2012-03-09

Application of Subjective Logic to Vortex Core Line Extraction and Tracking from Unsteady Computational Fluid Dynamics Simulations

Ryan Phillip Shaw
Brigham Young University - Provo

Application of Subjective Logic to Vortex Core Line Extraction and Tracking from Unsteady Computational Fluid Dynamics Simulations

Ryan Phillip Shaw

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Steven E. Gorrell, Chair
R. Daniel Maynes
Julie C. Vanderhoff

Department of Mechanical Engineering

Brigham Young University

April 2012

ABSTRACT

Application of Subjective Logic to Vortex Core Line Extraction
and Tracking from Unsteady Computational
Fluid Dynamics Simulations

Ryan Phillip Shaw
Department of Mechanical Engineering, BYU
Master of Science

Presented here is a novel tool to extract and track believable vortex core lines from unsteady Computational Fluid Dynamics data sets using multiple feature extraction algorithms. Existing work explored the possibility of extracting features concurrent with a running simulation using intelligent software agents, combining multiple algorithms' capabilities using subjective logic. This work modifies the steady-state approach to work with unsteady fluid dynamics and is designed to work within the Concurrent Agent-enabled Feature Extraction concept. Each agent's belief tuple is quantified using a predefined set of information. The information and functions necessary to set each component in each agent's belief tuple is given along with an explanation of the methods for setting the components. This method is applied to the analyses of flow in a lid-driven cavity and flow around a cylinder, which highlight strengths and weaknesses of the chosen algorithms and the potential for subjective logic to aid in understanding the resulting features. Feature tracking is successfully applied and is observed to have a significant impact on the opinion of the vortex core lines. In the lid-driven cavity data set, unsteady feature extraction modifications are shown to impact feature extraction results with moving vortex core lines. The Sujudi-Haimes algorithm is shown to be more believable when extracting the main vortex core lines of the cavity simulation while the Roth-Peikert algorithm succeeding in extracting the weaker vortex cores in the same simulation. Mesh type and time step is shown to have a significant effect on the method. In the curved wake of the cylinder data set, the Roth-Peikert algorithm more reliably detects vortex core lines which exist for a significant amount of time. the method was finally applied to a massive wind turbine simulation, where the importance of performing feature extraction in parallel is shown. The use of multiple extraction algorithms with subjective logic and feature tracking helps determine the expected probability that an extracted vortex core is believable. This approach may be applied to massive data sets which will greatly reduce analysis time and data size and will aid in a greater understanding of complex fluid flows.

Keywords: Feature Extraction, Feature Tracking, Vortex Core Lines, Computational Fluid Dynamics, Subjective Logic noabstract

ACKNOWLEDGMENTS

The author would like to thank his thesis advisor Dr. Steven Gorrell for his guidance and direction in the research as well as assistance in editing this thesis. The author would also like to thank committee members Dr. Daniel Maynes and Dr. Julie Vanderhoff for their input and aid. The author would like to thank the Air Force Office of Scientific Research for sponsoring the project through a Phase II STTR. Thanks also goes out to Dr. Robert Woodley and Mike Gosnell of 21st Century Systems, Inc. for their partnership in this research. Many thanks go to the CAFÉ research group – Matt Lively, Matthew Marshall, Kevin Hoopes, and Joshua Wilson – for assistance in the research and many interesting discussions about a wide variety of subjects. Finally, the author would like to thank his wife Ashley and son Kaleb for their wonderful support through the entire process of research and writing this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

a	Atomicity
$\boldsymbol{a}$	Acceleration, m/s^2
AA	Algorithm Agent
AA_E	Extracting Algorithm Agent
AA_NE	Non-extracting Algorithm Agent
b	Belief
$\boldsymbol{b}$	Jerk, m/s^3
C	Curvature of vortex core line
CAFÉ	Concurrent Agent-enabled Feature Extraction
$Corr$	Feature tracking line correspondence
d	Disbelief
D	Diameter
E	Probability expectation
$\boldsymbol{f}$	Feature Flow Field
J	Jacobian; Velocity gradient tensor $\nabla \boldsymbol{u}$
l	Line segment length
L	Total vortex core line length
MA	Master Agent
$\boldsymbol{\Omega}$	Rotation tensor
$\boldsymbol{\omega}$	Opinion; Belief tuple
P	Coordinate center of the vortex core line bounding box
PV	Parallel Vectors operator
R	Radius
Re	Reynolds number
RP	Roth-Peikert algorithm
$\boldsymbol{S}$	Strain rate tensor
S	Vortex strength
SH	Sujudi-Haimes algorithm
t	Time
T_{func}	Tracking tolerance for select attribute function
$\boldsymbol{t}$	Vortex core tangent vector
u	Uncertainty
$\boldsymbol{u}$	Velocity, m/s
$\boldsymbol{u_r}$	Reduced velocity, m/s
$\boldsymbol{\zeta}$	Vorticity

Mathematical Operators

$\oplus$	Consensus operator
$\otimes$	Discounting operator
Δ	Difference
∇	Gradient operator

CHAPTER 1. INTRODUCTION

1.1 Motivation

Computational Fluid Dynamics (CFD) is a discipline in which the equations governing fluid flow and heat transfer in a system are numerically solved on a computational mesh. With ever-increasing computational resources available to researchers, the ability to simulate complex fluid flows using CFD becomes progressively more feasible. The simulation of unsteady flows has also been recognized as a more accurate method of modeling the flow in complicated systems such as turbomachinery [1, 2]. The fine meshes required for a simulation can contain tens to hundreds of millions of nodes, and time-dependent data sets consist of many time steps, which may generate terabytes of raw data. A growing challenge due to the size of these data sets lies in analysis and post-processing, which can be extremely time-consuming and require large amounts of storage space. The problem is exacerbated in time-dependent simulations, where regions of interest may not be stationary and fluid interactions become more important.

Currently, post-processing is accomplished by the expertise of the analyst, and often much of the flow field is ignored due to prior prejudice or incomplete knowledge of the flow domain. Simple visualization techniques such as cutting planes, contour plots, and stream traces require a correct choice of placement and often the view becomes very cluttered. Isosurfaces may also be useful, but correct choice of scalar field and values are vital for viewing different flow structures. In time-dependent flows, the number of time steps under consideration requires an even greater effort to visualize the physics of the simulation, including development and interactions in the flow. In massive data sets, much of the physics is ignored while extracting time-averaged or surface flows.

Software programs have been created to assist in the visualization of large-scale CFD data sets. Some commercial packages, including Tecplot [3] and Ensight [4], include advanced post-processing techniques such as data mining to aid in viewing data sets. Data mining is defined as a method for analyzing large amounts data from different perspectives and summarizing it into useful information. Brigham Young University and 21st Century Systems, Inc. (21CSI) are creating a new data mining concept, called Concurrent Agent-enabled Feature Extraction (CAFÉ) to combine multiple feature extraction algorithms in an intelligent fashion. This research is a component of the CAFÉ program.

1.2 Feature Extraction

Analysts are often interested in viewing basic flow features in the data set to understand the physics of the flow field. Post et al. explained features as “phenomena, structures or objects in a data set, that are of interest for a certain research or engineering problem” [5]. Common features of interest in CFD include vortices, shock waves, and separation and attachment lines. Viewing these features as geometric primitives like lines and surfaces allows for fast location of important regions in the simulation.

To aid in visualization and reduce post-processing time, data mining algorithms and methods have been researched and created which “extract” relevant flow features in a simulation. These feature extraction algorithms employ flow variables obtained from the CFD simulation and may use cell or point values. Extraction algorithms may be as simple as finding regions within a certain flow property threshold, or they may calculate complex higher-level variables in order to locate specific features. Feature extraction methods have been created for use in steady-state and unsteady simulations.

Depending on the desired feature, there have been various algorithms created, each of which have respective strengths and weaknesses. Post et al. [5] provided an excellent review of the different methods for the extraction of features and concluded that there were many methods for feature extraction and tracking with little quantitative comparison between extraction algorithms. Ma [6] stated, “it is clear that there is no single best shock detection...algorithm.” Similarly, Roth [7] declared, “none of the [vortex extraction] methods is clearly superior in all the tested data sets.” This means that multiple feature extraction algorithms are required in order to successfully find all important features in a data set.

When extracting vortices Roth suggested the following:

> *An idea for a follow-up project situated in computer science is adding methods from computer vision and AI [artificial intelligence] techniques to combine the various proposed definitions into a single system. Such a system would calculate the vortex cores according to a set of definitions, and then try to use knowledge about the strengths and weaknesses of each method to determine a single set of vortex cores. For example, as long as the resulting vortices are sufficiently strong or almost straight, the zero curvature definition produces very good results. So by adding higher-level post-processing and considering the various feature detection algorithms as specialized knowledge bases, one could use a rule-based AI system to decide which definitions are most likely to give the best results in each particular situation.*

1.3 Feature Tracking

Feature tracking is important in unsteady data sets for analyzing important feature events and interactions. Often it is desired to understand how a feature evolves over time and the interactions that occur between different features in the data set. Many methods have been created to automate feature tracking, and they all attempt to solve what is known as the "correspondence problem" – matching all relevant features in all time steps of interest. As an example, consider the interaction of shock waves and vortices, a problem which has been researched by many [8,9]. As a vortex passes through the shock wave, it may be desired to view how the vortex changes in shape, direction, strength, etc. Feature tracking is also useful for understanding trends in the data set and the effects of design changes on the computational flow field. When many different features exist in the data set, it is desirable to automate tracking to aid the analyst.

Many methods have been proposed to track features over time, falling into two main categories: tracking as a post-processing step to extraction, and tracking coupled with extraction. The post-processing methods include region-based [10] and attribute-based [11] methods. Post-processing methods work quickly because of the data reduction during feature extraction, and attribute-based methods do well with event detection. However, existing approaches usually operate on region-type features (i.e. isosurfaces) and modifications must be made to apply the post-processing approach to line-type features. Coupled methods include feature flow fields [12] and scale space [13], among others, and these are attractive because of their run time capabilities and their ability to detect events. The coupled methods are more complex and often employ higher dimensional vector fields or surfaces.

1.4 Subjective Logic

Subjective logic [14–16] is a mathematics-based logic system that represents opinions which account for uncertainty in a system state using four basic elements: belief (*b*), disbelief (*d*), uncertainty (*u*), and atomicity (*a*). Atomicity is used in an opinion to give an a priori weight to a system's uncertainty. The entire opinion, or belief tuple, is shown in Eq. 1.1.

$$\omega = (b, d, u) \tag{1.1}$$

The three opinion values in subjective logic allow agents to form opinions that are not strictly true or false. In other words, if uncertainty exists in a given situation, an agent is not forced to assign belief or disbelief when formulating an opinion. An agent might then formulate an opinion based on how probable an outcome is rather than simply reducing the outcome to a binary situation. Subjective logic is also useful when making decisions about uncertain situations and/or when data is missing or incomplete. For example, missing or incomplete data can be taken into account when formulating a belief tuple's uncertainty value.

Prior work was undertaken to create a framework to utilize subjective logic in CFD data sets. Mortensen [17] utilized a trust network [18] to use subjective logic with multiple feature extraction algorithms. He formulated this method to be applicable in steady-state CFD data sets and the equations defining subjective logic were made on the basis of steady flow. This method was also created to be run concurrent to a running simulation and was meant to help discern the convergence of a simulation. The method was validated on steady-state CFD simulations and steady vortex cores were extracted from these data sets.

1.5 Objective

The objective of this research was to develop a methodology which employed subjective logic to view and track features extracted from transient CFD data sets by existing feature extraction algorithms. The developed method was designed as a part of the CAFÉ concept. The method utilized multiple algorithms, thus leveraging each algorithm's strengths to find different features in the same data set. The existing steady-state method created by Mortensen was modified to work in time-dependent data sets, which includes modifying the feature extraction algorithms as well as the parameters which influence the belief tuple of the features. A feature tracking method was modified to operate on vortex core lines. Feature tracking was accomplished in order to determine the belief of features and view the evolution of the features over time. The method was validated using vortex core lines as features, though other features may be found. Two CFD simulations are shown that contain vortex core lines in an unsteady environment in order to test this work. Vortex core lines were extracted and tracked from these two simulations. This method correctly defines the expected probability of moving vortex core lines in unsteady CFD data sets. Automation of different aspects of this method were also accomplished, which reduces the amount of user interaction and allows the method to be used on a broader range of data sets.

By combining subjective logic and multiple feature extraction algorithms, the most probable features are easily visualized and tracked through time. This method also allows the analyst to view one feature set which contains only highly probable features. Its applicability to massive data sets was shown through a large unsteady CFD data set, and it was shown that there is a significant data size reduction and an increased ease of visualization through use of the method.

1.6 Overview

This document is organized as follows: Chapter 2 gives background on unsteady vortex extraction, feature tracking methods, subjective logic, trust networks, and prior work in steady-state data sets. Chapter 3 outlines the method used to extract and track vortex core lines from unsteady CFD data sets. Chapter 4 shows the implementation of feature extraction and tracking in the agent trust network. Chapter 5 gives results of two benchmark unsteady CFD simulations as well as a large data set to analyze the effectiveness of the method in massive CFD data sets.

Chapter 6 gives recommendations for future research and Chapter 7 gives conclusions about the research.

CHAPTER 2. BACKGROUND & LITERATURE REVIEW

This chapter contains background on fluid vortices as well as prior methods which have been created to extract vortices from CFD data sets. Feature tracking is explained along with the different methods used to track features through time. A background is given on subjective logic, trust networks, and the prior work undertaken to use subjective logic in steady-state CFD feature extraction.

2.1 Vortices

Vortices are fluid structures which are common in many different types of flows, and an understanding of their location and attributes aid in understanding of the flow physics of engineering systems. They occur in areas of high rotation and may be utilized to enhance mixing, such as in a combustion chamber. In other applications, such as turbomachinery, the losses generated by vortices account for lower efficiencies and it is desirable to minimize the effect of vortices. Noise generation by vortices, especially in the case of shock-vortex interactions, is also another active area of current research. In any case, knowledge of vortex location, size, strength, and life is desirable for design changes. When a vortex is found, design geometry or flow conditions may be altered in order to understand the effect of these parameters on fluid vortices.

Though the intuitive concept of a vortex is clear, there is no agreement on a formal definition of a vortex. A well-known vortex, a tornado, may be seen in Figure 2.1. From a technical standpoint, the following definition of Robinson [19] is often used:

> *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.*

An example of this definition may be seen in Figure 2.2, where the wingtip vortex is nearly normal to the photograph, which allows for clear visualization of the vortex. However, this definition is

Figure 2.1: View of a tornado, a popular conception of a vortex. Photo taken by Eric Nguyen [20].

Figure 2.2: Wingtip vortex visualized with smoke. Photo taken from [21].

self-referential, meaning that the vortex core line direction must be known a priori to determine whether there is swirling flow. Also, the velocity of the vortex must also be known in order to select the correct frame of reference.

2.2 Vortex Extraction

A vortex consists of two interacting parts: the center of the vortex, or core line, and the swirling region around the core. The vortex core is the line along which there is zero velocity relative to the velocity of the vortex. Because of this vortex structure, two different general methods have been proposed to extract and visualize vortices: extracting vortex regions and extracting vortex core lines.

