain abstract proportion. We felt that it would be more intuitive to let the user choose the size and width of the arrow head, which in turn determines the width of the arrow base by means of the $\omega$ parameter, as is explained next.

The parameter $l_b$ equals the length of the base of the arrow, whereas $l_a$ equals the length of the arrow head, measured from the end of the arrow base. Thus, the total length of the arrow $L$ is given by $l_a + l_b$. The angle $\alpha$ determines the width of the arrow...
head $w_a$, and is given by

$$w_a = 2 l_a \tan \alpha.$$  \hspace{1cm} (5.3)

Using Equation 5.3 and $w$, we can calculate the width of the arrow base $w_b$ using

$$w_b = \omega w_a.$$ 

The angle $\beta$ influences whether the arrow head hangs over the arrow base. In case $\beta = 90^\circ$, there is no overhang. The amount of overhang $o$ is given by

$$o = (w_a - w_b) \cos \beta.$$ 

### 5.4 Stream arrow construction

For the discussion that follows we assume a steady flow field $\mathcal{V} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is given. We are visualizing vortices that occur in $\mathcal{V}$ and were previously detected using the vortex detection algorithm outlined in Chapter 3.

A stream arrow can be placed anywhere in the flow $\mathcal{V}$. Assume we pick a certain starting point $p$ for a stream arrow $S$, where $p$ is a point in the flow with $\mathcal{V}(p) \neq 0$. Note that the latter requirement is not strict, but is added to ensure that the stream arrow $S$ has a length greater than 0. Then, the path of $S$ is determined by the streamline through $p$, given by $r(\tau)$. An example stream arrow with the associated streamline is depicted in figure 5.4.

The length of a stream arrow is calculated using a function $L : \mathbb{R}^3 \times (\mathbb{R}^3 \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$, where $L(p, \mathcal{V})$ returns the length of the stream arrow originating from $p$ for the flow specified by the steady vector field $\mathcal{V}$. Using Definition 4 and the time-independent version of Equation 5.1, we can define $L$ as

$$L(p, \mathcal{V}) = \int_0^T \left| \frac{dr}{d\tau} \right| d\tau = \int_0^T |\mathcal{V}(r(\tau))| \, d\tau.$$  \hspace{1cm} (5.4)
where $T$ indicates the amount of time an imaginary particle is traced starting from $p$ along the streamline in order to determine the end point of the stream arrow. By varying $T$, we can influence the average length of all stream arrows in the flow. Increasing $T$ will result in longer stream arrows overall, whereas decreasing $T$ will result in a shorter overall stream arrow length. Note that this can be implemented in two different ways. One approach is to set a maximum threshold for $T$, and subsequently integrate the streamline starting from $p$ while calculating the total integration time elapsed since the tracing started. As a result, both the start and the end point of the stream arrow move according to the flow, whereas the length of the stream arrow varies.

Another method is to set a maximum threshold for $L$, and then in a similar way integrate the streamline while calculating the sum of the velocity magnitude for all discrete streamline segments. Both methods are in essence equivalent. Assuming the velocity $v$ along a streamline is constant, setting a threshold $T$ for the maximum amount of time an imaginary particle is traced along the streamline would be equal to setting a maximum threshold $v \cdot T = d$ for the stream arrow length. Our implementation utilizes the first approach as the points belonging to the streamlines resulting from the default \texttt{vtkStreamLine} class already contain a scalar attribute that equals the amount of time elapsed since tracing of the streamline began. This enabled a more efficient implementation of the first approach than of the second approach, which requires separate calculations of streamline segments to determine the length of the stream arrow.
Figure 5.5: Vertex normal calculation for vertices making up the arrow head.

The stream arrow itself is constructed by connecting 2D slices placed along the underlying streamline over the length of the stream arrow with each other, as depicted in Figure 5.4. Connecting two slices, both consisting of $N$ points, results in a triangle strip consisting of $2N$ triangles. Every point $s_i$ of a slice $S$ consisting of $N$ points, with $0 \leq i < N$, has an associated vertex normal $n_i$. Assume slice $S$ is centered at point $c$ and is part of the stream arrow base. Then, in case of a circular slice, $n_i$ is given by

$$n_i = \frac{s_i - c}{|s_i - c|}.$$  

(5.5)

Figure 5.6: Base slice mesh, with the black normal vectors representing the vertex normals of slice $A$ equal to the vector $-\frac{\mathbf{v}(c_0)}{||\mathbf{v}(c_0)||}$, whereas the gray vectors are the vertex normals associated with the points from slice $B$, obtained using Equation 5.5.

In case slice $S$ is part of the head of the stream arrow, using the angle $\alpha$ from Section 5.3, $n_i$ can be derived from Figure 5.5, and is given by

$$n_i = \frac{s_i - c}{|s_i - c|} \cos(\alpha) + \frac{\mathbf{v}(c)}{|\mathbf{v}(c)|} \sin(\alpha).$$  

(5.6)

The first slice $S_0$ requires special attention. First of all, it needs to be capped. This is done by adding an additional point to the slice coinciding with the center point of the slice, $c_0$. Then, a triangle fan is generated by creating $N$ triangles made up of the points $c_0$, $s_i$, and $s_{(i+1) \mod N}$, with $0 \leq i < N$. Furthermore, to obtain a sharp boundary between the triangles belonging to the capped slice $S_0$ and the triangles forming the connection between slice $S_0$ and $S_1$, two vertex normals are required for every point.
belonging to slice $S_0$. In practice, VTK only allows for one vertex normal to be associated with every vertex. Therefore, slice $S_0$ is duplicated to form two identical slices $A$ and $B$, both centered at $c_0$. All points $a_i$ from slice $A$ have vertex normal $-V(c_0)$, whereas Equation 5.5 is used to calculate the vertex normals for the points making up slice $B$. Now, the points from slice $A$ are used to construct the capped base of the stream arrow, whereas the points from slice $B$ are used to form the connection with slice $S_1$. The resulting situation is illustrated in Figure 5.6.

A similar situation occurs when connecting the final slice $S_j$ making up the base of the stream arrow with the first slice of the stream arrow head $S_{j+1}$. Both slices need to be duplicated since both slices are located at a sharp boundary, and therefore require two vertex normals per point. Assume that slice $S_j$ is duplicated resulting in two slices $D_0$ and $D_1$. Furthermore, assume that slice $S_{j-1}$ is connected with slice $D_0$. Then the vertex normals for the points belonging to slice $D_0$ can be calculated using Equation 5.5. Next, assume that slice $S_{j+1}$ is duplicated resulting in two slices $D_2$ and $D_3$, where slice $D_1$ is connected with $D_2$, slice $D_2$ is connected with slice $D_3$, and finally, slice $D_3$ is connected with slice $S_{j+2}$. The vertex normals for slice $D_3$ can be calculated using Equation 5.6. Using the angle $\beta$ defined in Section 5.3, the vertex normals $n_i$ associated with every point $s_i$ of the slices $D_1$ and $D_2$ are given by

$$n_i = -\frac{V(c)}{|V(c)|} \sin \beta - \frac{c - s_i}{|c - s_i|} \cos \beta$$

with $c$ the center of the slice under consideration. Figure 5.7 shows a 2D cross-section of a stream arrow, illustrating the different vertex normals associated with the vertices of the slices part of the connection between the arrow base and the arrow head.

![Figure 5.7](image-url)
coordinate frame $F_i$ is defined by three vectors $u_i$, $v_i$, and $w_i$. For coordinate frame $F_0$ we use a Frenet frame [10], with

\[
\begin{align*}
  u_0 &= \frac{V(p_0)}{|V(p_0)|} \\
v_0 &= \frac{V(p_0) \times (\nabla V(p_0))V(p_0)}{|V(p_0) \times (\nabla V(p_0))V(p_0)|} \\
w_0 &= u_0 \times v_0.
\end{align*}
\]

Here, $(\nabla V(p_0))V(p_0)$ represents the acceleration at $p$. Bloomenthal [10] gives a derivation for $v_0$.

![Figure 5.8: Mesh skew due to separately calculated local coordinate frames.](image)

To obtain a proper mesh, calculating the Frenet frame for all subsequent slices does not suffice. Due to vorticity or torsion, rotation is observed about the $u$ vector of $F_i$ with respect to $F_{i+1}$. This results in the final mesh being skewed as illustrated in Figure 5.8. Excessive skew results in polygons becoming long and thin, while their centers approach the streamline.

To prevent mesh skew, for all slices $S_i$ located at $p_i$, with $i > 0$, we project the coordinate vector $v_{i-1}$ belonging to frame $F_{i-1}$ on the plane with normal $V(p_i)$ centered at $p_i$. Projecting $v_{i-1}$ results in a new frame vector $v_i$. Together with $u_i = V(p_i)$ we can calculate $w_i$. This results in a new coordinate frame $F_i$ that is not rotated around the streamline’s tangent vector with respect to coordinate frame $F_{i-1}$. However, the amount of vorticity or torsion about the velocity axis might be an interesting information cue for the observer. Therefore, in Section 5.5.1 we describe a method to convey the vorticity information in the final visualization.