In transient flows, the movement of vortices leads to the question of the Galilean invariance of extraction methods. According to Roth [7], a feature extraction method is Galilean invariant "if [the feature extraction result] does not change with the choice of an arbitrary, constantly moving coordinate system." In general, most region-based extraction methods are Galilean invariant, while many of the vortex core line extraction methods rely upon the velocity field and are thus Galilean variant. Methods have been created to treat the Galilean variance of core line detection algorithms.

2.2.1 Extracting Vortex Regions

One vortex region extraction method involves finding regions with high magnitude of vorticity, where vorticity is calculated using Eq. 2.1. While it is true that a vortex region is one with high vorticity, a region of high vorticity may not always be a vortex region. This occurs in boundary layers, though there is no large-scale swirling motion in this case. Villasenor and Vincent [22] employed this method to extract vortex tubes from unsteady data sets. Vorticity-based vortex regions are Galilean invariant.

$$\boldsymbol{\zeta} = \nabla \times \boldsymbol{u} \tag{2.1}$$

Other authors have created methods which employ the velocity gradient tensor for finding vortex regions. Because they employ only the velocity gradient, which is shown in Eq. 2.2, all of these methods are Galilean invariant. In Eq. 2.2, u, v, and w are the components of $\boldsymbol{u}$.

$$\boldsymbol{J} = \nabla \boldsymbol{u} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \tag{2.2}$$

Hunt et al. [23] proposed a method called the Q criterion, where Q is calculated using Eq. 2.3:

$$Q = \frac{1}{2}\left[\|\boldsymbol{\Omega}\|^2 - \|\boldsymbol{S}\|^2\right], \tag{2.3}$$

where $\boldsymbol{S}$ and $\boldsymbol{\Omega}$ are calculated using Eqs. 2.4 and 2.5.

$$\boldsymbol{S} = \frac{1}{2}\left(J + J^T\right) \tag{2.4}$$

$$\boldsymbol{\Omega} = \frac{1}{2}\left(J - J^T\right) \tag{2.5}$$

When $Q > 0$, a vortex exists according to the authors.

Chong et al. [24] created the Δ criterion which is based on the assumption that a vortex region is a region with complex values of $\nabla \boldsymbol{u}$. Jeong and Hussain [25] created a method based on the eigenanalysis of the $\boldsymbol{S}^2 + \boldsymbol{\Omega}^2$ tensor. When two of the three eigenvalues of this symmetric matrix are negative ($\lambda_2 < 0$), a vortex region is found. This method, aptly called the λ_2 criterion, is the most widely used of the above three methods. Haller [26] presented a new region-based vortex detection method called the M_z-criterion, which was created to be rotation invariant as well as Galilean invariant. Each of these methods are Galilean invariant and have been shown to correctly extract vortex regions in many different flow domains, though each method failed in certain tests. Also, in turbomachinery simulations, vortex regions often blend together and most of the domain is shown to be a vortex region.

Vortex region extraction methods work well to quickly visualize vortices in many different data sets. Their complexity is often much less than that of vortex core line extraction methods and aid in quickly looking for key regions of possible vortex activity. Flow variables such as vorticity are well known and extraction of vortices using these variables is often more intuitive than other methods. Because most region extraction methods are Galilean invariant, they are applicable in time-dependent simulations without any modifications.

There are also specific shortcomings related to region extraction methods. First, vortex regions do not help the analyst pinpoint the exact location of the center of swirling flow, and when multiple vortices are closely spaced, the separate vortex regions cannot be easily differentiated. Second, most of the above methods are parameter-dependent, in that the correct choice of values must be known a priori to clearly visualize vortices. As stated above, the λ_2 criterion is satisfied when $\lambda_2 < 0$, but in some simulations, correct vortices have λ_2 values much less than 0, while spurious regions have a λ_2 of slightly less than 0. Regions are often more difficult to visualize than lines, and for these reasons, it is desirable to extract vortex core lines from CFD data sets.

2.2.2 Extracting Vortex Core Lines

Many different algorithms have been created to locate vortex core lines which each have specific strengths and weaknesses. Two specific algorithms were chosen because of their robustness and wide applicability. Two vortex core line extraction algorithms work on the same data set to show that the formulated method works in finding all vortex core lines in a data set. While only two algorithms are used in this research, multiple extraction algorithms may be used to increase the likelihood of finding all relevant features in a flow field.

Parallel Vectors (PV) Operator

Peikert and Roth [27] created the "Parallel Vectors" (PV) operator to group several different vortex core line algorithms into one general method. The general idea of the PV operator is that vortex core lines occur in areas of the data set where two vector fields are parallel, where the vector fields are approximated at nodes or cell centers in the data set. Many different vector fields have been used in the PV operator algorithm to find vortex core lines. Some vector fields include the pressure gradient [28] and vorticity [29]. Roth [7] provided an excellent overview of these several methods and noted their strengths and weaknesses in different flow situations.

Sujudi-Haimes Algorithm

The Sujudi-Haimes (SH) algorithm [30] was the first algorithm chosen for this research. This algorithm was formulated as a robust vortex core line detection algorithm and has been implemented in CFD post-processing software packages such as Ensight 9 [4] and pV3 [31]. The SH algorithm is based on critical point theory and uses eigenvalues and eigenvectors of the velocity gradient tensor.

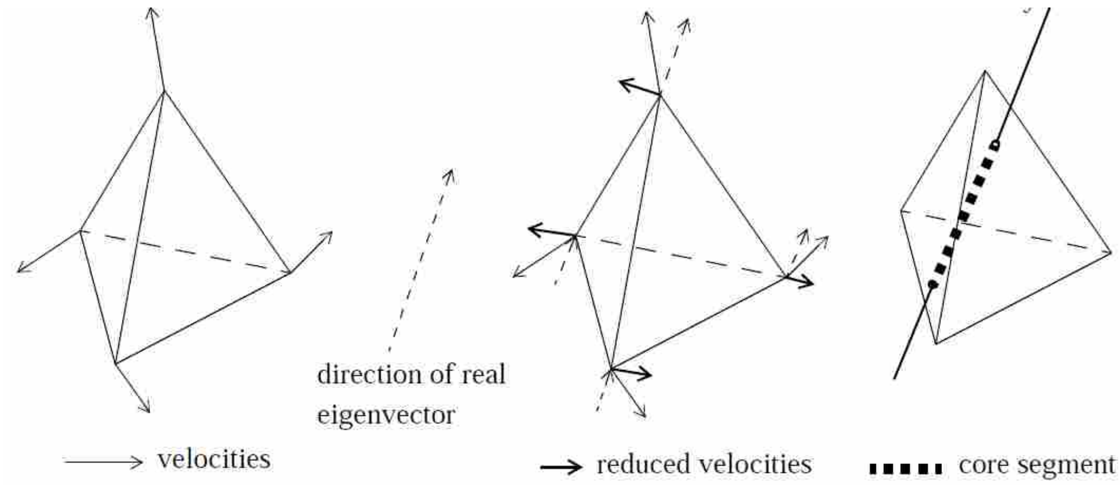


Figure 2.3: Visual representation of the critical point Sujudi Haimes algorithm. When $\boldsymbol{u_r} = 0$ at two points on the cell boundary, a vortex core line segment is added. Image by Martin Roth [7].

The SH algorithm operates on a cell by cell basis and locates points in the domain where the set of eigenvalues contain one real valued and two complex conjugate eigenvalues. Next, the reduced velocity is computed in Eq. 2.6, where $\boldsymbol{n}$ is the normalized eigenvector corresponding to the real eigenvalue. The reduced velocity is then linearly interpolated across the cell, and locations are found along the cell boundaries where $\boldsymbol{u_r} = 0$. If two points are found in the cell where the reduced velocity equals zero, the points are connected and the line segment is added to the vortex core line data set. A visual representation of the SH algorithm may be seen in Figure 2.3. However, gradient computations at neighboring cells produce line segments which do not meet at the cell faces, which results in a set of disjointed line segments in the data set. This method is also computationally intensive because of the eigenanalysis of the entire flow domain.

$$\boldsymbol{u_r} = \boldsymbol{u} - (\boldsymbol{u} \cdot \boldsymbol{n})\boldsymbol{n} \tag{2.6}$$

The assumption of $\boldsymbol{u_r} = 0$ is equivalent to stating the the velocity vector must be parallel to the eigenvector corresponding to the real eigenvalue of the velocity gradient tensor, as shown in Eq.2.7.

$$\boldsymbol{u} \parallel \boldsymbol{e_0} \tag{2.7}$$

Roth and Peikert [32] also showed that the eigenvector from the real eigenvalue may also be expressed as

$$\boldsymbol{u} \parallel \nabla\boldsymbol{u}\cdot\boldsymbol{u} \tag{2.8}$$

This can then be reformulated as

$$\boldsymbol{u} \parallel \boldsymbol{a} \tag{2.9}$$

since

$$\boldsymbol{a} = \frac{D\boldsymbol{u}}{Dt} = \frac{\partial \boldsymbol{u}}{\partial t} + \nabla\boldsymbol{u}\cdot\boldsymbol{u} \tag{2.10}$$

In the original formulation, the partial derivative of velocity with respect to time was neglected, since the algorithm was initially formulated for use in steady-state data sets.

Use of the PV operator allows the SH algorithm to find connected vortex core lines in the flow domain. It finds all points in the domain where Eq. 2.9 is true, then thresholds points with a discriminant of $\nabla\boldsymbol{u} > 0$ to ensure that there is one real eigenvalue and two complex conjugate eigenvalues. Points are connected into lines by use of a search map which finds points who share common cell neighbors. The PV operator version of the SH algorithm is the method used in this research.

The SH algorithm was designed for linear flow fields and thus has inherent strengths and weaknesses due to this assumption. It successfully extracts vortex core lines which are straight and have high vortex strength (high rotational velocity about the core). However, curved vortices or those which have low vortex strength are not well extracted by the SH algorithm. The SH algorithm also performs poorly when the flow has a non-constant acceleration along the vortex core line.

Roth-Peikert Algorithm

The Roth-Peikert (RP) algorithm [7,33] was the second algorithm chosen for this research. Roth and Peikert focused on turbomachinery data sets and formulated their vortex core extraction method to extract vortex core lines specific to these types of flow situations. Whereas the SH algorithm was designed with a linear flow field in mind, the RP algorithm was specifically designed to extract curved vortex core lines, which are more common in data sets with curved flow paths such as turbomachinery.

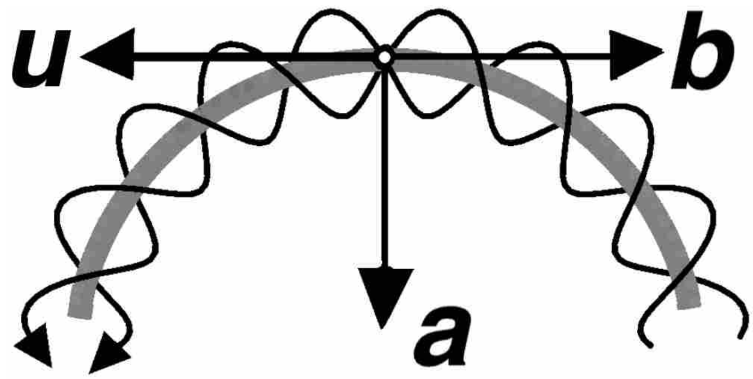


Figure 2.4: Model of a perfectly circular vortex core line with rotating streamlines. Image by Martin Roth [7].

The RP algorithm was designed after the model of a perfectly semi-circular vortex core line. Figure 2.4 shows such a core line with streamlines seeded around it, where the vectors $\boldsymbol{u}$, $\boldsymbol{a}$, and $\boldsymbol{b}$ are velocity, acceleration, and jerk, respectively. In this model, the SH algorithm fails to extract the vortex core line because the velocity is perpendicular instead of parallel to the acceleration. The RP algorithm is thus referred to as a higher-order method because it find points where

$$\boldsymbol{u} \parallel \boldsymbol{b} \tag{2.11}$$

The jerk is the second material derivative of velocity, which is shown in Eq. 2.12.

$$\boldsymbol{b} = \frac{D^2\boldsymbol{u}}{Dt^2} = \frac{\partial^2\boldsymbol{u}}{\partial t^2} + \nabla\left(\nabla\boldsymbol{u}\cdot\boldsymbol{u}\right)\boldsymbol{u} \tag{2.12}$$

In the original RP algorithm, the unsteady term was dropped from the equation, which results in the condition

$$\boldsymbol{u} \parallel \nabla\left(\nabla\boldsymbol{u}\cdot\boldsymbol{u}\right)\boldsymbol{u} \tag{2.13}$$

Using the PV operator, points are found similar to the SH algorithm and lines are aggregated using the same cell search map.

The RP algorithm, similar to the SH algorithm, also has strengths and weaknesses associated with its formulation. Because the RP algorithm was designed to extract a semi-circular model of a vortex core, it does well in extracting curved vortex cores with lower vortex strength than does the SH algorithm. However, due to the computation of higher-order derivatives, the RP algorithm extracts more noise and is more prone to numerical error. The RP algorithm also has a similar weakness to SH in that it may fail when the acceleration along a core line is not constant.

The SH and RP algorithms were chosen for this research for several reasons. First, the strengths and weaknesses of the two algorithms complement each other and ensure that different vortex core lines will be detected by each algorithm in order to prove the concept that multiple algorithms may be used to find all features in the spatiotemporal flow domain. However, both the RP and the SH algorithms are Galilean variant because they rely upon the velocity field to find vortex core points. Because of this, some modification must be made to ensure that the algorithms will correctly extract vortex core lines from time-dependent data sets.

Other Vortex Core Extraction Methods

Several other vortex core extraction algorithms have been created which were not utilized in this research. Many methods use the velocity field and are thus Galilean variant, but Sahner et al. [34] created a Galilean invariant method of extracting the valley or ridge lines of common vortex scalar quantities such as vorticity or λ_2. Jiang [35] created a vortex core line extraction method based on Sperner's lemma in combinatorial topology. Sperner's lemma was originally used to break a large triangle into smaller triangles and then label the subtriangles. It guarantees that any subdivision of a triangle into smaller triangles will result in an odd number of fully labeled triangles. Sperner's lemma can also be applied to 3D vector fields where a vector field is labeled in the same fashion as a triangle. A critical point, or a vortex core line, is found when a triangulation is fully labeled. Filtering must be done to separate saddle regions from the correct set of vortex cores. Other notable vortex core extraction algorithms have been given by Globus et al. [36], Pagendarm et al. [37], and Miura and Kida [38].

2.3 Vortex Core Line Characteristics

Vortex core line characteristics are required to compute the agent opinion in subjective logic. Many different vortex characteristics may be used, but the three variables used to characterize vortex core lines in this research are strength, quality, and curvature.

2.3.1 Vortex Strength

Vortex strength (S) is a measure of the local flow rotation around a vortex core. Vortex strength may be measured in two dimensional flow field by an eigenanalysis of $\nabla \boldsymbol{u}$. Helman and Hesselink [39, 40] characterized such critical points as center and repelling and attracting foci, all of which have only complex conjugate eigenvalues. Vortex strength is defined as the imaginary part of the complex eigenvalues. However, vortex core lines in three-dimensional data sets are rarely constrained to two dimensions, which requires creation of a two-dimensional plane to measure vortex strength. Roth [7] suggested to use a plane perpendicular to the velocity vector at the core line. The local flow field can be projected onto this plane and the local vortex strength can be found from the imaginary part of the complex conjugate eigenvalues.