The amount of 2D slices used to construct the stream arrow can be varied based on an abstract quality parameter $q$. Obviously, by increasing the amount of slices the resulting stream arrow becomes smoother, thus enhancing the quality of the final stream arrow visualization. Additionally, for slice types where the number of points per slice influences the appearance of the final stream arrow, larger values for $q$ result in a larger number of points used per slice.
5.5 Slice parametrization

An additional stream arrow parameter is introduced by allowing the shape of the 2D slices to be varied. The example stream arrow from section 5.4 consists of circular 2D slices. Two additional slice shapes are supported by our visualization tool; lobed circular slices and rectangular slices. The parameters for each slice shape as well as the unique visualization options introduced by each slice shape are discussed next. The circular slice has been discussed in the previous section.

5.5.1 Lobed circular slice

The lobed circular slice is based on the circular slice. An example slice is shown in Figure 5.9. A lobed circular slice has several parameters:

- \( L \), the number of lobes
- \( N_l \), the number of points per lobe
- \( d \), the lobe depth

The lobe depth \( d \) is a fractional value, with \( 0 \leq d \leq 1 \). Suppose \( r \) is the maximum distance of a point belonging to the slice relative to the slice center point \( c \). Thus, for a slice that is part of the base of the stream arrow, \( r = w_b \) holds, whereas of slice that is part of the stream arrow head, \( 0 \leq r \leq w_a \) holds. Then, \((1 - d)r\) equals the minimum distance of any point belonging to the slice to relative to the slice center point \( c \), whereas \( d \cdot r \) equals the depth of an individual lobe.

![Figure 5.9: An example lobed circular slice. The following values were used for the lobed circle slice parameters to generate the above slice; \( N_l = 5 \), \( L = 9 \), and \( d = \frac{1}{4} \).](image)

The lobes are standard sinusoid half period graphs. Assume we are calculating the location of the points belonging to a lobed circle slice \( S_i \), with maximum radius \( r \) and associated coordinate frame \( F_i \), as discussed in the previous section. Then, the location of a point on the \( k \)-th lobe \( k \) is given by the function \( p : \mathbb{R} \rightarrow \mathbb{R}^3 \), defined by

\[
p(t) = (r - d + d \sin t \pi) (v_i \cos \gamma + w_i \sin \gamma) + c
\]

(5.7)
where \( \gamma \) is equal to the angle offset from the first point of the slice, and is given by

\[
\gamma = (k + t) \frac{2\pi}{L}.
\]

Then, the location of the \( j \)-th point on the \( k \)-th lobe is given by \( p \left( \frac{j}{N_l} \right) \). Figure 5.9 also shows the vertex normals associated with the points of one lobe. Using Equation 5.7, the vertex normal of a point on the \( k \)-th lobe is given by the function \( n : \mathbb{R} \rightarrow \mathbb{R}^3 \), defined by

\[
n(t) = \left| \frac{dp}{dt} \cdot w \right| v_i - \left| \frac{dp}{dt} \cdot v \right| w_i.
\]

The total number of points per slice equals \( L \cdot N_l \).

In Section 5.4 we discussed the correction of the coordinate frame \( \mathcal{F}_i \) with respect to the previous coordinate frame \( \mathcal{F}_{i-1} \) to prevent mesh skew introduced by the vorticity at \( c_i \).

![Figure 5.10: (a) lobed circular stream arrow, along with the corresponding mesh (b).](image)

After the new coordinate frame for slice \( S_i \) has been properly calculated, we now allow the torsion or vorticity around the velocity vector at \( c_i \) to introduce a slight rotation of the coordinate frame with respect to the previous coordinate frame. This introduces an extra information cue to the final visualization, as the grooves in the stream arrow that result from the lobes clearly indicate the amount of torsion about the center line of the stream arrow. We use the helicity at \( c_i \) as a measure for the amount of relative rotation of the coordinate frames. The helicity is a scalar value calculated by taking the dot product of the vorticity and the velocity vector at a certain point \( p \). The helicity is a measure of the rotation of the velocity field around the velocity vector at \( p \).

Since the coordinate frame rotation introduces mesh skew, larger values for the helicity require the number of slices used to construct the stream arrow to be increased to prevent the mesh skew from being visible in the final visualization. Therefore, the abstract quality setting \( q \) that controls the number of slices in a stream arrow linearly depends on the average helicity at each point \( c_i \). Additionally, \( N_l \) linearly depends on \( q \). Thus, a larger quality setting results in smoother lobed stream arrows as the number of points per lobe increases. In Figure 5.10, a lobed circular stream arrow along with
the corresponding mesh is depicted. Additionally, at the end of this chapter, Figure 5.16 shows some stream arrows built using lobed circle slices.

![Diagram](image)

Figure 5.11: Rectangular slice, centered at point \( c \) with height \( h \) and width \( w \). Every vertex has two associated vertex normals.

### 5.5.2 Rectangular slice

The third slice variation, the rectangular slice, is depicted in Figure 5.11. One parameter \( h \) is associated with a rectangular slice, controlling the height of the slice. The height is fixed for every slice, resulting in an overall flat stream arrow. In practice, every slice variation is implemented in a separate class with a virtual `ScaleForArrowHead` method, allowing custom scaling to be performed for the slices which are part of the stream arrow head. In the case of (lobed) circular slices, the slices are uniformly scaled, whereas in the case of rectangular slices, scaling is only performed along the \( v \) axis.

Another virtual method `CreateBaseHeadConnection` provides custom mesh generation code for the connection of the first arrow head slice with the last base slice and the second arrow head slice. No custom mesh generation code is required to connect these slices for both the circular and lobed circular stream arrow variations, whereas for a rectangular slice simply connecting the slices as previously described does not suffice, as that would result in cracks showing up in the final mesh. Figure 5.12 gives an indication of the mesh layout for a rectangular stream arrow in the neighborhood of the arrow base and head connection.

![Stream Arrow](image)

(a) ![Mesh](image)

(b)

Figure 5.12: Rectangular stream arrow (a) and a close-up of the corresponding mesh near the connection of the last base and first arrow head slice (b).
To achieve sharp edge boundaries at the four corners of a rectangular stream arrow, two vertex normals are required. Therefore, every vertex has to be split, resulting in a total of 8 vertices per slice. Connecting \( n \) rectangular slices using triangle strips results in a total of \( 8(n - 1) \) triangles.

The width \( w \) equals \( w_b \) for a slice that is part of the base of the stream arrow. On the other hand, \( w = w_a \) holds for a slice that is part of the stream arrow head. Obviously, the quality parameter \( q \) has no effect on any of the parameters of the rectangular slice.

### 5.6 Vortex visualization using stream arrows

To be able to properly visualize a vortex using our stream arrow visualization primitive, we have to ensure that the streamlines used to construct the stream arrows are part of the vortical structure. Such a streamline that propagates through a vortical structure will be called a vortex streamline henceforth. Because of the corelines that resulted from the vortex detection step, we already have a good estimate of candidate streamline start points that will result in vortex streamlines. Informally, one can say that any streamline seed point that lies sufficiently close to a vortex coreline is a candidate seed point for a vortex streamline. We formalize the notion of 'sufficiently close' by defining a region of interest (ROI) around the corelines consisting of all the points that are candidate seed points for vortex streamlines. Bauer & Peikert [9] define the ROI as all the points that result from radially extending the coreline as long as the vortex strength is above a given threshold. We gave a definition for the vortex strength in Chapter 3; the vortex strength equals the absolute value of the imaginary part of the complex values of the velocity gradient tensor. Other definitions for the ROI or vortex hull exist. For example, Banks & Singer [6] define the vortex hull as the region of points around the coreline for which the pressure is below a given threshold and the associated vorticity vector does not make an angle larger than \( 90^\circ \) with the vorticity vector at the coreline. Alternatively, Garth & Tricoche [16] define the vortex hull by radially extending the coreline and calculating for each point \( p \), extended from a point \( c \) on the coreline, the tangential velocity. They define the tangential velocity as the projection of the velocity at \( p \) on the line perpendicular to the position vector of \( p \), given by \( p - c \). Then, every point \( p \) is considered part of the vortex hull until a maximum for the tangential velocity has been found along the position vector \( p - c \). Note that their definition of the vortex hull is firmly based on the Rankine vortex model as discussed in Chapter 3.

![Figure 5.13](image_url)

**Figure 5.13**: 2D depiction of a vortex with coreline \( C \), where the boundaries of the vortex hull are illustrated by the dashed lines. Here, the streamline \( \mathcal{L} \) is traced in both directions starting from point \( s \).

To obtain a proper visualization of a vortex \( v \), a vortex streamline \( \mathcal{L} \) that is traced
from a candidate seed point \( s \) is limited to the direct vicinity of \( v \) to ensure that every stream arrow constructed along \( \mathcal{L} \) is part of the flow around \( v \) instead of the global flow. Assume vortex \( v \) has an associated coreline \( \mathcal{C} \) consisting of \( N \) corepoints \( c_i \), with \( 0 < i < N \), where \( c_{i-1} \) is connected to \( c_i \) to form \( \mathcal{C} \). Similarly, assume \( \mathcal{L} \) is given by the parametric function \( \mathbf{r}(t) \), with \( 0 \leq t < 1 \), as defined in Section 5.1. In practice, \( \mathcal{L} \) is given by a discrete set of \( M \) points \( l_j \), with \( 0 \leq j < M \), where \( l_{j-1} \) is connected to \( l_j \) to form \( \mathcal{L} \). The function \( \mathcal{D}(\mathbf{l}, \mathbf{q}) \), with \( \mathcal{D} : \mathbb{R} \times \mathbb{R} \to \mathbb{B} \), indicates for a point \( \mathbf{l} \) along the streamline \( \mathcal{L} \) whether it is the closest point to \( \mathbf{q} \) on the streamline \( \mathcal{L} \), and is given by

\[
\mathcal{D}(\mathbf{p}, \mathbf{q}) = (\forall i \in \mathbb{N} : 0 \leq i < M : |\mathbf{p} - \mathbf{q}| \leq |\mathbf{l}_i - \mathbf{q}|).
\]

Then, the bounded streamline \( \mathcal{L}_b \) consists of the points \( l_a \) up to and including point \( l_b \), with \( 0 \leq a < b < M \), for which both \( \mathcal{D}(l_0, c_0) \) and \( \mathcal{D}(l_{b-1}, c_{N-1}) \) hold. Thus, the two points on the streamline closest to the first point and the last point of the coreline respectively define the boundary points of the bounded streamline, clearly limiting the streamline to the detected vortex feature. Another approach is to bound the streamline by the vortex hull, by determining for every point on the streamline whether it is part of the vortex volume. However, in practice this leads to problems where the bounded streamline extends beyond the limits of the associated vortex, since there is no guarantee that the vortex hull condition does not hold for points not radially extending from the coreline.

Now, the question what point within the vortex hull to choose as a seed point for a vortex streamline remains. Using a point part within the vortex hull as a seed point for a streamline does not guarantee that the resulting streamline will spiral around the full extent of the associated coreline. As illustrated in Figure 5.13, it is possible for a bounded streamline seeded from a point part of the vortex hull to leave the vortex hull. Any stream arrow constructed along such a streamline does not convey information about the vortex flow exclusively, and therefore these streamlines are not suited as a basis for the construction of stream arrows aimed at vortex visualization.

To guarantee that the streamline traced from a candidate seed point \( s \) is a vortex streamline for a certain vortex \( v \), for every point \( \mathbf{p} \) on the streamline, \( \mathcal{H}_v(\mathbf{p}) \) needs to hold, where \( \mathcal{H}_v \) is a Boolean function that indicates whether a given point is part of the vortex hull of \( v \). Observe that given a discrete bounded streamline \( \mathcal{L}_b \) consisting of \( M \) points \( l_i \), with \( 0 \leq i < M \), it is reasonable to assume that in case both \( \mathcal{H}_v(l_0) \) and \( \mathcal{H}_v(l_{M-1}) \) hold, \( \mathcal{H}_v \) holds for every point \( l_i \).

As discussed in Chapter 3, in general two classes of vortices can be distinguished; attracting and repelling vortices. In Figure 5.14, a vortex streamline is depicted for both types of vortices. For an attracting vortex, to guarantee that at least \( \mathcal{H}_v(l_0) \) holds, for an associated coreline \( \mathcal{C} \) consisting of \( N \) points \( c_i \), with \( 0 \leq i < N \), we choose the streamline seed point \( s \) on the plane \( \mathcal{P} \) located at \( c_0 \) with normal \( \mathbf{V}(c_0) \). Later on in this chapter, we give a parametrization for the exact location of \( s \) on \( \mathcal{P} \). Here, \( \mathbf{V} \) is a function defining the velocity field of the flow. Furthermore, the ordering of the corepoints is chosen in such a way that for every subsequent pair of corepoints \( c_j \) and \( c_{j+1} \), with \( 0 \leq j < N - 1 \),

\[
(c_{j+1} - c_j) \cdot \mathbf{V}(c_j) > 0
\]

holds. Thus, the corepoints are ordered along the direction of the velocity along the
coreline. Obviously, we require that $\mathcal{H}_v(s)$ needs to hold. Since $s$ was chosen very near to $c_0$, for the resulting bounded streamline $L_0$, $\mathcal{H}_v(l_0)$ holds as well. Given that the discretized bounded streamline consists of $M$ points, we now only need to check whether $\mathcal{H}_v(l_{M-1})$ holds as well. As can be derived from Figure 5.14, choosing a streamline seed point near the vortex hull boundary will result in a streamline for which $\mathcal{H}_v(l_{M-1})$ holds as well, since the streamline gradually approaches the vortex coreline as it is traced through the flow.

Again, by looking at Figure 5.14, we can derive that for a repelling vortex, choosing the streamline seed point $s$ near the vortex hull boundary in the plane with the origin at $c_{N-1}$ and normal $\mathcal{V}(c_{N-1})$ will result in a bounded streamline $L_b$ for which both $\mathcal{H}_v(l_0)$ and $\mathcal{H}_v(l_{M-1})$ hold.

Figure 5.14: Two 2D depictions of a vortex with coreline $C$, where the boundaries of the vortex hull are illustrated by the dashed lines. Here, the streamline $L$ depicts a vortex streamline for an (a) attracting vortex, and a (b) repelling vortex. The point $s$ indicates the streamline seed point used to generate the bounded vortex streamlines.

Figure 5.15: Different streamline seed point layout examples located in a plane centered at $c_0$. The following parameters were used to generate the layouts, (a) $N = 3, r = 0.5, \alpha = 0^\circ, \delta = 0.2$; (b) $N = 10, r = 0.75, \alpha = 36^\circ, \delta = 0$; (c) $N = 15, r = 0.35, \alpha = 35^\circ, \delta = 0.045$. 
Multiple seed points can be associated with a vortex to generate multiple vortex streamlines which in turn used for stream arrow generation. The layout of the seed points in the previously discussed 2D plane is parametrized. Suppose we are generating vortex streamlines for an attracting vortex with coreline \( C \) consisting of \( N \) points \( c_i \), with \( 0 \leq i < N \). Prior to seed point generation, we first determine the maximum seed point distance from the coreline by calculating the vortex hull slice for corepoint \( c_0 \) as described by Bauer & Peikert [9]. Assume \( R \) equals the average vortex hull boundary distance as measure from \( c_0 \). Now, the parameters associated with the streamline seed point layout are:

- \( N_s \), the number of seed points
- \( r \), a value between 0 and 1 indicating the minimum seed point distance from \( c_0 \) as a fraction of \( R \)
- \( \alpha \), the angular distance between subsequent seed points
- \( \delta \), a value between 0 and 1 equaling the radial distance between subsequent seed points as a fraction of \( R \)

Assuming we have calculated a Frenet frame \( F_0 \) located at \( c_0 \) as previously discussed in this chapter. Then, the location of the \( i \)-th seed point, with \( 0 \leq i < N_s \), is given by

\[
(r + i\delta)R(\cos(i\alpha)v_0 + \sin(i\alpha)w_0).
\]

Several example layouts are depicted in Figure 5.15 along with the corresponding parameter values. Note that it is possible that for a seed point \( s' \), the distance \( s' \) to the corepoint \( c_0 \) is larger than \( R \). This is no problem provided that \( \mathcal{H}_v(s') \) holds. For any seed point \( s \) for which \( \mathcal{H}_v(s) \) does not hold, no streamline is traced, and \( s \) is simply ignored.

After the streamlines have been traced and bounded, the stream arrows are generated as discussed in Section 5.4. Assume we are constructing stream arrows along a streamline \( \mathcal{L} \), consisting of \( M \) points \( l_i \), with \( 0 \leq i < M \). The first stream arrow is constructed starting from point \( l_0 \). Every point \( l_i \) along the streamline has an associated scalar time attribute \( t_i \) indicating the time elapsed between the addition of point \( l_i \) to the streamline and the start of the streamline tracing. Thus, \( t_0 = 0 \) holds for \( l_0 \). Initially, the slices of the first stream arrow are centered around the points \( l_0 \) up to and including the center point of the last slice \( e \), given by

\[
e = (1 - \lambda)l_p + \lambda l_{p+1},
\]

with \( 0 < p < M - 1 \). Thus, in case \( \lambda \neq 0 \) holds, the center point of the last stream arrow slice does not coincide with a streamline point, but lies on the streamline segment defined by the points \( l_p \) and \( l_{p+1} \). Then, assuming we use a time span threshold \( T \) to determine the length of a stream arrow as explained in Section 5.4,

\[
T = t_p + \lambda(t_{p+1} - t_p) - t_0
\]

holds. Note that it is possible for the end of the bounded streamline to be reached before 5.8 holds. In that case,
\[ T > t_{M-1} - t_0 \]

holds. Then, the last slice of the stream arrow is capped as is also done for the first stream arrow slice as previously illustrated in Figure 5.6. This in essence clips the stream arrow primitive. An example clipped stream arrow is shown in at the end of this chapter in Figure 5.16.