2.3.2 Quality

Quality is a vortex characteristic originally defined by Roth [7]. Quality is measured as the angle between the vortex core line and the velocity at that point and is computed using Eq. 2.14. Since vortex core lines are really multiple points connected by line segments, the tangent vector $\boldsymbol{t}$ to the vortex core line at a point is the line segment vector at that point which minimizes θ. Roth noted that although vortex core lines are not usually streamlines, they are generally close to a streamline in the flow. This assumption leads to the calculation of a low velocity-core angle, or low quality. A visualization of this characteristic may be seen in Figure 2.5. At the start of the core, the quality is low, which is more likely to be correctly extracted than the end of the core line, which has high quality.

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{u}}{|\boldsymbol{u}|} \cdot \frac{\boldsymbol{t}}{|\boldsymbol{t}|}\right) \tag{2.14}$$

2.3.3 Curvature

Because a major delineation between the RP and SH algorithms is curvature, geometric curvature of the vortex core line is calculated. Curvature is found by circumscribing a circle to points in the vortex core and computing the radius of the circle, as shown in Figure 2.6. Here, A, B, and C represent three points in the vortex core, a, b, and c are the distances between the

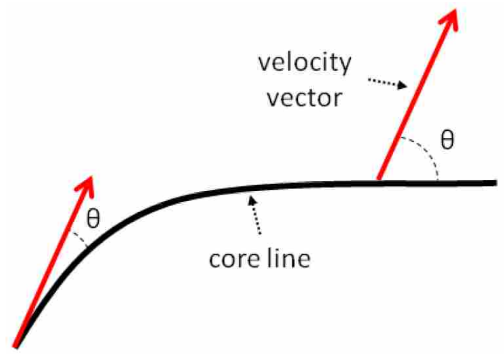


Figure 2.5: Vortex quality at both ends of an extracted vortex core line.

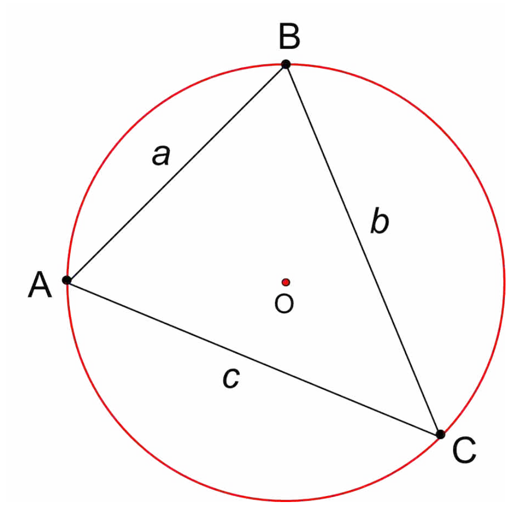


Figure 2.6: Three points (A, B, C) in a vortex core line circumscribed by a circle.

points, and O is the center of the circle. The radius of the circle is calculated using Eq. 2.15 [41]. Curvature is then calculated as the reciprocal of the circle radius , as shown in Eq. 2.16.

$$R = \frac{abc}{\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}} \tag{2.15}$$

$$C = \frac{1}{R} \tag{2.16}$$

2.4 Unsteady Vortex Extraction

As stated before, the RP and SH algorithms are both Galilean variant, and many modifications have been proposed to allow for extraction of moving vortex core lines from time-dependent CFD data sets.

2.4.1 Parallel Vectors Modifications

The parallel vectors operator is a widely used method for extracting vortices in steady-state simulations, and modifications to this method have been made by others to extend its usefulness to the unsteady domain. In the steady-state formulations, the unsteadiness of a vector field over time was ignored, which demands a modification for unsteady flow.

Time Derivatives

Fuchs et al. [42] created a simple derivative-based modification for algorithms which use derivatives of a time-dependent vector field. For unsteady vector fields which are implemented in the PV operator, one may merely calculate the proper time derivatives and include the derivatives in the vector field to find vortex cores. This modification essentially shifts the focus from streamline topology to path line topology. The SH algorithm [30] calculates the locations where velocity and acceleration are parallel: $\boldsymbol{u} \parallel \boldsymbol{a}$. The material derivative of the velocity field is the acceleration, as shown in Eq. 2.10. Similarly, the RP algorithm [33] calculates the locations where the velocity and jerk are parallel: $\boldsymbol{u} \parallel \boldsymbol{b}$. The jerk is the second material derivative of the velocity field, as seen in Eq. 2.12. In the steady state algorithms, the partial derivatives with respect to time were removed, and so by calculating these time derivatives, the unsteady nature of the flow field can be taken into consideration.

Fuchs et al. showed their success at more correctly extracting unsteady vortices using the SH and RP algorithms. They demonstrated that the vortex core lines extracted with the influence of time derivatives were shifted closer to a local pressure minimum and were more spatially accurate. They also investigated the impact of time step width on derivative calculations and showed that the amount of time steps between saved data sets has a large impact on the correctness and completeness of the extracted vortex core lines.

Schindler et al. [43] also used temporal derivatives to extract features and applied the method to Smoothed Particle Hydrodynamics data sets. Though their approach was somewhat different because of the special nature of their data, the general method was the same. They found that inclusion of time derivatives worked well in data sets in which there was smaller changes between time steps. To account for data sets with a high amount of change between time steps, they suggested the use of higher-order interpolation methods to calculate time derivatives.

Scale-Space

The theory of scale-space and feature-based methods can also be used to extract and track vortices in unsteady flow. Bauer and Peikert [13] proposed a method to apply Gaussian smoothing to the data set, which also simplifies the calculation of time derivatives. They then select a proper scale to extract relevant features from the flow. The extracted features are then brought down to a new scale to track them over time. Their method involves 5 dimensions – 3 spatial, 1 temporal, and 1 scale. The method is attractive because it combines feature extraction and tracking in one algorithm and allows for the use of the PV operator. However, the idea of scale-space is quite complicated and involves such computations as solving a Gaussian scalar field convolution, assigning hypercube vertices, and calculating element stiffness matrices.

Feature Flow Fields

Feature Flow Fields ($\boldsymbol{f}$) have been created as a method to "represent the dynamics behavior of features as the streamlines of a higher dimensional vector field" [44]. More simply put, vortex core lines extracted by RP and SH are streamlines of $\boldsymbol{f}$. This method was also treated by Theisel et al. [45]. Streamline integration is well understood, and once $\boldsymbol{f}$ is obtained, it can be integrated to extract vortex cores. The calculation for $\boldsymbol{f}$ in a 3D vector field is as follows:

$$\boldsymbol{f}(x,y,z,t) = \begin{pmatrix} +det(\boldsymbol{u}_y, \boldsymbol{u}_z, \boldsymbol{u}_t) \\ -det(\boldsymbol{u}_z, \boldsymbol{u}_t, \boldsymbol{u}_x) \\ +det(\boldsymbol{u}_t, \boldsymbol{u}_x, \boldsymbol{u}_y) \\ -det(\boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_z) \end{pmatrix} \tag{2.17}$$

From Eq. 2.17, it is clear that use of $\boldsymbol{f}$ also requires time derivatives. The advantage of using the feature flow field method over merely calculating time derivatives is that $\boldsymbol{f}$ can also be used to track features over time, thus eliminating the need to find a separate feature tracking method. Another requirement to find $\boldsymbol{f}$ is that vortex cores must be extracted at t_{min} and t_{max} in order to extract all lines in between and track them.

Weinkauf et al. [12] presented a method similar to the PV operator, which they called the Coplanar Vectors operator and used the feature flow field to extract vortex core lines. In unsteady flows, path lines are calculated as follows:

$$\boldsymbol{p}(x,y,z,t) = \begin{pmatrix} \boldsymbol{v}(x,y,z,t) \\ 1 \end{pmatrix} = \begin{pmatrix} u(x,y,z,t) \\ v(x,y,z,t) \\ w(x,y,z,t) \\ 1 \end{pmatrix} \tag{2.18}$$

The Jacobian of $\boldsymbol{p}$ is

$$\boldsymbol{J}(\boldsymbol{p}) = \begin{bmatrix} u_x & u_y & u_z & u_t \\ v_x & v_y & v_z & v_t \\ w_x & w_y & w_z & w_t \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.19}$$

and has the 4 eigenvectors

$$\begin{pmatrix} \boldsymbol{e}_1 \\ 0 \end{pmatrix}, \begin{pmatrix} \boldsymbol{e}_2 \\ 0 \end{pmatrix}, \begin{pmatrix} \boldsymbol{e}_3 \\ 0 \end{pmatrix} =: \boldsymbol{e}^s, \boldsymbol{f} \tag{2.20}$$

where s denotes the steady-state eigenvectors. With these vectors calculated, Weinkauf et al. stated that cores of swirling particle motion occur when $\boldsymbol{p}$, $\boldsymbol{e}^s$, and $\boldsymbol{f}$ are coplanar. After some manipulation, they come up with the following:

$$\lambda_2 \underbrace{\begin{pmatrix} e^s_1 \\ e^s_2 \\ e^s_3 \end{pmatrix}}_{\boldsymbol{a}} + \lambda_3 \underbrace{\left(\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} - f_4 \begin{pmatrix} u \\ v \\ w \end{pmatrix} \right)}_{\boldsymbol{b}} = 0 \tag{2.21}$$

The vectors are thus coplanar when $\boldsymbol{a} \parallel \boldsymbol{b}$. With this method, they extracted critical points and vortex cores from the 2D unsteady cavity flow problem. They showed that the vortex cores were extracted in the center of swirling particle motion instead of swirling streamline motion, which is more correct in unsteady flow data sets.

2.4.2 Alternative Methods

Though the PV operator is a strong and robust method for extracting vortex core lines, other researchers have created significantly different methods which are also viable for use. Jiang et al. [46] recommended that only Galilean invariant extraction algorithms be used in order to avoid the difficulty knowing the vortex core reference velocity a priori, which negates the use of the RP and SH algorithms in their view. Some of the following methods are Galilean invariant and provide a different method to account for unsteady flows.

Potential Flow

Peikert et al. [47] presented a method that extracts vortices from time-dependent flows by measuring the deviation of an actual flow from potential flow, which they called "localized flow". They used a Helmholtz-Hodge decomposition to find a potential flow that shared the same boundary conditions as the actual flow. This allowed them to extract vortices from unsteady flows without using a moving frame of reference. The main application they gave of this method was in situations where the main flow direction was not constant.

"Trigger" Method

Marusic et al. [48] presented a very different method which extracts features of interest by use of "triggers." Instead of extracting features from the entire data set, these triggers are used to write the data in key regions of interest to disk. Though it reduces the required storage space, the regions of interest or the triggers must be known a priori and require user interaction to find the correct triggers. After the determination of the triggers, they can be updated to write only the regions where features have been previously extracted, thus reducing future computational demand.

Lagrangian Methods

In order for Galilean invariance to be satisfied, several authors provided Lagrangian methods which investigate the motion of all particles in the flow instead of the Eulerian view. Fuchs et al. [49] investigated critical points and Lagrangian flow topology in the unsteady domain and created a measure for unsteadiness of the flow. This unsteadiness measurement describes the rate of change of the velocities of the fluid element over time. This method did not produce line features but rather provided a view of the flow field and its critical points. The importance of critical points is that vortices swirl around these critical points. Kasten et al. [50] proposed a method for extracting Lagrangian equilibrium points that exist for multiple time steps, since these are where the most important features are located. In both of these methods, time derivatives were also involved and the output was quite dissimilar from the SH and RP algorithms due to the inclusion of critical points and saddle regions.

Path/Streak Line Methods

Several authors employ path line attributes to extract vortices from complicated flows. Fuchs et al. [51] integrated path lines through a data set with user-defined integration lengths to extract regions of swirl and correlated it using the λ_2 method. This algorithm required sustained user interaction as integration line lengths became incorrect. Shi et al. [52] similarly integrated path lines, but they computed many different attributes of the path lines that they felt were important in extracting regions of swirling flow. Again, this method was very interactive and included selection of regions of interest from charts of these path line attributes. While these methods lent to an understanding of swirling regions, they do not apply well to automatic extraction and tracking of vortex cores in a subjective logic framework.

Weinkauf and Theisel [53] extracted vortex core lines based on streaklines by creation of a streakline vector field. This vector field was created through dense path line integration of the flow, which is computationally expensive. They showed that the core lines obtained from the streakline vector field were more accurate than those obtained by streamline and path line methods. This method is also Galilean invariant, and as in all other techniques which employ path line integration, multiple time steps or the entire data set are necessary to find the paths of particles over time.

2.5 Feature Tracking

Feature extraction alone in time-dependent flows often provides insufficient information about the temporal evolution and interactions of features. For this reason, feature tracking has been researched and implemented along with feature extraction in order to follow salient features over time. Researchers have approached this problem using techniques from image processing, feature extraction, and fluid mechanics. The problem is not trivial and many different approaches have been used over the years.

Feature tracking methods can be generalized into two main categories: tracking as a post-processing step to extraction, and tracking concurrent with and often as a step of feature extraction. Post et al. [5] reviewed a number of the state-of-the-art feature tracking methods at that time but made no conclusions as to the superiority of any method. In the post-processing method, features are first extracted from all considered time steps, then feature tracking is performed to find the features which best correspond to each other throughout all time steps. The concurrent, or co-processing, method employs higher-dimensional vector fields or isosurfaces to abstract the 3-dimensional features through time and is often used to extract and track features in the same step.

Event detection is another aspect of feature tracking which has received much consideration. As features move and evolve, it is important to understand how they interact and affect the simulation. Samtaney et al. [54] were among the first to classify important feature events, which are as follows: continuation, creation/dissipation, entry/exit, and amalgamation/bifurcation. These events can be visualized in Figure 2.7. Creation and dissipation refer to the birth or death of a feature in a certain time step, respectively. Entry and exit refers to the case when a feature enters or leaves the computational domain boundaries. Amalgamation refers to the event in which two or more separate features in one time step merge into one feature in the next time step, while bifurcation describes the opposite case of one feature splitting into multiple features. Different authors have used various names but essentially look for the same events.