After having constructed the first stream arrow and provided that the end point of the stream arrow did not coincide with the last point along the stream line \( l_{M-1} \), a second stream arrow is constructed. The distance between two stream arrows is parametrized, and is determined by a parameter \( \tau \). Assuming that the last point of the last constructed stream arrow equals \( e \) as introduced in Equation 5.6, the start point of the next stream arrow is \( b \) given by

\[
b = (1 - \rho)l_q + \rho l_{q+1},
\]

with \( 0 < p < q < M - 1 \), such that

\[
\tau \cdot T = t_q + \rho(t_{q+1} - t_q) - (t_{p+1} - (1 - \lambda)(t_{p+1} - t_p))
\]

holds. In other words, the distance between two subsequent stream arrows is equal to the distance traveled by a point traced along the streamline starting at \( e \) for \( \tau \cdot T \) seconds.

After having constructed the stream arrows, it is possible to animate the stream arrows along the streamline. For every time step in the animation, the start point of a stream arrow \( S \) with start point \( b \) is advanced by tracing a point along the streamline starting from \( b \) for \( \phi \cdot T \) seconds. Using the newly calculated start point \( b' \), a new stream arrow is constructed. In case tracing along the streamline from \( b \) reaches the last point of the streamline \( l_{M-1} \) within \( \phi \cdot T \) seconds no new stream arrow is created, since it just left the vortex context defined by the bounded streamline extent. A new stream arrow flows in at the beginning of the bounded streamline in case a particle traced from the beginning of the stream arrow would need \( \tau \cdot T \) seconds to reach the first point of the first stream arrow. Note that in case a new stream arrow enters the vortex context, no complete stream arrow can be constructed. However, to construct a proper partial stream arrow we use the data associated with the unbounded streamline to determine the length of a stream arrow just entering the bounded streamline context. Figure 5.16 shows various stream arrow variants around a bounded streamline with stream arrows entering and leaving the bounded streamline context.
Figure 5.16: Different stream arrows variants circling around a coreline, with each variation made up of (a) circular, (b) rectangular, and (c) lobed circular slices respectively. The coreline colormap depicts the vortex strength, while both the ground plane and the stream arrow colormaps show the local velocity magnitude. Note that for the lobed circular stream arrows, the first stream arrow just enters the bounded streamline, and is clipped.
Chapter 6

VortexView

In this chapter the vortex visualization tool (VortexView) that was implemented to experiment with the stream arrow visualization primitive is discussed. A software manual along with two different use cases giving an overview of the features provided by the visualization toolkit is followed by a short discussion of the implementation details.

6.1 VortexView use case scenarios

Our tool is designed to gain insight in the configuration of vortical structures within a flow. Given a flow dataset, optionally consisting of multiple time steps, our tool is able to detect the vortical structures and track them over time. Currently, three different vortex detection techniques have been implemented using the parallel vector operator:

- Banks & Singer’s predictor-corrector method [5]
- Sujudi & Haimes’ eigenvector method [40]
- Roth & Peikert’s method [34]

VortexView introduces the concept of an experiment, containing an arbitrary amount of datasets. Initially, a default experiment is created containing no datasets. The user can then add datasets to the experiment, or load a previously saved experiment. Typically, every dataset corresponds with one time step in the experiment.

6.1.1 A single timestep use-case scenario

Suppose you have a single flow dataset, and you are interested whether the dataset contains any vortices, and if so, how these vortices look like. First, the original dataset needs to be converted to a standard VTK dataset. VortexView supports uniform, rectilinear, and structured grids. From the VTK website (http://www.vtk.org) a document containing an overview and exact description of all supported file formats can be downloaded, so it should be relatively easy to write a converter for any particular data format. Then, a new experiment has to be created by selecting New experiment from the File menu. In case a modified experiment was already open, VortexView asks whether to save any modifications to that experiment, after which a new experiment is created. The flow dataset can now be added to the experiment by using the ‘+’ button in the
Experiment tab. The Experiment tab gives an overview of all the datasets in one experiment, and their relative ordering. VortexView assumes that the first dataset in the list of datasets corresponds with the results from timestep 0 in the experiment, whereas the last dataset in the list corresponds with timestep $N - 1$, assuming the experiment consists of $N$ timesteps. It is possible to add multiple datasets at once.

![VortexView interface](image)

**Figure 6.1:** Cutting plane depicting local velocity magnitude.

After loading the first dataset, a dataset boundary is displayed in the 3D view. Optionally, this boundary can be turned off using the Show dataset boundary checkbox on the flow visualization tab. At this stage, basic visualization primitives can be used to quickly gain insight in the general flow of the dataset. For example, a cutting plane can be selected to display the velocity magnitudes along a certain plane through the dataset, as displayed in Figure 6.1. The colormap used for the cutting plane can be edited by right clicking the cutting plane model, and selecting the option Edit colormap. A color map editor pops up allowing the user to edit the scalar ranges that map to certain colors, the actual colors used, or the kind of scalar value displayed by the color map. For example, in case the original VTK dataset also contains pressure information, the user has the option to show pressure information instead. The cutting plane itself can be interactively placed by clicking it and subsequently dragging it to change its position, or manipulate the normal to change its orientation.

Alternatively, a stream trace can placed at an arbitrary point in the flow. A stream trace is a bundle of stream lines with their seed points located at random locations in a bounded region around the location of the stream trace.

To start the vortex detection process, the user has to maneuver to the Detection panel, and choose one of the three available detection methods. Note that in case no pressure information is available, Banks & Singer’s method is disabled since it requires the pressure information to be available. To enhance the detection results, it is possible to manipulate the starting point grid as previously discussed in Chapter 3 by clicking the Edit Newton-Raphson start point grid button. A popup window will appear
that enables the user to interactively choose a starting point grid layout. The detection process is started after a click on the Detect coreline button. The vortex detection is started in a separate thread, so it is possible for the user to explore the dataset further, or add extra datasets to the experiment, while vortex detection is in progress.

After the detection process finishes, the list of Detected vortices on the Detection tab is filled with entries for every vortex, sorted by vortex length. All vortices meeting the current visualization criteria are displayed as corelines in the 3D view. By default, every coreline has an associated colormap displaying the vortex strength. Now, using the Vortex strength threshold spin control, it is possible to influence the vortices that are displayed in the final visualization. To prevent any weak vortex from being

Listing 6.1: Parts of the XML code representing the flow experiment depicted in Figure 6.2.

```xml
<experiment>
  <name>
    Red cube experiment
  </name>
  <cube>
    <coordinates>
      0.000355 0.000485 0
      0.00014 0.000415 0.000545
    </coordinates>
    <color>
      1 0 0
    </color>
  </cube>
  ...
  <detectionresult>
    <detectionmethod>
      Roth & Peikert
    </detectionmethod>
    <timestep>
      <vortex enabled="true">
        <name>
          Small, weak vortex
        </name>
        <coreline numberofCorepoints="6">
          <radius>
            0.1
          </radius>
          <corepointsVisible>
            0
          </corepointsVisible>
        </coreline>
      </vortex>
    </timestep>
  </detectionresult>
</experiment>
```
visible in the final visualization, increasing the vortex strength threshold ensures that any vortex with a vortex strength lower than the active threshold value will not appear in the final visualization. Alternatively, small vortices can be hidden by increasing the minimum amount of corepoints a vortex should consist of by adjusting the Minimum points per coreline spin control. It is still possible to show or hide individual vortices from the 3D view or the vortex list. For example, to hide an unwanted vortex from the 3D view, right click it and select Hide. Alternatively, click on any listed vortex in the vortex list. Then, the corresponding vortex in the 3D view will start to blink. The selected vortex can then be hidden by clicking the 'eye' checkbox in the vortex list. We now assume that the user is presented with the resulting corelines as depicted in Figure 6.2 after the vortex detection process has finished. In Figure 6.2, the red cube represents an obstacle present in the flow during the experiment. It is possible to augment the experiment data with basic 3D primitives, as is explained later in this chapter.

![Figure 6.2: Corelines resulting after a vortex detection process.](image)

Now, after having eliminated all weak and small vortices from the collection of detected vortices, the user can augment the coreline visualization with stream arrows illustrating the flow around the corelines. To do so, the stream arrows are enabled by checking the Show stream arrows checkbox from the Stream arrows tab. Now, for every visible coreline, stream arrows are displayed along a streamline in the direct vicinity of every coreline. The seed point of every streamline is chosen as described in Chapter 5. The stream arrow parameters can be edited by clicking the Edit stream arrow parameters button from the Stream arrows tab. This brings up the stream arrow editor as depicted in Figure 6.3. The user can influence the global representation of every stream arrow, the number of stream arrows per vortex, and so forth. For example, the relative distance between every stream arrow as measured in a percentage of the total streamline length can be adjusted, such that the total number of stream arrows displayed along a streamline is adapted. Furthermore, the extent of the streamline along which the stream ar-
Figure 6.3: The stream arrow editor.

rows are placed can be changed. This allows the user to get a better insight in the way the flow behaves around the detected vortices. The stream arrows can be animated to get a feeling for the relative flow velocity around every coreline.

It is also possible to construct a collection of stream arrows for a single vortex by right-clicking on the coreline under consideration, and then selecting Edit stream arrows... from the popup menu. In our use case scenario, one of the detected vortices is singled out for closer inspection, and a collection of stream arrows is constructed to investigate the local vortex flow. The streamline that is used to construct the stream arrows is extended beyond the context of the vortex. The resulting visualization is displayed in Figure 6.4(a). The color map for the stream arrows depicts the velocity magnitude along the stream arrow, where the minimum velocity magnitude maps to blue, and the maximum velocity magnitude to red. The ground plane's color map also has a velocity magnitude color map. For that colormap, the minimum velocity magnitude maps to white and the maximum velocity magnitude maps to orange, to ensure that the stream arrows stand out from in the visualization and do not blend in with the background. The color map for the corelines depicts the vortex strength. A negative vortex strength is associated with a counter-clockwise rotating vortex, and is mapped to blue area of the colormap, whereas red is associated with the maximum positive vortex strength.

From Figure 6.4(a) it becomes apparent that the flow comes from behind the obstacle in the flow, and then enters the selected vortex. The direction of the flow in the vortex is opposite to the overall direction of the flow, as it moves back in the direction of the obstacle. Finally, after leaving the vortex context, the flow resumes along the original direction away from the obstacle in the flow. As explained in Chapter 5, the length of a stream arrow is determined by the velocity along the underlying streamline. This can be seen in Figure 6.4(a) as well; the red colored stream arrows are larger than the more
Figure 6.4: Stream arrows around the previously detected corelines giving an indication of the vortex flow behavior at the selected corelines.
blue colored stream arrows. Note that the proportions used to construct the stream arrows are the same for every stream arrow.

Continuing with our use case scenario, the user now decides to generate a collection of stream arrows for the vortex that lies in the continuation of the previously visualized vortex. The result is depicted in Figure 6.4 (b). Combining the two previous visualizations results in the situation illustrated in Figure 6.4 (c). From the figures we can derive that both vortices are repelling vortices, as the distance of the stream arrows to the coreline increases as they progress along the coreline. This information is also available by first selecting a vortex and then clicking the Details... button located on the detection tab. A small popup window informs the user of the type of vortex, the amount of corepoints that make up the vortex coreline and so forth. Continuing our analysis of Figure 6.4 (c), we derive that the left vortex is rotating clockwise, whereas the right vortex rotates counter-clockwise, as is already suggested by the colors of both corelines. Different stream arrow variations were used to be able to properly distinguish between the stream arrows associated with the left and right vortex. Note that the arrow head of the last rectangular stream arrow is capped, since it extends beyond the bounded streamline.

In case the user is interested in the results of another vortex detection method, it is possible to select another detection method and run the vortex detection process again. This will result in a new list of detected vortices, which will be displayed instead of the previously detected vortices. Note that the detection results associated with every detection method are retained in memory. Thus, it is possible to switch back to the detection results of any previously employed detection method.

Now, after having analyzed the detected vortices, the user can save the detection results to a human-readable XML based experiment file. The XML file contains all the data associated with every detected vortex, such as the vortex strength at every point along the coreline, the velocity vector at every point along the coreline, but also whether the vortex was explicitly hidden by the user. Furthermore, information about the datasets associated with every time step is stored in the XML file.

The user may edit the XML file by hand, to add basic graphic primitives representing obstacles present in the original flow, as was done for the experiment in this use case scenario. In Listing 6.1, parts of the XML code taken from the experiment file loaded in Figure 6.2 are displayed. Unfortunately, at the moment there is no way to add basic graphic primitives using the application itself.

### 6.1.2 A multiple timestep use-case scenario

Using VortexView, it is also possible to track the development of vortices over time. As in the previously described use case, this is done by creating a new experiment, and adding datasets to the experiment by using the ‘+’ button in the Experiment tab. The ordering of the datasets as listed should correspond with the ordering of the timesteps represented by the datasets. Thus, the first dataset in the list of datasets should correspond with the results of the first timestep in the experiment, whereas the last dataset in the list should correspond with the last timestep in the experiment.

Now, vortex detection proceeds in the same way as in the single timestep use-case scenario. In this case, after clicking the Detect coreline button, the vortices for each dataset in the experiment will be detected.

After vortex detection has been completed, already a rough indication of the way the
vortices develop over time can be obtained by stepping through the detection results for every time step. Using the animation controls present in the toolbar, VortexView automatically steps through the detection results for every time step, where each timestep in the experiment corresponds with one frame in the animation. From the preferences dialog, which can be reached by selecting Preferences from the Edit menu, the number of frames per second can be adjusted to either speed up or slow down the animation.

As discussed in Chapter 4, a streamsurface is generated by tracking a coreline moving through the flow. For every vortex, the streamsurface can be generated by clicking the Track vortices button from the Tracking tab. The Tracking tab is only enabled in case the experiment contains multiple datasets. After clicking the Track vortices button, a small window pops up asking the user for which time window the vortex corelines should be tracked. In case a user has loaded \( N \) different datasets, he or she may track the vortex corelines between the timesteps \( i \) and \( j \), with \( 0 \leq i < j < N - 1 \). After selecting a suitable time window, the tracking process is started in a separate thread. The log window reports on every step performed during the tracking process, while at the same time keeping the user informed about the overall progress of the tracking process.

![Figure 6.5](image_url)

**Figure 6.5:** (a) stream surfaces resulting from tracking a coreline over three timesteps; (b) the meshes of the stream surfaces from (a), resulting from Loop's subdivision algorithm.

After the tracking process has finished, the resulting stream surfaces are displayed. Using datasets from the same experiment that was used in the previous use case scenario, some of the resulting stream surfaces are displayed in Figure 6.5. It is possible to simultaneously display stream surfaces resulting from multiple time steps at once, by specifying a vortex fading factor value \( f \) between 0 and 1 on the Vortex visualization tab. Then, when viewing the results from time step \( i \), the vortex and associated stream surface from time step \( i - k \) will be displayed with an opacity of \( 1 - k \cdot f \). A fully opaque model has an associated opacity value of 1, whereas a fully transparent model has an associated opacity value of 0.
 CHAPTER 6. VORTEXVIEW

Figure 6.6: Architectural overview of VortexView. VVTK is a collection of classes supplementing VTK and providing functionality required by both the vortex detection and vortex tracking algorithms.

6.2 Implementation of VortexView

Our visualization tool is implemented using C++. The wxWidgets [2] library was used to implement the user interface. Currently, only an implementation for GNU/Linux based systems exists, but it should be fairly straightforward to port the code for use under, e.g., Microsoft Windows™. We use VTK [1] to provide us with a visualization framework and at the same time providing basic visualization primitives. Both toolkits are open source and free to use. Both the vortex detection code and the vortex tracking code are implemented as supplemental classes to the VTK library, and are bundled as a sub-library called The Vortex Visualization Toolkit (VVTK). Figure 6.6 gives an architectural overview of VortexView.

The VTK library makes a distinction between data objects and process objects. Data objects represent information, and provide methods to create, access, and delete this information. Examples of data objects include vtkStructuredGrid and vtkPointData, representing a structured grid and a container of vertex property data respectively. Process objects operate on input data to generate output data. Three different kinds of process objects are distinguished: source objects, filter objects, and mapper objects. Source objects generate output data using a external data as input, or locally based on certain input data. An example source class is vtkStructuredGridReader, which is used to read a structured grid dataset from file. Filter objects generate output data based on input data in the form of a data object. We implemented a filter class called vtkGradient, which can be used to calculate the gradient field given a vector field using a least linear squares method. Finally, mapper objects terminate the visualization pipeline by turning input data into graphical primitives.

All classes designed for the vortex detection and tracking process are filter classes. They operate on structured grids only. For every step in the vortex detection process a separate class was implemented. The parallel vector operator is implemented by the vtkParallelVector class. It takes as input two structured grids each containing a vector field that is used as an argument for the parallel vector operator. A list of candidate corepoints is returned.