2.5.1 Post-Processing Methods

Post-processing methods include tracking algorithms which have been designed to work after features have been extracted from all time steps. Many of these methods have been borrowed

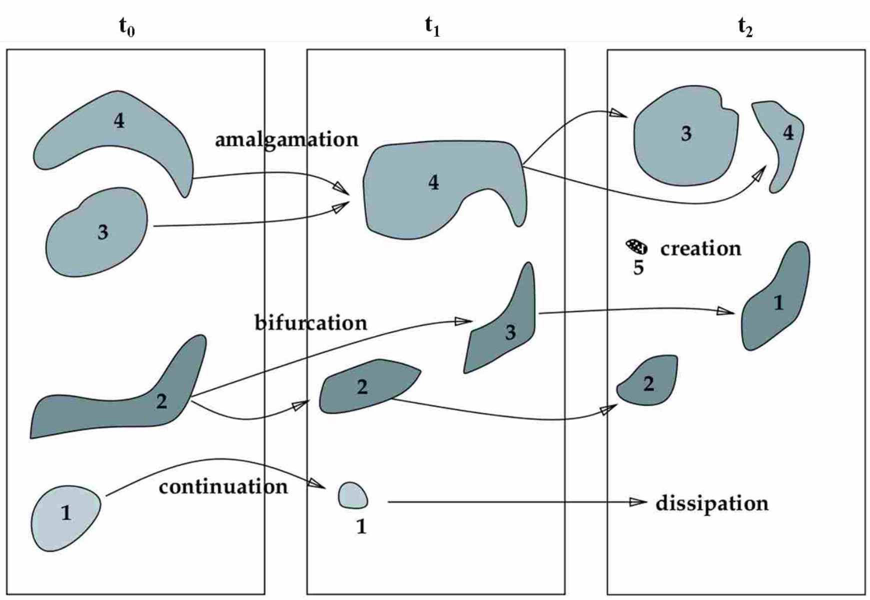


Figure 2.7: Feature events as defined by Samtaney et al. Image from [54].

from other scientific fields such as medical imaging and surveillance. The two main methods in this category include region-based methods, which match feature regions, and attribute-based methods, which correlate calculated attributes of the features in different time steps. Post-processing methods have the advantage of speed because they operate on already extracted features, though they must solve the difficult "correspondence problem" – the issue of matching a set of features in different time steps – by exhaustive search or some other search method.

Region-Based Methods

Region-based methods are some of the earliest feature tracking methods, and they operate by matching feature regions in successive time steps. This is mainly done by either measuring the distance between features or by using spatial overlap. Kalivas et al. [10] used a 2D linear affine transformation matrix to correlate the movement of 3D objects. Spatial overlap indicates that features overlap in successive time steps, and the assumption that the sampling frequency is high enough for this to be the case has been made by numerous authors [54–58]. Often these methods correlate incorrect features because of the overlap assumption, though corrections can be made which also correlate feature volume.

A novel region-based method was created which tracks vortex core lines. Schafhitzel et al. [59, 60] tracked vortex core lines using a pathline predictor-corrector method. They started with extracted core lines at the first time step and seeded particles along the pathlines. At the next time step, they found the new locations of the particles and correlated the particle locations to core lines in the time step. If enough particles from a single core line at t_i fell within the vortex region (specified by the λ_2 criterion) at t_{i+1}, the two cores were matched. The authors had difficulty when dealing with events such as split and merge, though they created a method to detect birth/death and entry/exit events. This method requires both a core line and a region in order to correlate particles and core positions.

Region-based feature tracking is also used in a unique way with Particle Image Velocimetry (PIV) [61]. In PIV, a laser sheet illuminates a section of an experimental flow field with particles seeded in the fluid. Image pairs are cross-correlated in order to predict and calculate the displacement of particles in the flow. With each particle tracked between an image pair, a velocity field can then be constructed from the images, and other flow variables such as strain and vorticity can then be calculated.

Region correspondence has certain strengths and weaknesses, especially in the context of vortex tracking. Event detection can be handled using region-based methods, especially amalgamation and bifurcation, though it is not a clear focus of these methods. These methods are quite simple to code and use as a post-processing step, but the most pressing concern is that these are *region*-based methods, while vortex extractions are line features. While some of the aspects of region correspondence may be applied to line features, spatial overlap is infeasible, and the minimum distance method may not work well in data sets with closely packed features.

Attribute-Based Methods

Attribute correspondence refers to the method of tracking features by use of calculated attributes such as position, size, volume, and orientation. In the case of vortex tracking, volume may appropriately be replaced by length. This correspondence method works well with event detection, since the attributes of split or merged features are the sum of the original features in the previous time step. This method also involves multiple passes to correlate features and detect events.

Samtaney et al. [54] pioneered the use of attribute correspondence in CFD and employed such methods as distance minimization and search algorithms to track features. This method assumed that the sampling frequency was high enough that neighborhood thresholding could be used to track features and reduce the amount of feature comparisons between time steps. They also employed a search octree to remove features as soon as they had been attached to a tracked feature. Last, they placed high importance on distance minimization to correspond features, which also assumes a high sampling frequency.

Reinders et al. [11, 62, 63] used attribute correspondence and a predictor-corrector method to track different features over time. By using the attributes from the previous time step, they were able to predict the future movement of a feature and match it to a feature in the current time step. After a feature tracking path was created, they linearly extrapolated feature attributes to predict and match the feature in the next time step. They also created a graph viewer to aid in the visualization of tracked features.

Silver et al. [55,56] focused on turbulent data sets and feature tracking in a parallel environment. They employed spatial overlap as a main requirement of feature correspondence and used octree forests to detect events and differentiate between tracked features. Chen et al. [56] focused on the application of feature tracking over a distributed network, which is extremely important when tracking features in massive data sets. They accomplished this with local merging of features and exhaustive search to find the best match between features across processor boundaries. While the idea of feature tracking in a parallel environment is necessary for massive data sets, the concept of exhaustive search seems infeasible for data sets which contain many features and could be quite time-consuming.

2.5.2 Co-Processing Methods

In contrast with post-processing methods, certain authors have created algorithms which extract and track features in the same step, where tracking is often used as a step of feature extraction. These methods generally employ higher-dimensional objects or vector fields and track features through the time axis instead of using feature attributes or spatial overlap to correspond previously extracted features. The methods used here range from imaging techniques to fluid dynamics principles, and each has certain strengths and weaknesses.

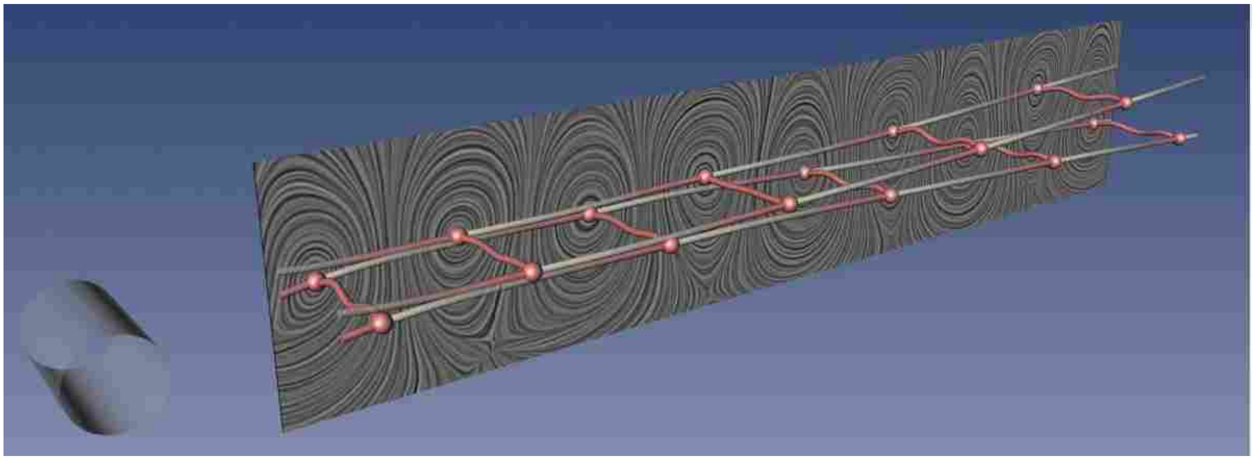


Figure 2.8: Vortex core lines extracted and tracked from the 3D cylinder data set using a feature flow field. Grey paths indicates future movement, and red paths indicates past movement. Image by Tino Weinkauf [45].

Feature Flow Fields

Feature Flow Fields $\boldsymbol{f}$ were discussed in Section 2.4.1 as a feature extraction method, since extraction and tracking usually occur together in this method. The motivation behind the use of $\boldsymbol{f}$ for feature tracking was that streamline integration is a well-known method in CFD, and by integration of the streamlines of $\boldsymbol{f}$, the path of features may be found through time.

Feature flow fields have been applied to the Parallel Vectors (PV) formulation of feature extraction algorithms with good success. This was applied to a 3D cylinder data set, as seen in Figure 2.8. It can easily be seen that through time, the vortex core is lifted from a 2 dimensional line to a 3 dimensional surface. Visualization of feature movement was accomplished by coloring future and previous movement with translucent grey and red paths, respectively.

Event detection is also handled by $\boldsymbol{f}$. Birth and death events can be visualized as a closed loop at some time step t_i, and split/merge events were also classified by the authors. The visualization of feature events was more abstract than that of the attribute-based methods.

The feature flow field approach has some limitations which are important to discuss. Some formulations of $\boldsymbol{f}$ do not guarantee streamlines which always converge on the features, though Weinkauf et al. [53] discussed a correction factor which may guarantee convergence on features. Another limitation of $\boldsymbol{f}$ is that, like streamlines, it requires proper seeding points to capture all of the features of interest. This requires the extraction of features at key time steps and integrating streamlines of $\boldsymbol{f}$ from points on these features. Thus, the use of $\boldsymbol{f}$ for tracking as a simulation runs is infeasible.

Scale Space

Another method which performs tracking concurrent to feature extraction is the scale space method, which allows one to track a feature through scale and time using imaging methods. Bauer et al. [13, 64] applied this method to feature extraction and tracking of vortex core lines and used the PV algorithm of Roth and Peikert. They used the Marching Cubes algorithm to search the data set on a cell-by-cell basis, then constructed a hypercube from the data at t_i and t_{i+1}. A hypercube is fundamentally a cube in 4 dimensions where each of the vertices of the hypercube is a cell boundary at one of the time steps. With this information, the authors found the sets of points of a hypercube where the vector fields were parallel.

After construction of the hypercube, a feature mesh was created. Vortex cores were added to the feature mesh when the Parallel Vectors algorithms was satisfied. Event detection was not implemented in scale space methods, but the authors theorized that such would be possible using feature mesh attributes. Birth/death events would appear as sharp points of the feature mesh, while split and merge events would be characterized by a separation or reconnection of the mesh, respectively.

Scale space feature tracking is computationally expensive and requires the implementation of some fairly complex algorithms. It has the advantage of working with Parallel Vectors algorithms, but was not proven in 3 dimensional CFD data sets.

Other Methods

Tzeng et al. [57] used adaptive transfer functions to predict and track features in large-scale 4D simulations. Their approach was a region-based one, which is not appealing for vortex core tracking. Also, their method was quite complex, and they used such techniques as neural networks, support vector machines, and interactive machine learning to accomplish the task.

Muelder et al. [58] also used a region-based interactive method, which may be applied to line-type features. In the first time step, they extracted features and used a predictor-corrector method to extract and track features concurrently. As they made a prediction in a subsequent time step, they would search the neighborhood for cells which satisfied the feature extraction criterion. By use of feature region growing/shrinking they would match the region in t_{i-1} to the region in t_i. This eliminated the need to correspond features later, as they correlated features as they extracted them. This method also lends well to interactive and runtime simulations, but requires modification for non-region features such as vortex core lines.

2.6 Subjective Logic

As stated in Section 1.4, subjective logic incorporates four basic elements: belief (b), disbelief (d), uncertainty (u), and atomicity (a). In this research the assumption of $a = 0.5$ is used, which denotes that equal weight is given to each agent. This assumption was made so that the method could be used generally in any CFD data set. To maintain uniformity and provide for mathematical constructs, the summation of an opinion's components, also called the belief tuple, is always equal to unity as displayed in Eq. 2.22.

$$b + d + u = 1 \tag{2.22}$$

Furthermore, belief, disbelief, and uncertainty can only take on values between 0 and 1. These basic prerequisites provide much of the framework necessary for working with opinions in a mathematically rigorous fashion.

2.6.1 Opinion Triangle

An opinion may be visualized by use of a triangle due to the formulation of Eq. 2.22. Figure 2.9 shows an example of an opinion of $\omega_x = (b,d,u,a) = (0.4, 0.1, 0.5, 0.6)$ visualized on the opinion triangle. The opinion can be located by following two of the three arrows located at the midpoint of each triangle side from the side opposite the arrowhead. The dotted lines which are perpendicular to each arrow delineate each value by a width of 0.1. For example, ω_x may be found by first traveling 0.4 steps on the belief arrow. Next, follow the dotted line from the 0.1 value of

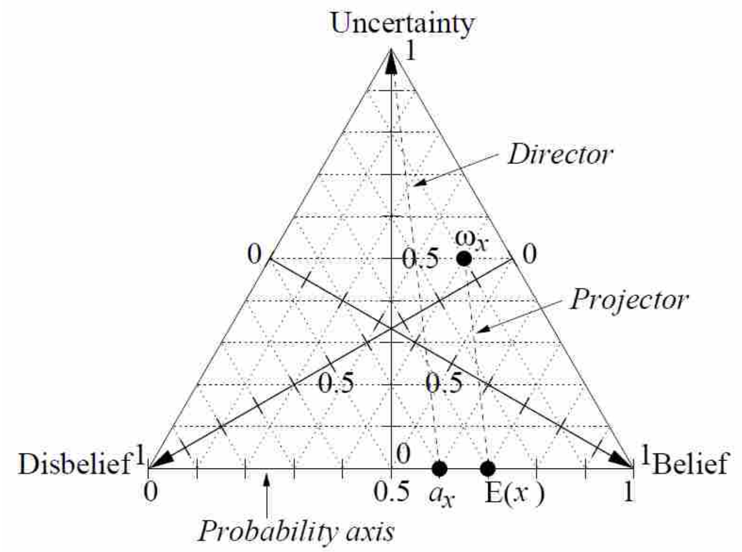


Figure 2.9: A subjective logic triangle with $\omega_x = (0.4, 0.1, 0.5, 0.6)$ as an example. Image by Audun Josang [14].

the disbelief line and find where it intersects the dotted line from the 0.4 value of the belief. The uncertainty value may also be substituted to find the location of the opinion in the opinion triangle.

2.6.2 Probability Expectation

Subjective logic attempts to remove strict notions of TRUE and FALSE. Thus, instead of specifically stating if a feature is present, the opinion of a detected CFD feature can express if that feature has a high expected probability of occurring. When evaluating an opinion, probability expectation (E) gives the expected probability of an outcome based on the opinion and can be calculated using Eq. 2.23:

$$E = b + au \tag{2.23}$$

It takes the entire opinion into account and incorporates the atomicity base rate proportionally to the uncertainty. Uncertainty is taken into account because it is a measure of the unknowns in an outcome and the atomicity is the expected outcome in the absence of any additional information. Due to the assumption that $a = 0.5$, The probability expectation reduces to

$$E = b + \frac{1}{2}u \tag{2.24}$$

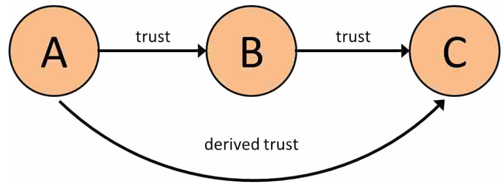


Figure 2.10: Simple trust network showing A's derived trust in C from B.

The probability expectation identifies what an agent expects the probability to be and is not an exact measure of probability. However, mappings also exist which allow subjective logic opinions to be expressed as probabilistic distributions [14].

The opinion triangle lends to a clearer understanding of the effect of changing atomicity. As seen in Figure 2.9, with an atomicity of $a = 0.6$, the line connecting the opinion value to the probability axis, the projector, is parallel to the director, thus denoting that a weight is given to the belief of the feature. This results in a probability expectation of $E = 0.7$ Given an atomicity of 0.5, the probability expectation in this example would be closer to 0.65, which proves visually that as one increases atomicity of an agent, the probability expectation also increases.