Since every vortex detection method requires additional processing after the parallel vector operator has been applied to two vector fields, additional classes are required. For example, as discussed in Chapter 3, Banks & Singer's vortex detection method requires the corepoints that are returned by the vtkParallelVector class to lie on local pressure minima. We ported Roth's max_min_criterion AVS module to a separate VTK class; vtkMaxMinCriterion. Using the Hessian of any scalar field and a vector, the vtkMaxMinCriterion class projects the hessian at the candidate corepoints perpendicular-
lar to the specified vector. The resulting $2 \times 2$ Hessian is symmetric and has two eigenvalues. Based on eigenvalue analysis it is possible to calculate whether the candidate corepoint lies on a local maximum or minimum with respect to the scalar field. After the detection process is finished, the resulting corepoints are connected based on heuristics defined in Roth’s connect_lines AVS module. This module was ported to a VTK mapper class, vtkConnectLines, and is the last step in the detection process resulting in the final corelines.

For every detection method, compound filter classes were designed functioning as wrapper classes around all the specific steps required for each detection process. Three detection filter classes were implemented; vtkBanksSinger, vtkSujudiHaimes, and vtkRothPeikert, all deriving from vtkCorepointDetectionMethod. They take as their input a flow dataset contained in a vtkStructuredGrid instance, and return polygonal objects representing the final corelines. Our flow datasets, containing relatively large, profound vortices were best detected using Roth & Peikert’s vortex detection method.

The tracking process is implemented by the vtkTracker class. It takes two flow datasets as input representing two subsequent timesteps $i$ and $i + 1$ of a flow experiment. Furthermore, a vtkCorepointDetectionMethod instance indicating the vortex detection method is required, and will be used to detect the space-time corepoints. The result of the tracking process is a collection of polygonal objects representing the stream surfaces that result from the vortices as they move between timestep $i$ and $i + 1$.

Using the threading primitives introduced by the wxWidgets library, calculation intensive operations are performed in separate threads from the main user interface thread. This ensures that the application is responsive to user input all the time. For example, it is possible to track vortices between timestep $i + 1$ and $i + 2$, while at the same time observing the stream surfaces that resulted from tracking the vortices from timestep $i$ to $i + 1$ since the tracking calculations are performed in a separate thread.

The total project consists of more than 24,000 lines of code. Seven bugs were filed against VTK’s code base and subsequently resolved.
Chapter 7

Conclusions

We implemented a vortex detection and a feature based visualization framework, VortexView, based on the parallel vector operator introduced by Roth.

In Chapter 2, we gave a short introduction to feature based visualization in general, and presented a short overview of the state of the art of flow visualization. Then, in Chapter 3, the parallel vector operator as first introduced by Roth was discussed. We presented improvements that were made to the original detection algorithm, and we were able to show that they have a positive influence on the number and quality of the detected vortices.

Bauer & Peikert's vortex tracking algorithm, again based on the parallel vector operator, was discussed next in Chapter 4. We proposed a new optimized stream surface generation algorithm, and we were able to show that our algorithm is roughly 40% faster than Bauer & Peikert's version. Furthermore, the quality of the resulting stream surface is improved by avoiding incorrect stream surface patches to be generated.

In Chapter 5, we introduced a new vortex visualization primitive, the stream arrow. A discussion on the construction of a stream arrow was given. Subsequently, the stream arrow primitive was put in a vortex visualization context. We described a method to generate a collection of streamlines near a vortex coreline, such that the resulting stream arrows give a clear indication as to how the flow behaves within the vortex context. Note that since a streamline can be generated from every non-critical point in the flow, using the stream arrow, global flow properties can be visualized as well.

Finally, in Chapter 6, we briefly discussed some implementation details regarding our visualization tool, VortexView. Also, we presented two use case scenario's of VortexView, and we showed that, using a dataset from the SMARTER research project, the application of stream arrows can improve the insight in the configuration, properties, and behavior over time of a collection of vortices present in the flow dataset.

7.1 Future work

The most time consuming step in both the vortex detection and tracking process is the Newton-Raphson iteration used to find candidate corepoints, as previously discussed in Chapter 3. Future research can be focused on improving the time efficiency of the corepoint detection algorithm. In theory, it should be possible to detect beforehand whether a candidate corepoint can be present on a grid cell in the first place. Ideally, such an early-out detection algorithm is able to judge, based on the two vectors located
at every corner of the face under consideration, whether the Newton-Raphson iteration should be performed or not. Being able to prevent the Newton-Raphson iteration from being performed will save considerable amounts of time.

Using the vortex tracking algorithm and the event heuristics introduced in Chapter 4, it is fairly straightforward to add an event detection framework on top of the vortex visualization toolkit. The stream surfaces that result from the tracking algorithm are connected with \( n_i \) corelines from timestep \( i \), and \( n_{i+1} \) corelines from timestep \( i + 1 \). For example, using the heuristics introduced in Chapter 4, in case \( n_{i+1} > n_i \) holds we know a bifurcation event has occurred. Additionally, the vortex tracking results can be used to augment the vortex detection process in the following way. Suppose the event detection algorithm detects that a dissipation event for a certain vortex \( V \) at timestep \( i \) occurred, whereas a creation event is detected at timestep \( i + 1 \). The cause of these events might be the fact that \( V \) is not properly detected at timestep \( i \), but again is detected at timestep \( i + 1 \). Therefore, the application can decide to increase the resolution of the starting point grid, and repeat the detection process for timestep \( i \), thereby increasing the chance that \( V \) is properly detected for timestep \( i \). Of course, the tracking algorithm then has to be repeated for \( V \) between timestep \( i - 1 \) and \( i \) to check whether the dissipation event still occurs for \( V \).

The efficiency of the streamsurface propagation algorithm can be greatly improved. Currently, when looking for candidate corepoints when propagating a boundary segment located in a grid cell \( G \) at timestep \( i \), all corepoints with time attribute \( t \), for which \( i \leq t \leq i + 1 \) holds, are checked. However, this is unnecessary, as only the corepoints located in grid cell \( G \), and optionally all surrounding grid cells, need to be considered. This requires an additional bucket datastructure where each grid cell number maps to a collection of corepoints. Obviously, this will greatly improve the speed efficiency of the algorithm. In practice, since we only tested our algorithm for relative small datasets with only a small number of vortices, the current implementation met our speed requirements. However, for larger datasets with a large number of vortices, the proposed optimizations might be required to still achieve proper performance.

Currently, no time-dependent version for the stream arrow visualization primitive exists. One possible research direction would be to base the time-dependent version of a stream arrow on path lines instead of streamlines. Additional research is required to determine whether this is possible in the first place, since several problems are associated with the idea. For example, not enough datasets might be available to generate a pathline that extents over the length of the vortex that needs to be visualized. Another problem is where to start the path line in such a way that it does not diverge from the associated vortex over time.

Additionally, the stream arrow primitive itself can be extended by allowing the dimensions of the slices along the stream arrow center line to be varied according to the flow.

Finally, VortexView itself can be extended in several directions. For example, adding additional vortex detection methods should be relatively easy. Also, adding basic flow visualization primitives like stream ribbons and stream balls is straightforward as well, since VTK already features classes that implement these basic visualization primitives.
## List of Figures

3.1 Velocity and vorticity distribution for a solid body rotation .................................. 19  
3.2 Velocity and vorticity distribution for a potential vortex ...................................... 19  
3.3 Velocity and vorticity distribution for a Rankine vortex ....................................... 20  
3.4 Vortex strength calculation ...................................................................................... 21  
3.5 Critical point classification ..................................................................................... 22  
3.6 Banks & Singer predictor-corrector method .............................................................. 24  
3.7 Cell faces considered by the parallel-vector algorithm ............................................. 26  
3.8 An example graph of $|h(\sigma_i, \sigma_j)|$ defined over the gray-shaded cell face ....... 27  
3.9 A $4 \times 4$ starting point grid on top of a cell face .................................................... 29  
3.10 Number of detected vortices and time required for the detection process in relation with the amount of starting points .................................................. 29

4.1 Tracking 0D features in 1D space .................................................................................. 33  
4.2 A 4D hypercube or tesseract ...................................................................................... 34  
4.3 Application of Bauer & Peikert's tracking algorithm ................................................ 34  
4.4 Two neighboring hypercubes ................................................................................... 36  
4.5 Space-time cell ........................................................................................................ 36  
4.6 Example situation where Bauer & Peikert's algorithm might result in an incorrect streamsurface patch ................................................................. 37  
4.7 Neighboring edges in relation to the proposition $I$ ................................................... 40  
4.8 Propagation of one coreline to another coreline over one grid cell ......................... 41  
4.9 Propagation of one coreline to another coreline ..................................................... 42  
4.10 Problem situation during streamsurface boundary propagation ............................. 43  
4.11 The situation sketched in Figure 4.6 re-evaluated using our streamsurface propagation algorithm .............................................................. 45