2.7 Trust Networks

A means of combining output from multiple feature extraction algorithms into a single coherent feature set was needed. Intelligent software agents designed in the form of a trust network accomplish this task. Trust networks [18] are a way to quantify trust that is transferred from one individual to another. For example, Figure 2.10 shows a simple trust network where individual A has trust in individual B, but does not know individual C. Individual B trusts individual C and can then "refer" individual C to individual A, thus giving individual A derived trust in individual C. In the trust network individuals are called 'agents' and the means by which trust is quantitatively transferred between agents is subjective logic.

2.7.1 Discounting Operator

In a trust network there are two critical operators that transfer trust: the discounting operator and the consensus operator. The discounting operator ($\otimes$) is used when agents in a trust network lie along the same path as in Figure 2.10. In this situation, B has formed some opinion of C which is unknown to A. For A to formulate an opinion of C, A discounts B's opinion $(A \otimes B)$ deriving trust of C based on A's opinion of B and B's opinion of C. The discounting operator is associative but not commutative. The opinion of A in C is shown by

$$\omega_C^A = \omega_B^A \otimes \omega_C^B \tag{2.25}$$

where the superscripts represent an agent having the trust and the subscripts represent an agent, or piece of information, on which the trust is based. For example, ω_B^A represents the trust that A has in B. To compute the opinion of A in C, Eqs. 2.26-2.28 are used.

$$b_C^A = b_B^A b_C^B \tag{2.26}$$

$$d_C^A = b_B^A d_C^B \tag{2.27}$$

$$u_C^A = d_B^A + u_B^A + b_B^A u_C^B \tag{2.28}$$

2.7.2 Consensus Operator

The consensus operator is used to create an opinion reflecting two opinions in a fair and equal way. Different observations can create different opinions of the same event with independent values of belief, disbelief, and uncertainty. An important aspect of the trust network is being able to combine multiple opinions of the same event. The consensus operator is able to combine opinions with the effect of reducing uncertainty (belief and disbelief of the opinions proportionally aggregate while uncertainty decreases). The consensus operator is represented by the symbol $\oplus$ and is given by

$$\omega_Z^{XY} = \omega_Z^X \oplus \omega_Z^Y \tag{2.29}$$

To compute the opinion using the consensus operator, the following equations are used to find belief, disbelief, and uncertainty.

$$
for \kappa \neq 0 \quad
\begin{aligned}
b_Z^{XY} &= \left(b_Z^X u_Z^Y + b_Z^Y u_Z^X\right) / \kappa && (2.30)\\
d_Z^{XY} &= \left(d_Z^X u_Z^Y + d_Z^Y u_Z^X\right) / \kappa && (2.31)\\
u_Z^{XY} &= \left(u_Z^X u_Z^Y\right) / \kappa && (2.32)
\end{aligned}
$$

$$
for \kappa = 0 \quad
\begin{aligned}
b_Z^{XY} &= \frac{\gamma b_Z^X + b_Z^Y}{\gamma + 1} && (2.33)\\
d_Z^{XY} &= \frac{\gamma d_Z^X + d_Z^Y}{\gamma + 1} && (2.34)\\
u_Z^{XY} &= 0 && (2.35)
\end{aligned}
$$

where

$$\kappa = u_Z^X + u_Z^Y - u_Z^X u_Z^Y \quad (2.36)$$

and

$$\gamma = \frac{u_Z^Y}{u_Z^X} \quad (2.37)$$

2.8 Steady-State Trust Network

Mortensen [17] created a trust network and used it to extract features from steady-state CFD data sets. A graphical representation of the CFD trust network is shown in Figure 2.11. The algorithm agent AA contains actual feature extraction algorithms with subscripts 1 and 2 denoting separate algorithms. The master agent MA combines information from multiple AA's to form its opinion. The MA can be thought of as the governing, or controlling, agent. It has the most influence on the believability of extracted features. Its job is to synthesize information from multiple AA's and provide a final decision on the extracted features. R refers to a grid point contained in the extracted feature under inspection by the agents to find whether or not the feature is probable. The end goal is for the MA to form an opinion on R, meaning that the MA will have some belief, disbelief, and uncertainty about the feature contained in R.

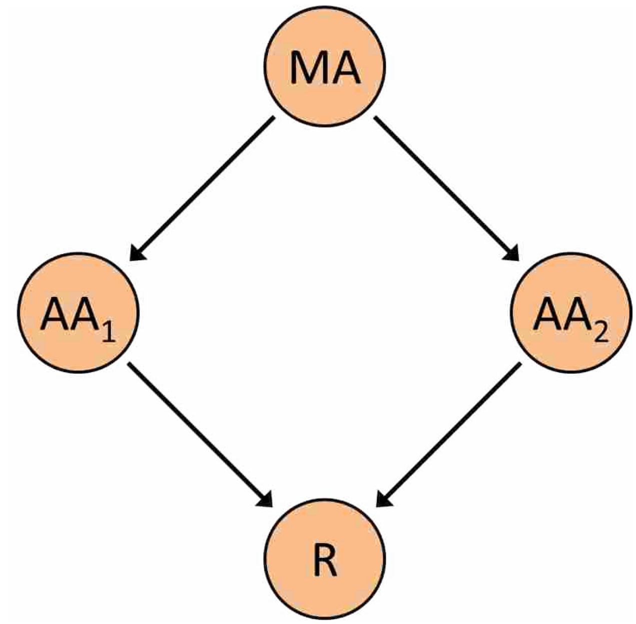


Figure 2.11: Graphical representation of two algorithm trust network.

Once features have been extracted and sent through simple filters, agents can begin to form opinions on extracted features. When agents form their opinions it means that a belief, disbelief, and uncertainty value is defined within an agent opinion adhering to Eq. 2.22. Agents form their opinions based on a user-defined set of information known to influence the extraction of the feature.

The belief tuple is defined as follows: belief is set by extraction algorithm strengths, disbelief is set by extraction algorithm weaknesses, and uncertainty is set by flow feature characteristics. Belief corresponds to the strengths of the algorithm matching with the conditions where the feature was detected. Disbelief is set similar to belief except the weaknesses, or situations where a feature extraction algorithm may spuriously extract a feature, govern the value. The weakness characteristics may be the exact opposite of the strength characteristics. Uncertainty is set from scientifically known characteristics of the flow feature which provide a measure of the unknowns in an outcome. Some of the unknowns may positively affect an outcome while some may negatively affect an outcome.

Mortensen created and outlined first-order equations defining belief, disbelief, and uncertainty equations. These equations were based on the following parameters: vortex strength, curvature, and quality (the angle between the vortex core and velocity vector). He also made the assumption that in steady-state data sets, a converged data set contains features that do not move between iterations and based the MA opinion on this assumption. This method was shown to be a useful tool for visualizing features concurrent to a running simulation and gauging convergence in steady-state data sets. The architecture of his method is modified and used to operate on time-dependent CFD data sets.

CHAPTER 3. VORTEX CORE EXTRACTION & TRACKING METHOD

This chapter outlines the modifications made to the steady state algorithms to account for unsteady flow. The feature tracking method is also described. Chapter 4 will outline in greater detail the opinion calculations and subjective logic methods.

The steps which describe this method are as follows:

1. Extract vortex core lines from the CFD data set using unsteady feature extraction algorithms.
2. Track extracted vortex cores through time.
3. Create agent opinions for each vortex core line.
4. Combine agent opinions to form final opinions of vortex core lines.
5. Aggregate believable vortex cores from separate data sets into one final feature set.

3.1 Transient Vortex Extraction

The steady state feature extraction algorithms were modified in order to correctly extract vortices from unsteady simulations. The time derivative method of Fuchs et al. [42] as discussed in Section 2.4.1 was implemented to modify existing steady-state vortex extraction algorithms. This method was utilized because of its relatively simplicity and success in other flow fields as shown by Fuchs et al. As stated before, the SH algorithm employs acceleration, the material derivative of velocity, which was shown in Eq. 2.10. Similarly, the RP algorithm calculates the jerk, which is the second material derivative of the velocity field, as seen in Eq. 2.12. In the steady state algorithms, the partial derivatives with respect to time were neglected, so by calculating the time derivatives in the Eqs. 2.10 and 2.12, the unsteady nature of the flow can be taken into consideration.

Derivatives were calculated in a manner that minimized numerical error without requiring an excessive amount of memory. The partial derivatives with respect to time in Eqs. 2.10 and 2.12

were computed using a central-differenced Taylor series approximation. These approximations may be seen in Eqs. 3.1 and 3.2.

$$\frac{\partial \boldsymbol{u}}{\partial t} = \frac{\boldsymbol{u}_{i+1} - \boldsymbol{u}_{i-1}}{2\Delta t} + O(\Delta t^2) \tag{3.1}$$

$$\frac{\partial^2 \boldsymbol{u}}{\partial t^2} = \frac{\boldsymbol{u}_{i+1} - 2\boldsymbol{u}_i + \boldsymbol{u}_{i-1}}{\Delta t^2} + O(\Delta t^2) \tag{3.2}$$

Both derivative approximations are second order accurate. This required storage of 3 time steps in memory, but the computational cost was necessary in order to minimize numerical error.

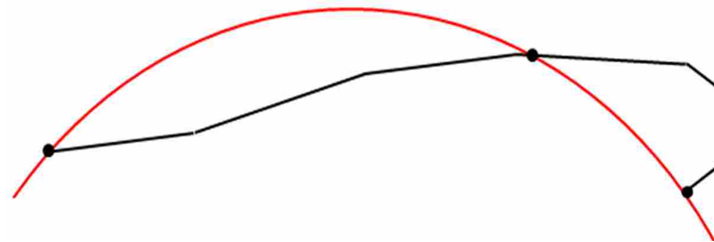
Central-differenced time derivatives require information from the previous and future time steps and thus vortex core lines were not extracted from the first and last time steps in consideration. For example, if 50 time steps of the CFD data set were written out and extraction were to be performed on the data set, features would be extracted from only 48 of the data files. For the purposes of this tool, it was felt that discarding two time steps was more prudent than using forward- and backward-difference approximations for the first and last time steps, respectively, and risk the numerical errors associated with these first-order approximations.

3.2 Modifications to Vortex Core Line Characteristics

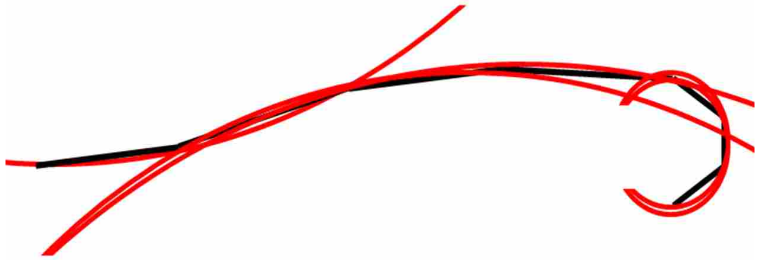
The vortex core attributes used to compute the opinion are strength, curvature, and quality. Mortensen [17] created methods for calculating these characteristics in steady state vortex core extraction. Because vortex strength is computed using the velocity gradient tensor $\nabla \boldsymbol{u}$, it is a Galilean invariant quantity and thus requires no modification in unsteady vortex core extraction. However, curvature and quality calculations were modified in order to more correctly reflect the opinion of vortex cores extracted from unsteady data sets.

3.2.1 Curvature

In the steady state method, only one curvature value was calculated per core line, which was too rough an estimate to correctly reflect the local curvature of a line. Prior geometric curvature was calculated using the core endpoints and the midpoint. An example of this can be seen in Figure 3.1(a). As seen, the one-circle curvature approximation does not accurately describe the



(a) Vortex core line curvature approximated by one circle.



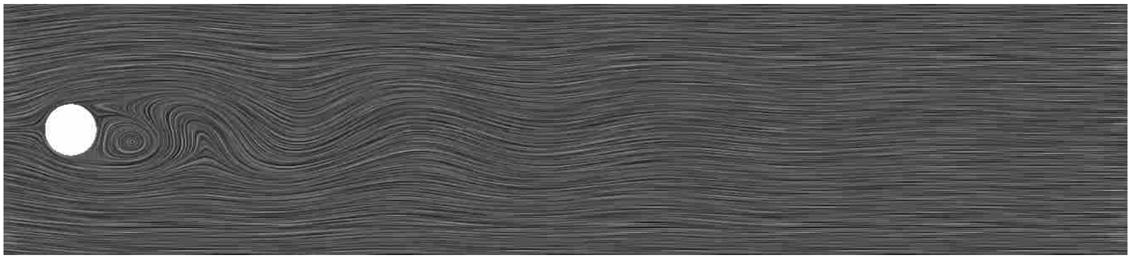
(b) Vortex core line curvature approximated by multiple circles.

Figure 3.1: Curvature approximation of a vortex core line (black segmented line) using circumscribed circles (red curves).

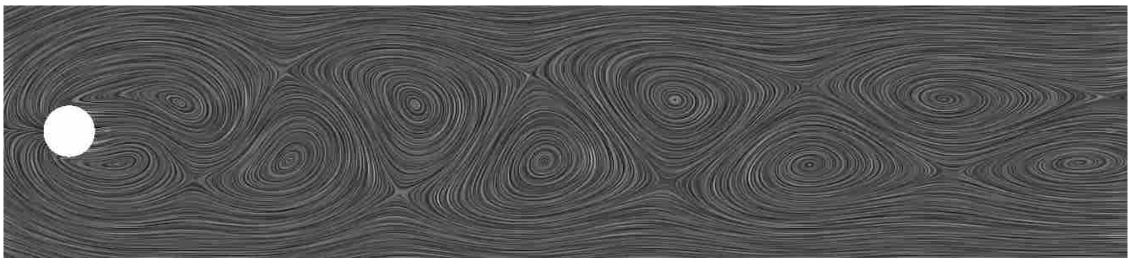
local curvature, especially in the hooked right end of the vortex core. To correct this problem, a local curvature may be calculated for each point by using the immediately adjacent points. The same vortex core with the local curvature approximation may be seen in Figure 3.1(b). Here it can be seen that the point-by-point method more closely approximates the local curvature of the line. At the endpoints of the vortex core line, there are not two adjacent points, which presents a problem to the local curvature calculation. To bypass this problem, the curvature of the point next to each endpoint is assumed to be the same at the endpoint. If desired, every 2^{nd} or greater point may be used in data sets with fine grids to capture higher curvature.

3.2.2 Quality

In steady-state simulations, quality was used successfully to threshold spurious cores and to determine the opinion of the remaining cores. However, when vortex cores move, as is the case in transient simulations, the velocity field often does not indicate swirling flow, and a proper convection velocity must be chosen in order to analyze the moving vortex core. One example of this problem may be seen in Figure 3.2, which was taken from case of a cylinder in cross flow,



(a) In the original frame of reference, swirling flow can only be seen near the cylinder.



(b) In a frame of reference moving with the vortex cores, the von Kármán vortex street may be clearly seen.