5.1 Streamline and pathline differences ........................................................................... 48  
5.2 Parameters that can be applied to a two-dimensional arrow to influence its appearance ........................................................................................................... 49  
5.3 Two stream arrows along a streamline ...................................................................... 50  
5.4 Stream arrow construction based on a streamline .................................................... 51  
5.5 Vertex normal calculation for vertices making up the arrow head ......................... 52  
5.6 Mesh of the base of a stream arrow .......................................................................... 52
5.7 Vertex normals associated with the slices connecting the stream arrow base with the stream arrow head ................. 53
5.8 Mesh skew due to separately calculated local coordinate frames .... 54
5.9 Lobed circular slice ........................................... 55
5.10 Lobed stream arrow mesh .................................... 56
5.11 Rectangular slice ............................................. 57
5.12 Rectangular stream arrow mesh ............................. 57
5.13 Streamline spiraling out of the vortex region ............... 58
5.14 Schematic 2D vortex slices illustrating attracting and repelling vortex streamlines ................................. 60
5.15 Streamline seed point layouts ............................... 60
5.16 Different stream arrow variations circling around a coreline .... 63
6.1 Cutting plane depicting the velocity magnitude ............... 66
6.2 Corelines resulting after a vortex detection run .............. 68
6.3 The stream arrow editor ...................................... 69
6.4 Stream arrows around the previously detected corelines ..... 70
6.5 Stream surfaces .................................................. 72
6.6 Architectural overview of VortexView ......................... 73
Bibliography


Appendix A

Vector Calculus

A.1 Notation

To minimize confusion, we introduce our notation for vectors, vector fields and scalar fields. We denote vectors as a lowercase boldface letter such as $v$, $w$, \ldots.

Example 1 (row vectors and column vectors):
An $n$-component row vector is of the form
\[ v = \begin{pmatrix} v_1 & v_2 & \cdots & v_n \end{pmatrix}. \]
For instance, $v = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$. An $n$-component column vector is of the form
\[ v^T = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}. \]
For instance, $v^T = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$.

The partial derivative of the function $f(x, y) = z$ with respect to the variable $x$ is denoted as
\[ \frac{\partial z}{\partial x} = \frac{\partial}{\partial x} f(x, y), \]
or as a shorthand notation $f_1(x, y)$, where the subscript ‘1’ refers to the first variable of $f$. The partial derivative of $f$ with respect to the variable $y$ is denoted as
\[ \frac{\partial z}{\partial y} = \frac{\partial}{\partial y} f(x, y). \]

Again, as a shorthand notation we use $f_2(x, y)$. In case $f$ is a vector function, we use the notation $f_{1,x}$ to denote the $x$-component of the partial derivative of $f$ with respect to the first variable.

Values of a partial derivative $f_1$ at a particular point $(a, b)$ are denoted as
\[ \left. \frac{\partial z}{\partial x} \right|_{(a, b)} = \left. \left( \frac{\partial}{\partial x} f(x, y) \right) \right|_{(a, b)} = f_1(a, b). \]
A good introduction to basic calculus concepts like the partial derivative can be found in [4, 21].

A.2 Definitions

Let us now formally define the terms scalar field and vector field.

**Definition 6 (Scalar Field):**
A scalar field \( k \) on the \( n \) dimensional Euclidean space \( \Omega \subset \mathbb{R}^n \) is a real-valued function of \( n \) spatial variables defined by \( k : \Omega \rightarrow \mathbb{R} \).

**Definition 7 (Vector Field):**
A steady vector field \( v \) on the \( n \) dimensional Euclidean space \( \Omega \subset \mathbb{R}^n \) is a real-valued \( m \)-component vector function of \( n \) spatial variables defined by \( v : \Omega \rightarrow \mathbb{R}^m \).

Definition 6 and 7 pertain to steady scalar and vector fields only, i.e., fields that do not vary as a function of time. Time-dependent fields, also referred to as varying fields, are defined as follows.

**Definition 8 (Varying Scalar Field):**
A varying scalar field \( l \) on the \( n \) dimensional Euclidean space \( \Omega \subset \mathbb{R}^n \) is a real-valued function of \( n \) spatial variables, and a time variable \( t \in \mathbb{N} \) defined by \( l : \Omega \times \mathbb{N} \rightarrow \mathbb{R} \).

**Definition 9 (Varying Vector Field):**
A varying vector field \( w \) on the \( n \) dimensional Euclidean space \( \Omega \subset \mathbb{R}^n \) is a real-valued \( m \)-component vector function of \( n \) spatial variables, and a time variable \( t \in \mathbb{N} \) defined by \( w : \Omega \times \mathbb{N} \rightarrow \mathbb{R}^m \).

A.3 Scalar and Vector Field Operations

We proceed by discussing several operations defined on scalar or vector fields which are frequently used in the field of flow visualization. Both the mathematical definition as well as the inherent (geometrical) interpretation of each operation is discussed. We only give definitions for each operator operating on steady scalar and vector fields. All definitions can easily be extended to time-dependent fields by adding an extra time variable \( t \in \mathbb{N} \). For practical purposes, we only consider fields on the 3 dimensional Euclidean space.

A.3.1 Gradient

The gradient is an operator defined on scalar fields with a vector field as result. Definition 10 gives a definition for the gradient operator.
**Definition 10 (Gradient):**
At any point \((x, y, z)\) where the first partial derivatives of the scalar function \(f(x, y, z)\) exist, the gradient is the vector function defined by

\[
\text{grad } f = \frac{\partial f}{\partial x} \mathbf{i} + \frac{\partial f}{\partial y} \mathbf{j} + \frac{\partial f}{\partial z} \mathbf{k}
\]

where \(i, j, \text{ and } k\) represent the unit basis vectors, with

\[
i = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad j = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \text{and} \quad k = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}
\]

As a shorthand notation, the nabla or del differential operator is often used:

**Definition 11 (Nabla Operator):**
The nabla or del differential operator, denoted as \(\nabla\), is defined by

\[
\nabla = \frac{\partial}{\partial x} \mathbf{i} + \frac{\partial}{\partial y} \mathbf{j} + \frac{\partial}{\partial z} \mathbf{k}.
\]

We can now write the gradient of \(f(x, y, z)\) as \(\nabla f\). For example, if \(f(x, y, z) = 5x + 4y^2z + 13z\), then \(\nabla f = 5i + 8yzj + (4y^2 + 13)k\). \(\nabla f(x, y, z)\) results in the gradient vector associated with the scalar value \(f(x, y, z)\). The result of the gradient operator is a directional derivative. Examples of directional derivatives are the partial derivatives \(f_1(a, b, c)\) and \(f_3(a, b, c)\). They give the rate of change at point \((a, b, c)\) measured in the directions of the positive \(x\)- and \(z\)-axes respectively, for a scalar function \(f\).

**Physical Interpretation**

We will first define the directional derivative, and then explain how it relates to the result of the gradient operator.

**Definition 12 (Directional Derivative [4]):**
Consider the unit vector \(\mathbf{u} = ui + vj + wk\). The directional derivative of \(f(x, y, z)\) at \((a, b, c)\) along a ray in the direction of \(\mathbf{u}\) is the rate of change of \(f(x, y, z)\) with respect to the distance measured at \((a, b, c)\) along a ray in the direction of \(\mathbf{u}\) in the \(xyz\)-space. This directional derivative is given by

\[
D_uf(a, b, c) = \left. \frac{d}{dt} f(a + tw, b + tv, c + tw) \right|_{t=0}.
\]

provided that the derivative on the right side exists.
The directional derivatives in the directions parallel to the coordinates axes result in the first partials.

**Example 2 (directional derivative):**

Consider the directional derivative in the direction of the unit basis vector $i$, with $u = 1$, $v = 0$, and $w = 0$. Using the Chain Rule we obtain

$$D_i f(a, b, c) = \left. \frac{d}{dt} f(a + t, b, c) \right|_{t=0}$$

$$= \left. \frac{\partial f(a + t, b, c)}{\partial x} \frac{\partial x}{\partial t} \right|_{t=0}$$

$$= \left. \frac{\partial f(a, b, c)}{\partial x} \right|_{t=0}$$

$$= f_1(a, b, c)$$

with $x = a + t$. Thus, the directional derivative in the direction of the unit basis vector $i$ at $(a, b, c)$ equals the partial derivative of $f(x, y, z)$ with respect to $x$ at point $(a, b, c)$.

Using the previous definition of the gradient and the directional derivative, we now give a different definition of the directional derivative.