Figure 3.2: Line Integral Convolution (LIC) of a cylinder in cross flow (Section 5.2. Flow moves from left to right.

which is presented in Section 5.2. By subtracting a constant velocity field that corresponds to the vortex convection velocity, the swirling flow in the cylinder wake may be clearly seen. This then allows for the proper calculation of vortex quality. In order to select a proper convection velocity, the average velocity of each core line was calculated, which was then used as the convection velocity of the core line. The convection velocity of the core line was then subtracted from the velocity at the point and quality was calculated from the reduced point velocity. This individual treatment of each core line was a new method created for unsteady vortex extraction and allowed for separate line convection velocities.

3.3 Attribute-Based Vortex Core Tracking

Feature tracking is helpful to more fully understand the physics of unsteady flows and the complex feature interactions that occur. The attribute-based method created by Reinders et al. [11] was modified for use with vortex core lines. This method was used because of its robustness in applications where features behave predictably through time and because of its low computational cost.

3.3.1 Vortex Core Attributes

Reinders et al. created their tracking method based on the assumptions that features behave predictably between time steps and used certain feature attributes to correspond features. However, they created the feature tracking method for region-type features and utilized such attributes as volume, mass, orientation, and position. Since vortex core lines do not possess most of these attributes, other vortex core attributes were chosen for use in the tracking method.

The first three vortex core attributes were the same that were computed for use in subjective logic and were explained in Section 2.3: vortex strength, quality, and curvature. These values were computed at each point in the line, but a line-based attribute was needed as input in feature tracking. To utilize these attributes, the values of the attributes at each point in the line were averaged to obtain a line-based value for vortex strength, quality, and curvature.

The next two vortex core attributes were length and position, which relied more on the geometric properties of the line but were still considered valid parameters for use in feature tracking. As stated previously, each vortex core line consists of connected line segments, and an illustration of this may be seen in Figure 3.3. The total line length is then computed using Eq. 3.3. The position P of the vortex core line was a coordinate in 3-dimensional space and was approximated as the geometric center of the vortex core line bounding box. An example of the position approximation may be seen in Figure 3.4, where the position P of the vortex core is represented as a red point.

$$L = \sum_{i=1}^{n} l_i \tag{3.3}$$

In summary, five vortex core line attributes are used to compute feature correspondence: vortex strength, curvature, quality, length, and position. With these calculated, the task of feature tracking can then begin.

3.3.2 Calculating Feature Correspondence

Attribute functions were created for vortex core line attributes which are used to compare two lines that are contained in separate consecutive time steps. The attribute functions followed the format of Reinders et al. [11] and may be seen in Eqs. 3.4–3.8. In these equations, O_1 and O_2

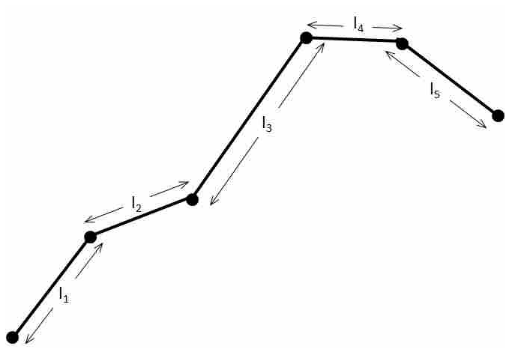


Figure 3.3: Vortex core line which is made up of several line segments. Line length is the sum of all segments that make up the line.

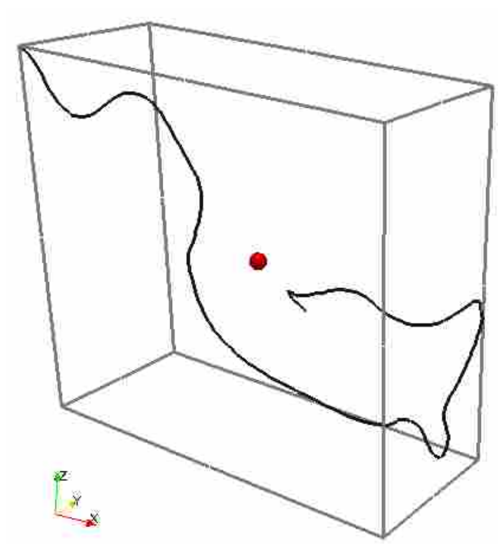


Figure 3.4: Computation of the position of a vortex core by placing a bounding box around the core line and finding the box's geometric center.

denote the two lines that are currently under comparison. With the exception of Eq. 3.8, each of the below equations results in a value between 0 and 1. The Euclidean distance between two points as calculated by Eq. 3.8 is strongly influenced by the size of the data set under consideration.

$$VortexStrength(O_1, O_2) = \frac{||S_1| - |S_2||}{max(|S_1|, |S_2|)} \tag{3.4}$$

$$Curvature(O_1, O_2) = \frac{|C_1 - C_2|}{max(C_1, C_2)} \tag{3.5}$$

$$Quality(O_1,O_2) = \frac{|Q_1 - Q_2|}{max\,(Q_1,Q_2)} \tag{3.6}$$

$$Length(O_1,O_2) = \frac{|L_1 - L_2|}{max\,(L_1,L_2)} \tag{3.7}$$

$$Position(O_1,O_2) = \|P_1 - P_2\| \tag{3.8}$$

Correspondence functions are then created from the attribute functions. The general form of a correspondence function may be seen in Eq. 3.9, where $func(O_1,O_2)$ corresponds to Eqs. 3.4–3.8. This formulation allow for values of C_{func} between $-\infty$ and 1, where 1 denotes a perfectly matched attribute, 0 denotes a barely matched attribute, and negative values indicate attribute matching of less than the tolerance T_{func}. T_{func} values are chosen based on the user's preference. For example, if one wished to match features which had attributes which were within 90% of each other, then a T_{func} of 0.1 would be chosen for all tolerances except position. As stated before, the position attribute function is very simulation-dependent and a specific position tolerance corresponding to the data set must be used. For example, in a simulation of flow past an airfoil, a position tolerance of 10% of the chord may be used, whereas in an atmospheric simulation, the position tolerance may be on the order of kilometers.

$$C_{func}(O_1,O_2) = 1 - \frac{func(O_1,O_2)}{T_{func}} \tag{3.9}$$

The overall feature correspondence parameter *Corr* is next computed and used to decide whether two features correspond. The correspondence parameter is computed according to Eq. 3.10. Here, weights are assigned to each correspondence function, which may be changed if one attribute is felt to be better suited for tracking vortex cores. For this research equal weight is given to each correspondence function. The correspondence parameter also has a range similar to each correspondence function, i.e. $-\infty \leq Corr(O_1,O_2) \leq 1.0$.

$$Corr(O_1,O_2) = \frac{\sum_{i=1}^{N_{func}} C_i(O_1,O_2)W_i}{\sum_{i=1}^{N_{func}} W_i} \tag{3.10}$$

Prediction of feature attributes may be made once a feature has been tracked for at least two time steps by use of linear extrapolation. A tracking path must first be initialized by using the given feature attributes for a line. When a tracking path has been made, attributes in the next time step may be predicted using Eq. 3.11.

$$P_{i+1} = O_i + \frac{t_{i+1} - t_i}{t_i - t_{i-1}} (O_i - O_{i-1}) \tag{3.11}$$

In the case of a constant time step, which is true for this research, Eq. 3.11 simplifies to the following:

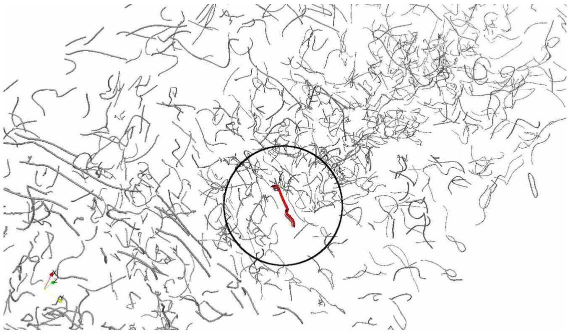
$$P_{i+1} = 2O_i - O_{i-1} \tag{3.12}$$

Eq. 3.12 is used to predict attributes of a vortex core in t_{i+1}, which are then used to compute Eqs. 3.4 through 3.10. Use of linear extrapolation assumes that features behave linearly between time steps, which is not usually the case, but it will generally be a better prediction than using feature attributes of a line in time t_i.

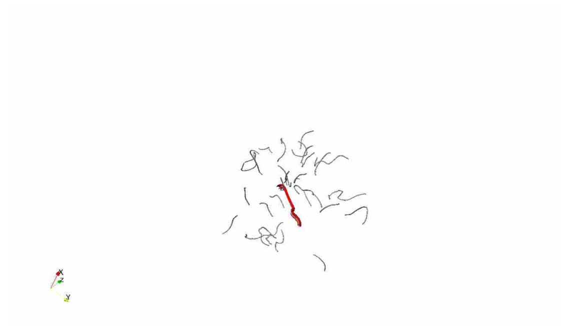
Feature tracking is accomplished by sweeping through the data set multiple times and relaxing the attribute tolerances on each sweep. Reinders at al. reported that the best success in tracking comes when strict tolerances are initially used to find the most obvious tracking paths. By performing forward and backward passes through the data set while gradually increasing the tracking tolerances T_{func}, less and less obvious tracking paths may be created and added upon. In this research, each successive pass results in a tolerance relaxation of 10% of the initial tracking tolerance.

3.3.3 Efficient Search Method

Attribute-based feature tracking was initially created as an exhaustive search method where each feature was compared to every other feature in the next time step. The exhaustive search method was improved upon by only considering untracked features in the next time step, but for massive CFD data sets with perhaps thousands of features, even the improved search method can be a prohibitively long process. A more efficient search method was created for this research in the form of a sphere of influence. A sphere with a radius equal to the length of the vortex core line is placed at the center of the vortex core bounding box. Any vortex cores in the next time step which are contained in this “sphere of influence” then become the candidate vortex cores against which



(a) The original data set showing a sphere placed around the vortex core under consideration (heavy red line).



(b) The reduced data set which shows the candidate vortex cores for feature tracking.

Figure 3.5: Example of efficient search method created to reduce the necessary number of vortex cores to compare against for feature tracking.

the current vortex core is compared. An example of this method may be seen in Figure 3.5, which was visualized from the wind turbine data set (Section 5.3). It can be seen that from a complex vortical data set, only a handful of vortex core lines are close enough in the next time step to be considered for feature tracking. A similar assumption of feature predictability was made as in the feature tracking method, i.e. the vortex core will not move drastically in between time steps.

3.3.4 Measuring Feature Lifetime

The lifetime of the feature, or the number of time steps in which it exists, is measured so that it may be used in the subjective logic formulation. As a new tracking path is created during the tracking process, a unique "tracking ID" is assigned to the new path. As new vortex cores are added onto a certain path, they also receive the tracking ID of the initial path. This is performed throughout the tracking process, with untracked features receiving a tracking ID of 0. After tracking has been performed throughout the entire data set, another pass is made to measure the lifetime of features. This is accomplished by creating an array the size of the number of unique tracking paths created. The feature lifetime of a feature within a certain tracking ID path is thus incremented by one as the same tracking ID is found in different time steps. After the lifetime measurement pass has completed, another pass is made through the data set to assign the measured feature lifetimes of all vortex cores. Vortex core lines with a tracking ID of 0 receive a feature lifetime of 1, since they existed one time step in the data set.

CHAPTER 4. FORMING OPINIONS ON VORTEX CORE LINES

This chapter outlines the method used to form opinions on vortex cores. Also, the method to aggregate believable features from separate algorithm outputs into one final feature set is presented.

4.1 Trust Network Setup

The trust network set up by Mortensen [17] was outlined in Chapter 2 and is now explained in greater detail. Figure 2.11 shows the agent-based trust network, which contains the Master Agent (MA), two algorithm agents (AA_1 and AA_2) and the region (R) which contains the feature. The final goal is to find the opinion of the MA in R – $\omega_{\mathrm{R}}^{\mathrm{MA}}$. To accomplish this, four belief tuples must be calculated: $\omega_{\mathrm{R}}^{\mathrm{AA}_1}, \omega_{\mathrm{R}}^{\mathrm{AA}_2}, \omega_{\mathrm{AA}_1}^{\mathrm{MA}}$ and $\omega_{\mathrm{AA}_2}^{\mathrm{MA}}$. The discounting operator ($\otimes$) is used to compute the MA opinion through each AA, and the consensus operator ($\oplus$) must be used to combined both linear opinions into $\omega_{\mathrm{R}}^{\mathrm{MA}}$. Eq. 4.1 show the use of the consensus and discounting operators to give the final opinion and Eqs. 4.2–4.4 give the belief tuple values in the final opinion for the realistic assumption of $\kappa \neq 0$.

$$\omega_{\mathrm{R}}^{\mathrm{MA}} = \left(\omega_{\mathrm{AA}_1}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA}_1}\right) \oplus \left(\omega_{\mathrm{AA}_2}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA}_2}\right) \tag{4.1}$$

$$b_{\mathrm{R}}^{\mathrm{MA}} = \frac{(b_{\mathrm{AA}_1}^{\mathrm{MA}} b_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) + (b_{\mathrm{AA}_2}^{\mathrm{MA}} b_{\mathrm{R}}^{\mathrm{AA}_2})(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})}{\kappa} \tag{4.2}$$

$$d_{\mathrm{R}}^{\mathrm{MA}} = \frac{(b_{\mathrm{AA}_1}^{\mathrm{MA}} d_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) + (b_{\mathrm{AA}_2}^{\mathrm{MA}} d_{\mathrm{R}}^{\mathrm{AA}_2})(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})}{\kappa} \tag{4.3}$$

$$u_{\mathrm{R}}^{\mathrm{MA}} = \frac{(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2})}{\kappa} \tag{4.4}$$

where

$$\begin{aligned} \kappa = &(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1}) + (d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) \\ &-(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) \end{aligned} \tag{4.5}$$

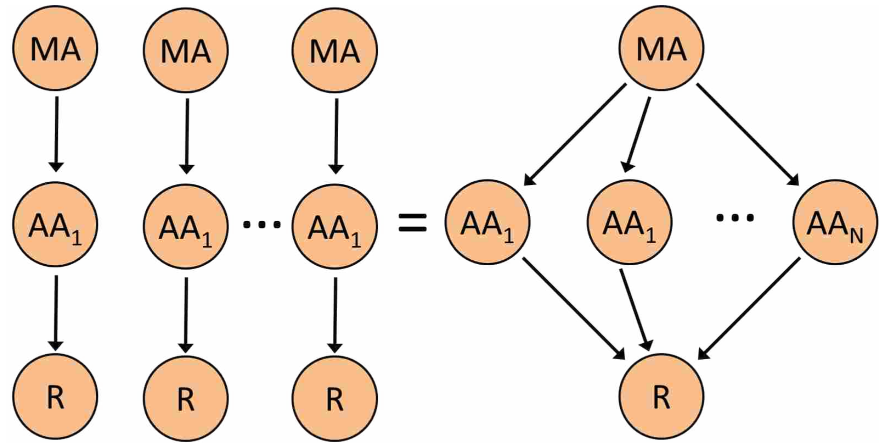


Figure 4.1: Graphical representation of modular agent structure.

It can be seen in Eqs. 4.2–4.4 the effect of the discounting and consensus operators as described in Section 2.7. For example, the two terms in the numerator of Eq. 4.2 are the effects of the discounting operator and the consensus operator is shown in the equation as the sum of the two terms divided by κ.