**Theorem 1 (The directional derivative in terms of the gradient [4]):**

Suppose that $f(x, y, z)$ is differentiable at $(a, b, c)$, and $\mathbf{u} = u\mathbf{i} + v\mathbf{j} + w\mathbf{k}$ is a unit vector, then the directional derivative of $f$ at $(a, b, c)$ in the direction of $\mathbf{u}$ is given by

$$D_\mathbf{u} f(a, b, c) = \mathbf{u} \cdot \nabla f(a, b, c).$$

**Proof** With $x = a + tu$, $y = b + tv$, $z = c + tw$, and using the Chain Rule

$$D_\mathbf{u} f(a, b, c) = \left. \frac{d}{dt} f(a + tu, b + tv, c + tw) \right|_{t=0}$$

$$= \left. \frac{\partial f}{\partial x} \frac{\partial x}{\partial t} \right|_{t=0} + \left. \frac{\partial f}{\partial y} \frac{\partial y}{\partial t} \right|_{t=0} + \left. \frac{\partial f}{\partial z} \frac{\partial z}{\partial t} \right|_{t=0}$$

$$= u f_1(a, b, c) + v f_2(a, b, c) + w f_3(a, b, c) = \mathbf{u} \cdot \nabla f(a, b, c).$$

Now we get back to the physical interpretation of the gradient. Using theorem 1 we can write

$$D_\mathbf{u} f(a, b, c) = \mathbf{u} \cdot \nabla f(a, b, c) = |\nabla f(a, b, c)| \cos \theta$$

where $\theta$ is the angle between the vectors $\mathbf{u}$ and $\nabla f(a, b, c)$. Since $\cos \theta$ only takes on values between -1 and 1, $D_\mathbf{u} f(a, b, c)$ only takes on values between $-|\nabla f(a, b, c)|$ and $|\nabla f(a, b, c)|$. The directional derivative takes on its maximum value $|\nabla f(a, b, c)|$ when $\mathbf{u}$ is parallel to $\nabla f(a, b, c)$, since then $\cos \theta = 1$ holds.

This gives rise to the physical interpretation of the gradient — at $(a, b, c)$, $f(x, y, z)$ increases most rapidly in the direction of $\nabla f(a, b, c)$, where the maximum rate of increase is $|\nabla f(a, b, c)|$. 
Another important observation is that for any vector \( v \) perpendicular to \( \nabla f(a, b, c) \), the rate of change at \((a, b, c)\) in the direction of \( v \) is zero, since \( \cos \theta = 0 \) holds.

Finally, observe that \( \nabla f(a, b, c) \) is a normal vector of the level surface \( S \) of \( f(x, y, z) \) that passes through \((a, b, c)\). Therefore, any vector perpendicular to \( \nabla f(a, b, c) \) is tangent to \( S \).

### A.3.2 Vorticity

Vorticity is a physical property of a flow that corresponds with the curl of a velocity field.

**Definition 13 (Curl):**

Let \( \mathbf{v}(x, y, z) = v_1 \mathbf{i} + v_2 \mathbf{j} + v_3 \mathbf{k} \) be a differentiable vector function. A vector function defines a vector field, previously defined in Section A.2. Then the function

\[
\text{curl } \mathbf{v} = \nabla \times \mathbf{v} = \begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\
\frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\
v_1 & v_2 & v_3
\end{vmatrix}
\]

\[
= \left( \frac{\partial v_3}{\partial y} - \frac{\partial v_2}{\partial z} \right) \mathbf{i} + \left( \frac{\partial v_1}{\partial z} - \frac{\partial v_3}{\partial x} \right) \mathbf{j} + \left( \frac{\partial v_2}{\partial x} - \frac{\partial v_1}{\partial y} \right) \mathbf{k}
\]

is called the curl of the vector function \( v \).

**Physical Interpretation**

The physical interpretation of the curl can best be explained using Stoke’s theorem. Therefore, we briefly introduce Stoke’s theorem, after giving a short introduction on line and surface integrals which form the essence of Stoke’s theorem.

A line integral is an integration of a given function along a curve \( C \) in the plane or in space. A curve \( C \) is represented by a parametric representation

\[
\mathbf{r}(t) = x(t) \mathbf{i} + y(t) \mathbf{j} + z(t) \mathbf{k}.
\]

The path of integration is defined from an initial point \( \mathbf{r}(a) \) to a terminal point \( \mathbf{r}(b) \), with \( a \leq t \leq b \). We assume \( \mathbf{r}(t) \) is differentiable and the derivative \( \mathbf{r}'(t) = \frac{d\mathbf{r}}{dt} \) is continuous and different from the zero vector at every point of \( C \).

**Definition 14 (Line Integral):**

The line integral of a vector function \( \mathbf{F}(\mathbf{r}) \) over a curve \( C \) characterized by \( \mathbf{r}(t) \), with \( a \leq t \leq b \), is defined as

\[
\int_C \mathbf{F}(\mathbf{r}) \cdot d\mathbf{r} = \int_a^b \mathbf{F}(\mathbf{r}(t)) \cdot \frac{d\mathbf{r}}{dt} dt
\]

If \( \mathbf{r}(a) = \mathbf{r}(b) \), i.e., the path of integration \( C \) in definition 14 is a closed curve, we may also write

\[
\int_C \mathbf{F}(\mathbf{r}) \cdot d\mathbf{r} \quad \text{as} \quad \oint_C \mathbf{F}(\mathbf{r}) \cdot d\mathbf{r}.
\]
Note that the result of a line integral is a scalar value. It denotes the tangential component of $F$ along $C$.

We now give a formal definition for a surface integral.

**Definition 15 (Surface Integral):**
Let $S$ be a surface in space defined by the parametric representation

$$r(u, v) = x(u, v)i + y(u, v)j + z(u, v)k$$

with $(u, v)$ in $R$, where $R$ is some region in the $(u, v)$ plane. $S$ has a normal vector defined by the function $N(u, v) = r_u(u, v) \times r_v(u, v)$, with unit normal vector $n = \frac{1}{|N|}N$. For a given vector function $F$, we can define the surface integral over surface $S$ as

$$\int \int_S F \cdot n \, dA = \int \int_R F(r(u, v)) \cdot N(u, v) \, du \, dv.$$ 

Stoke’s theorem shows how to express a surface integral as a line integral, and vice versa.

**Theorem 2 (Stoke’s Theorem [21]):**
Let $S$ be a surface in space and let the boundary of $S$ be closed curve $C$ characterized by a parametric representation $r(t)$. Then

$$\int \int_S (\nabla \times F) \cdot n \, dA = \oint_C F \cdot dr$$

holds.

For a proof of Stoke’s theorem, refer to [4, 21]. The curl characterizes rotation in a vector field. Using Stoke’s theorem, we can now give a more precise explanation of the physical interpretation of the curl [21].

Let $S_r$ be a circular disc of radius $r$ and center $P$ bounded by circle $C_r$, which is characterized by a parametric representation $r(t)$. Furthermore, let $F(x, y, z)$ be a continuously differentiable vector function in a domain containing $S_r$. $n(Q)$ is the unit normal vector of $S_r$ at $Q$, with $Q$ a point of $S_r$. Using Stoke’s theorem (Theorem 2) and the mean value theorem for surface integrals [21], we may write

$$\oint_{C_r} F \cdot dr = \int \int_{S_r} (\nabla \times F) \cdot n \, dA = (\nabla \times F) \cdot n(P')A_r$$

with $A_r$ the area of $S_r$, and $P'$ a point of $S_r$ for which the mean-value theorem holds. This can be rewritten to

$$(\nabla \times F) \cdot n(P') = \frac{1}{A_r} \oint_{C_r} F \cdot dr.$$ 

In our case, where the vector function $F$ is usually represented by a velocity field $v$, the integral

$$\oint_{C_r} v \cdot dr$$

would represent the circulation of the fluid around the boundary of the disc.
is called the circulation of the flow around $C_r$. The integral measures the extent to which the fluid motion is a rotation around the circle $C_r$. If we let the radius $r$ approach zero, we find

$$\left(\nabla \times \mathbf{v}\right) \cdot \mathbf{n}(P) = |\nabla \times \mathbf{v}| \cos \theta = \lim_{r \to 0} \frac{1}{A_r} \oint_{C_r} \mathbf{v} \cdot d\mathbf{r}$$

with $\theta$ the angle between $\nabla \times \mathbf{v}$ and $\mathbf{n}(P)$.

Here we have the interpretation of the curl. The component of the curl in the direction of the positive normal direction of a surface $S$ can be regarded as the circulation per unit area of the flow in the surface $S$ at the corresponding point. The direction of the curl describes the local axis of that (right-handed) rotation. Again, the curl of a physical flow field (velocity field), is called its vorticity.

### A.3.3 Helicity

Helicity is strongly related to vorticity. It is a scalar value indicating the amount of vorticity at a certain point in the flow around the velocity axis.

**Definition 16 (Helicity):**
Assume we have a 3-dimensional vector field $\mathbf{v}$ as defined by Definition 7. Then, the helicity at a point $p$ in $\mathbf{v}$ is given by

$$\left(\nabla \times \mathbf{v}(p)\right) \cdot \mathbf{v}(p).$$