While only two AA's were used in this research to create the trust network, any number of AA's may be added to the trust network to add more feature extraction algorithms. Figure 4.1 shows how this is accomplished by simply adding linear paths along which the MA computes its opinion of R through AA_N. Adding and removing agents from this structure is quite simple, since the structure of the entire trust network is modularized. Eq. 4.6 shows the necessary extension from Eq. 4.1 to have N AA's in the trust network. The effect of adding AA's is to reduce uncertainty in the features extracted by the AA's.

$$\omega_{\text{R}}^{\text{MA}} = \left(\omega_{\text{AA}_1}^{\text{MA}} \otimes \omega_{\text{R}}^{\text{AA}_1}\right) \oplus \left(\omega_{\text{AA}_2}^{\text{MA}} \otimes \omega_{\text{R}}^{\text{AA}_2}\right) \oplus \cdots \oplus \left(\omega_{\text{AA}_\text{N}}^{\text{MA}} \otimes \omega_{\text{R}}^{\text{AA}_\text{N}}\right) \tag{4.6}$$

4.2 Algorithm Agent Opinions

The first step to compute the final opinion of a feature is to compute the agent opinions $\omega_{\text{R}}^{\text{AA}_1}$ and $\omega_{\text{R}}^{\text{AA}_2}$. Though each AA extracts features from the same data set, they extract different features and should be thought of as separate feature sets. This is illustrated in Figure 4.2, where

Figure 4.2: Two separate line-type features extracted by AA_1 (black) and AA_2 (red).

the black line was extracted by AA_1 and the red line was extracted by AA_2. While these lines may be visualized together, they are contained in different feature sets.

Though there are two separate feature sets, each AA must compute an opinion at each point in each feature set. To explain this more clearly, consider again Figure 4.2. AA_1, which extracted the black line, must compute an opinion at each point in the black line as well as in the red line. The same applies to AA_2 and raises the question, Why must an AA compute an opinion on a feature that it did not extract? This can be explained by looking at the structure of Figure 2.11 and Eq. 4.1. As seen, each AA computes an opinion on R. R is defined as every point that was extracted by *both* AA's; thus, each AA must compute an opinion at every point in both feature sets.

To compute the AA opinions at all points contained in R, the algorithm agents are separated into two parts: extracting agents (AA_E) and non-extracting agents (AA_{NE}). AA_1 extracted the features in feature set 1 (black line) and thus is AA_E at these points, while AA_2 becomes AA_{NE} at points in feature set 1. The setup is reversed in feature set 2, where AA_1 becomes AA_{NE} and AA_2 is AA_E. With extracting and non-extracting algorithm agents, each AA may compute an opinion at each point in all feature sets.

This methodology of extracting and non-extracting agents works with the current two-agent trust network as well as with multiple agents. At each point in a feature set, there will be one AA_E, with all other AA's assigned as AA_{NE}. The AA_E and AA_{NE} opinion calculations will be explained below.

Table 4.1: AA_E belief tuple setup.

AA_E	Set by
$b_R^{AA_E}$	AA_E Strengths
$d_R^{AA_E}$	AA_E Weaknesses
$u_R^{AA_E}$	Feature Characteristics

4.2.1 Extracting Algorithm Agent Opinion

The belief tuple set for AA_E is defined as follows: belief is set by extraction algorithm strengths, disbelief is set by extraction algorithm weaknesses, and uncertainty is set by flow feature characteristics. This may also be seen in Table 4.1.

From Table 4.1, it is clear that a good understanding of the AA as well as feature characteristics are required for successful opinion calculation. Both the belief and disbelief components of the AA_E opinion depend on a good working knowledge of the AA_E's strengths and weaknesses. When an extracted feature contains attributes that correspond to strengths of the algorithm, then belief is high. Conversely, when the extracted feature has attributes which correspond to algorithm weaknesses, disbelief will be high. Feature characteristics are scientifically known attributes of the feature being extracted, such as vortex core lines. For example, a vortex core line is the center of swirling flow in simple terms, and this physics-based characteristic may be used to define AA_E uncertainty. A requirement for the characteristics that make up the AA_E opinion is that they be quantifiable and can be manipulated such that Eq. 2.22 is true.

First-order functions are utilized for the equations which define belief, disbelief, and uncertainty. This framework was created by Mortensen and was shown to work well in different CFD data sets. The general form of the b, d, u equations can be seen in Eq. 4.7, where y is the opinion component and x is the parameter used to define the opinion component. The two values m_1 and m_2 are constants that are selected in order to satisfy Eq. 2.22. While this first-order assumption was made to calculate the belief tuple, other equations may be used, such as a quadratic fit. This equation format is also used for most belief components with a few exceptions. This setup will be explained below.

$$y = m_1 x + m_2 \tag{4.7}$$

Table 4.2: AA_E opinion values set for the SH vortex core extraction algorithm.

AA_E	Set by	Sujudi-Haimes
$b_R^{SH_E}$	AA_E Strengths	Straight core, high strength, low quality
$d_R^{SH_E}$	AA_E Weaknesses	Curved core, low strength, high quality
$u_R^{SH_E}$	Feature Characteristics	λ_2 criterion

Sujudi-Haimes Belief Tuple

Table 4.2 shows the strengths, weaknesses, and feature characteristics when Sujudi-Haimes is the AA_E. The SH algorithm was formulated with a linear flow field in mind and is designed to detect straight vortex cores; thus, straight lines were used as one of the strengths. Because of the linear flow field assumption, vortex cores with high rotational strength are well extracted, so high vortex strength is another of the strengths. Quality is a vortex core attribute which is independent of the extraction algorithm and low quality is a strength in both algorithms to define high belief.

The weakness characteristics defining the SH algorithm are the opposite of the strength characteristics. Curved core, low strength, and high quality are all characteristics that increase disbelief of vortex core lines extracted by SH. In other algorithms, the characteristics defining belief and disbelief need not be the same, though this was the case for the SH algorithm.

The λ_2 criterion was used as the vortex core characteristic defining the AA_E uncertainty for both the SH and RP algorithms. Because it is a Galilean invariant vortex definition, it performs well in finding moving vortex cores in unsteady simulations. However, it is not the only characteristic that may be used to define vortex core uncertainty. Streamline rotation may also be used if the convection velocity of the vortex core is subtracted from the surrounding flow field, but this would be a computationally expensive step for each vortex core in the flow field. Other methods such as the Q and the Δ criteria may also be added to the AA_E uncertainty in the future to increase the effectiveness of the uncertainty computation.

The AA_E belief tuple is created by quantifying the strengths, weaknesses, and feature characteristics. The belief tuple is calculated using Eqs. 4.8–4.10.

$$b_R^{SH_E} = 0.4 \cdot NormalAverage + 0.6 \tag{4.8}$$

$$d_{\mathrm{R}}^{\mathrm{SH_E}} = -0.4 \cdot NormalAverage + 0.4 \tag{4.9}$$

$$u_{\mathrm{R}}^{\mathrm{SH_E}} = \frac{1}{1+e^{-10\cdot\lambda_2}} \tag{4.10}$$

where

$$NormalAverage = \frac{NormalVortexStrength + NormalCurvature + NormalQuality}{3} \tag{4.11}$$

and

$$NormalVortexStrength = \begin{cases} \left|\frac{VortexStrength}{VortexStrengthNorm}\right|, & \left|\frac{VortexStrength}{VortexStrengthNorm}\right| < 1 \\ 1, & \left|\frac{VortexStrength}{VortexStrengthNorm}\right| \geq 1 \end{cases} \tag{4.12}$$

$$NormalCurvature = \begin{cases} \left|\frac{Curvature}{CurvatureNorm} - 1\right|, & \frac{Curvature}{CurvatureNorm} < 1 \\ 0, & \frac{Curvature}{CurvatureNorm} \geq 1 \end{cases} \tag{4.13}$$

$$NormalQuality = \begin{cases} \left|\frac{Quality}{QualityNorm} - 1\right|, & \frac{Quality}{QualityNorm} < 1 \\ 0, & \frac{Quality}{QualityNorm} \geq 1 \end{cases} \tag{4.14}$$

Eqs. 4.11–4.14 were created for the steady-state trust network in order to quantify the strengths and weaknesses of the SH algorithm. *NormalAverage* is created in such a way that $0 \leq NormalAverage \leq 1$. When $NormalAverage = 1$, this is the case when all of the strengths are satisfied and results in $b_{\mathrm{R}}^{\mathrm{SH_E}} = 1$ and $d_{\mathrm{R}}^{\mathrm{SH_E}} = 0$. Conversely, when $NormalAverage = 0$, $b_{\mathrm{R}}^{\mathrm{SH_E}} = 0.6$ and $d_{\mathrm{R}}^{\mathrm{SH_E}} = 0.4$, because the SH algorithm should have some belief in its own extraction. Eq. 4.12 was formulated so that high vortex strength contributed to a high *NormalAverage*. However, Eqs. 4.13 and 4.14 were created in such a way that low values of curvature (straight line) and quality contributed to a high *NormalAverage*.

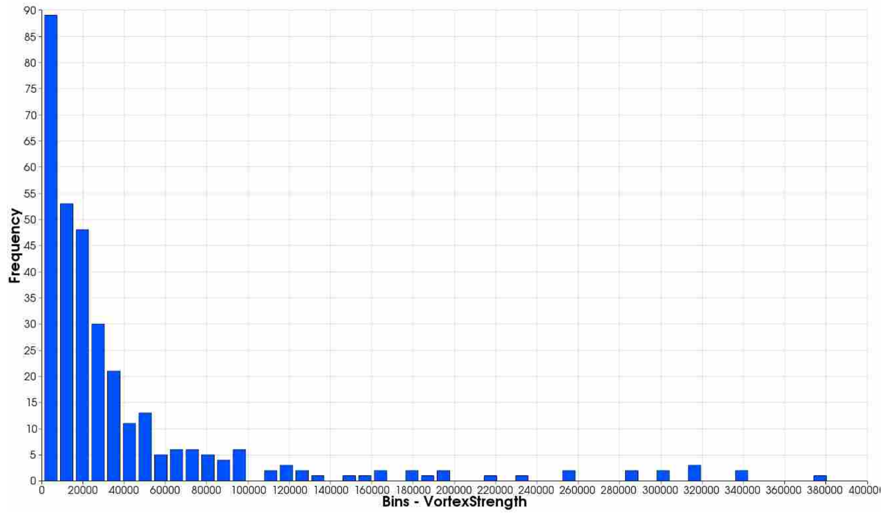
The constants in Eqs. 4.8 and 4.9 were created by Mortensen [17] to ensure a good spacing in the final opinion of the extracted features. Mortensen noticed that if certain constants were used in the belief and disbelief equations, the final opinion of the vortex core data set was bunched around one value, which increased the difficulty of discerning believable from spurious vortex cores. Figure 4.3 gives a graphical representation of vortex core opinions with good and poor spacing. In Figure 4.3(a), it is clear that the vortex core represented by the red circle is the most

(a) (b)

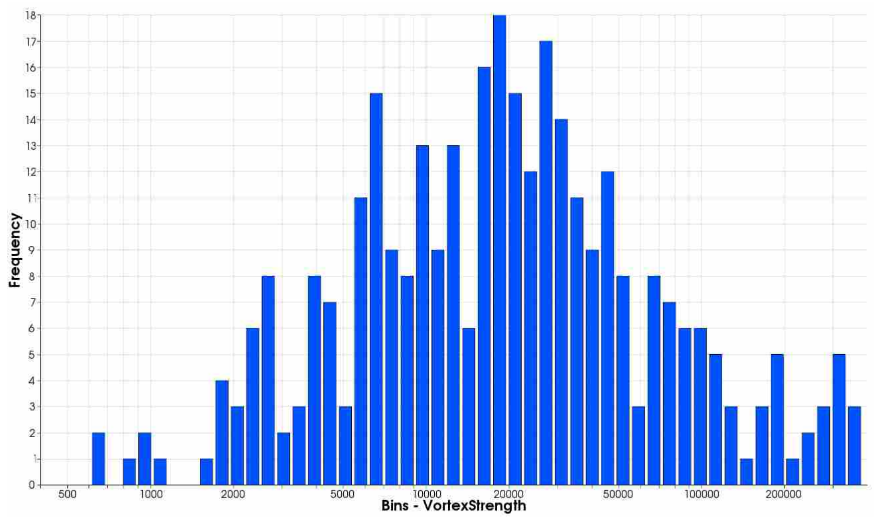
Figure 4.3: Opinions of vortex cores represented by circles on a scale of either belief or probability expectation. (a) Vortex core opinions with good spacing. (b) Vortex core opinions with poor spacing.

believable in the data set, with the vortex core represented by the blue circle as the second most believable. In Figure 4.3(b), it is much more difficult to tell that the red and blue circles are the most believable. Mortensen showed that the constants in the AA_E belief and disbelief equations resulted in well-spaced vortex core opinions. The same constants are used in the unsteady trust network and have been observed to also result in well-spaced vortex core opinions.

The normalization values *VortexStrengthNorm*, *CurvatureNorm*, and *QualityNorm* are used to require that *NormalVortexStrength*, *NormalCurvature*, and *NormalQuality* stay in the range of 0 and 1. In the steady state agent-based method, the normalization values were manually set for each data set. Quality has a known range from 0 to 90 degrees, so choice of *QualityNorm* is independent of the data set. However, vortex strength and curvature can vary widely from data set to data set, so an automated method of finding a proper *VortexStrengthNorm* and *CurvatureNorm* was created in this research. The distribution of the vortex core variables of vortex strength and curvature was found to be extremely positively skewed, which pulls the mean of the data toward the tail of the distribution. Using the mean of the data to normalize would then cause too few vortex cores to be believable, so a logarithmic transformation was applied to normalize the data. As seen in Figure 4.4, the original vortex strength data was highly skewed and most of the vortex strength values were less than 12,000. When a logarithmic transformation is performed on positively skewed data, the resulting distribution much more resembles a normal distribution [65]. The curvature data behaved in much the same way and became much more normally distributed after the logarithmic transformation. The anti-log of the mean of the transformed data is called



(a) Original data, which is very skewed and has high kurtosis.



(b) Logarithmically transformed data, which much more resembles a normal distribution.

Figure 4.4: Transformation of vortex strength data set to find a proper normalization value.

the geometric mean and is often used for data analysis when the data is highly skewed. The geometric mean of vortex strength and curvature were used to define *VortexStrengthNorm* and *CurvatureNorm*, respectively. In this manner, the choice of normalization values is much more robust and allows for a more general use of the overall method.

Another modification made to the steady-state trust network was the manner in which the AA_E uncertainty was calculated. Eq. 4.10 is not patterned after a first-order curve, but rather a logistic function, which is a type of sigmoidal curve that is often used in statistics and scientific

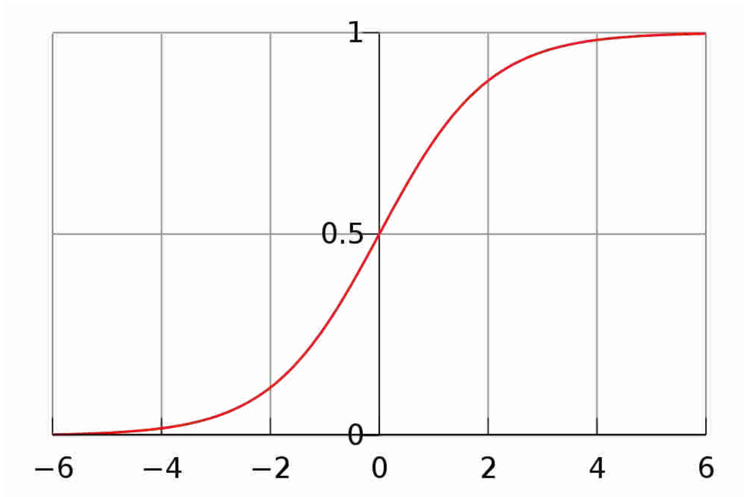


Figure 4.5: Representation of the logistic function of Eq. 4.15, where $m_1 = 1$ and $m_2 = -1$.

modeling. A general form of the logistic function is shown in Eq. 4.15 and a visual representation may be seen in Figure 4.5. In this function, m_1 controls the maximum function value and m_2 represents the slope of the curve around $t = 0$. This type of curve was well suited to the uncertainty computation when the λ_2 criterion was used, since points with λ_2 values of less than zero will be much less uncertain than points where $\lambda_2 > 0$. The stark contrast in uncertainty between negative and positive values of λ_2 would not be correctly reflected if a linear function were to be used. The slope of the function $m_2 = -10$ was chosen so that even slightly positive λ_2 values denote a relatively high uncertainty.

$$P(t) = \frac{m_1}{1 + e^{m_2 t}} \tag{4.15}$$

The constants which were used in the Eqs. 4.8–4.10 were chosen so that $b + d + u = 1$ is close to satisfied. When that condition is violated, belief is held constant while disbelief and uncertainty are decreased equally until the Eq. 2.22 is satisfied.

Roth-Peikert Belief Tuple

Table 4.3 shows the strengths, weaknesses, and feature characteristics when Roth-Peikert is the AA_E. Since the RP algorithm was created using a model of a perfectly semi-circular vortex core, one of its strengths is that it reliably detects curved vortex cores. The RP algorithm also detects cores with lower rotational strength and therefore is a strength of the algorithm. Although

Table 4.3: AA_E opinion values set for the RP vortex core extraction algorithm.

AA_E	Set by	Roth-Peikert
$b_R^{RP_E}$	AA_E Strengths	Curved core, low to high strength, low quality
$d_R^{RP_E}$	AA_E Weaknesses	Straight core, near zero strength, high quality
$u_R^{RP_E}$	Feature Characteristics	λ_2 criterion

RP can correctly extract weaker cores, it also performs well with high-strength cores which is also factored into the strengths.

RP algorithm weaknesses are set up similar to SH weaknesses with one exception: near zero vortex strength. Instead of quantifying weakness as the opposite of the strength of low vortex strength, merely a smaller magnitude is defined to set the RP weakness characteristic. Setting a straight core as a weakness may seem odd, since because the RP algorithm can reliably extract both straight and curved core lines. This was set as a weakness because there is more belief that the SH algorithm will extract straight lines more correctly than the RP algorithm. In this manner, each algorithm's belief and disbelief equations are set up to reflect the specific application for which they were created.

The λ_2 criterion was also used to define AA_E uncertainty for the RP algorithm. When multiple agents are extracting the same feature, the same feature characteristic may be used to define AA_E uncertainty for each algorithm since characteristics which define feature physics are not algorithm dependent.

The RP AA_E belief tuple is created similar to the method used for the SH algorithm. With the exception of Eq. 4.18, the following equations were created by Mortensen [17]. The belief tuple is calculated using Eqs. 4.16–4.18.

$$b_R^{RP_E} = 0.4 \cdot NormalAverage + 0.6 \tag{4.16}$$

$$d_R^{RP_E} = -0.4 \cdot NormalAverage + 0.4 \tag{4.17}$$

$$u_R^{RP_E} = \frac{1}{1 + e^{-10 \cdot \lambda_2}} \tag{4.18}$$

where

$$NormalAverage = \frac{NormalVortexStrength + NormalCurvature + NormalQuality}{3} \tag{4.19}$$

and

$$NormalVortexStrength = \begin{cases} \left|\frac{VortexStrength}{VortexStrengthNorm}\right|, & \left|\frac{VortexStrength}{VortexStrengthNorm}\right| < 1 \\ 1, & \left|\frac{VortexStrength}{VortexStrengthNorm}\right| \geq 1 \end{cases} \tag{4.20}$$

$$NormalCurvature = \begin{cases} \frac{Curvature}{CurvatureNorm}, & \frac{Curvature}{CurvatureNorm} < 1 \\ 1, & \frac{Curvature}{CurvatureNorm} \geq 1 \end{cases} \tag{4.21}$$

$$NormalQuality = \begin{cases} \left|\frac{Quality}{QualityNorm} - 1\right|, & \frac{Quality}{QualityNorm} < 1 \\ 0, & \frac{Quality}{QualityNorm} \geq 1 \end{cases} \tag{4.22}$$

The setting of the AA_E opinion for the RP algorithm only differs from the SH algorithm in the curvature calculation. The RP algorithm is designed to extract curved vortex cores so *NormalCurvature* will equal one when $Curvature \geq CurvatureNorm$. The automation method of using the geometric mean for *VortexStrengthNorm* and *CurvatureNorm* is also used for the RP algorithm AA_E. One of the RP algorithm's strengths is extracting weaker vortex cores, so the question of the validity of the automation method may be raised. However, after comparison of *VortexStrengthNorm* for both SH and RP, the RP data set always has a *VortexStrengthNorm* considerably lower than the SH data set. This is because the RP algorithm extracts many more cores which are weak, which shifts the geometric mean closer to zero.

4.2.2 Non-extracting Algorithm Agent Opinion

The belief tuple set for AA_{NE} is defined as follows: belief is set by extraction algorithm strengths, disbelief is set by extraction algorithm weaknesses, and uncertainty is set by distance from the current extracted vortex core line. This may also be seen in Table 4.4. For the AA_{NE}, belief and disbelief are set from the AA_E strengths and weaknesses. For example, if the SH algorithm is the AA_{NE}, then the RP algorithm strengths and weaknesses will be used for belief and disbelief. To compute minimum distance, the current point is compared to every other point in the

Table 4.4: AA_{NE} belief tuple setup.

AA_{NE}	Set by
$b_{\text{R}}^{\text{AA}_{\text{NE}}}$	AA_{E} Strengths
$d_{\text{R}}^{\text{AA}_{\text{NE}}}$	AA_{E} Weaknesses
$u_{\text{R}}^{\text{AA}_{\text{NE}}}$	Minimum distance from AA_{NE} extracted point

other extraction output. Again, as an example, if the SH algorithm is the AA_{NE}, the closest point is found in the SH algorithm's output and the distance between the two points is set as the minimum distance.

Linear functions, which Mortensen also created, are also used similar to the AA_{E} to define the belief, disbelief, and uncertainty of the AA_{NE} and can be seen in Eqs. 4.23–4.25.

$$b_{\text{R}}^{\text{AA}_{\text{NE}}} = 0.8 \cdot NormalAverage + 0.2 \tag{4.23}$$

$$d_{\text{R}}^{\text{AA}_{\text{NE}}} = -0.8 \cdot NormalAverage + 0.8 \tag{4.24}$$

$$u_{\text{R}}^{\text{AA}_{\text{NE}}} = 0.5 \cdot NormalMinimumDistance \tag{4.25}$$

where *NormalAverage* is computed from Eq. 4.11 if the RP algorithm is the AA_{NE} or Eq. 4.19 if the SH algorithm is the AA_{NE}. *NormalMinimumDistance* is computed using Eq 4.26.

$$NormalMinimumDistance = \begin{cases} \left|\frac{MinimumDistance}{MinimumDistanceNorm}\right|, & \left|\frac{MinimumDistance}{MinimumDistanceNorm}\right| < 1 \\ 1, & \left|\frac{MinimumDistance}{MinimumDistanceNorm}\right| \geq 1 \end{cases} \tag{4.26}$$

The reasoning behind the use of *NormalMinimumDistance* to calculate AA_{NE} uncertainty is that if the AA_{NE} extracts a vortex core very near to the AA_{E}, then the AA_{NE} will have very low uncertainty in the vortex core under consideration. The geometric mean of *MinimumDistance* was attempted to automate the choice of *MinimumDistanceNorm*, but resulted in an unfavorable value in many data sets. A much better choice of *MinimumDistanceNorm* is some key length scale from the data set, which requires a user input of this value. This is a very problem-dependent value but results in a better representation of AA_{NE} uncertainty. For this research, *minimumDistanceNorm* was changed for each data set to increase the range of belief values in the data set.

Table 4.5: MA belief tuple setup.

MA	Set by
$b_{\mathrm{AA_i}}^{\mathrm{MA}}$	Feature life
$d_{\mathrm{AA_i}}^{\mathrm{MA}}$	Feature life
$u_{\mathrm{AA_i}}^{\mathrm{MA}}$	Feature correspondence

From Eqs. 4.23–4.25, it can be seen that $b+d+u>1$ in certain cases. For example, if $NormalAverage = 1$, then $b_{\mathrm{R}}^{\mathrm{AA_{NE}}} = 1$ and $d_{\mathrm{R}}^{\mathrm{AA_{NE}}} = 0$. Rarely will two vortex core algorithms extract the same exact point for a simulation, which means that generally $u > 0$. To satisfy Eq. 2.22, the uncertainty is held constant while belief and disbelief are decreased equally until $b+d+u=1$.

4.3 Master Agent Opinion

Because the MA is the agent which computes the final opinion of the feature in R, it has the most influence on the believability of extracted vortex cores. It performs the duty of combining the opinions of all the AA's and providing a final belief tuple on the extracted vortex cores – $\omega_{\mathrm{R}}^{\mathrm{MA}}$. The MA opinion is set up to be impartial to an individual algorithm's strengths and weaknesses and is more related to the type of data set from which the features are extracted, i.e. steady-state or time-dependent, rather than certain algorithm characteristics or feature flow physics. This results in a markedly different computation of the MA belief tuple from that of AA belief tuples.

In steady-state data sets, the assumption that was used to compute the MA opinion was that believable features moved very little in between iterations, since a converged simulation should contain stationary features. However, this assumption fails in time-dependent simulations, where features are expected to move and interact through time. In transient data sets, the MA belief tuple is formed on the assumption that believable vortex cores will be those which exist for multiple time steps and behave predictably through time. Feature tracking is the method used to determine the parameters of $FeatureLife$, or how many time steps the vortex core exists, and feature correspondence (*Corr*), or how well the feature was tracked to a feature in the next time step. These parameters, which are used to define the MA belief tuple, can be seen in Table 4.5.

Eqs. 4.27–4.29 are used to compute the belief, disbelief, and uncertainty of the Master Agent (MA).

$$b_{\mathrm{AA_i}}^{\mathrm{MA}} = 0.5 \cdot NormalFeatureLife + 0.5 \tag{4.27}$$

$$d_{\mathrm{AA_i}}^{\mathrm{MA}} = -0.5 \cdot NormalFeatureLife + 0.5 \tag{4.28}$$

$$u_{\mathrm{AA_i}}^{\mathrm{MA}} = \frac{1}{1 + e^{5 \cdot Corr}} \tag{4.29}$$

where $NormalFeatureLife$ is computed from Eq. 4.30.

$$NormalFeatureLife = \begin{cases} \frac{FeatureLife}{FeatureLifeNorm}, & \frac{FeatureLife}{FeatureLifeNorm} < 1 \\ 1, & \frac{FeatureLife}{FeatureLifeNorm} \geq 1 \end{cases} \tag{4.30}$$

The constant in the MA belief and disbelief equations were chosen to give the MA opinion equal weight for belief and disbelief. The MA opinion operates on generic vortex cores and is impartial to the extraction algorithm, which is why the constants of 0.5 were chosen. If a vortex core has been well tracked ($NormalFeatureLife = 1$), then $b = 1$ and $d = 0$. On the opposite extreme, if $NormalFeatureLife = 0$, then $b = 0.5$ and $d = 0.5$. It may seem that a poorly tracked line should receive a belief value of 0, but this is accounted for in uncertainty. When $b+d+u > 1$, uncertainty is held constant while belief and disbelief are decremented by an equal value until $b+d+u = 1$ is satisfied.

$NormalFeatureLife$ is formulated in a similar manner to other normalization parameters so that it is on the range of 0 and 1. $FeatureLifeNorm$ is a parameter which must be selected by the user. This is the number of time steps that a believable feature is expected to exist and may be any integer value greater than or equal to 2. In this research, $FeatureLifeNorm$ was selected after visual inspection of the data set to give good spacing to the opinion. If $FeatureLifeNorm$ is set too low, most of the vortex cores will have belief values clustered around 1, and if $FeatureLifeNorm$ is set too high, the belief of all vortex cores is reduced in a similar manner. In the three data sets that will be considered in Chapter 5, $FeatureLifeNorm$ was a value from 10 to 30, depending upon the data set. In a data series with a low number of time steps, $FeatureLifeNorm$ would need to be smaller to give a good spacing to the vortex core opinions.

The MA uncertainty is based on line correspondence, *Corr*, and is computed using a logistic function. This is because *Corr*, which is computed using Eq. 3.10, has a range of $-\infty$ to 1. A *Corr* greater than 0 denotes a tracked feature, and thus much lower uncertainty is imparted to vortex cores with a $Corr < 0$. A slope of 5 at the origin is used which gives a perfectly matched line ($Corr = 1$) an uncertainty of $u^{\mathrm{MA}}_{\mathrm{AA_i}} = 0.007$. A logistic function is well suited to a parameter such as *Corr* and eliminates the need to normalize the characteristic defining MA uncertainty. Because each vortex core may be tracked to two other vortex cores (one in the previous time step and one in the next time step), the higher *Corr* parameter is used to define MA uncertainty.

4.4 Aggregation of Believable Features into a Final Data Set

One of the end goals of the intelligent feature extraction method is to select believable features extracted from different extraction algorithms and combine them into one final feature set. In previous work, visual inspection was used to find and remove low-belief and duplicate features. A two-step method was developed to automate the feature set combination so that larger data sets with many features could be operated upon. Bear in mind that the final opinion $\omega^{\mathrm{MA}}_{\mathrm{R}}$ is computed before the automated feature set combination. The two steps are as follows:

1. Remove features below a user-defined opinion threshold.

2. Find duplicate features and remove the duplicate with lower belief.

The first step of removing low-belief features is fairly trivial except for the selection of variable and threshold value. Different variables such as belief, disbelief, or uncertainty may be used, though probability expectation (E) is the most commonly used in subjective logic to define feature opinion. The next issue becomes selection of threshold value, since in subjective logic there is no hard-and-fast rule for what is believable. In this research, line-averaged E is used, and vortex core lines with an average value of $E < 0.75$ are removed. In point-based applications of subjective logic, $E > 0.85$ is commonly used as a metric for believable features, so a lower threshold value was used for line-averaged E since probability expectation can vary considerably in a vortex core line.

The next step of finding and removing duplicate vortex cores is accomplished by using length and position tolerances. It is a rare occurrence that two extraction algorithms will extract

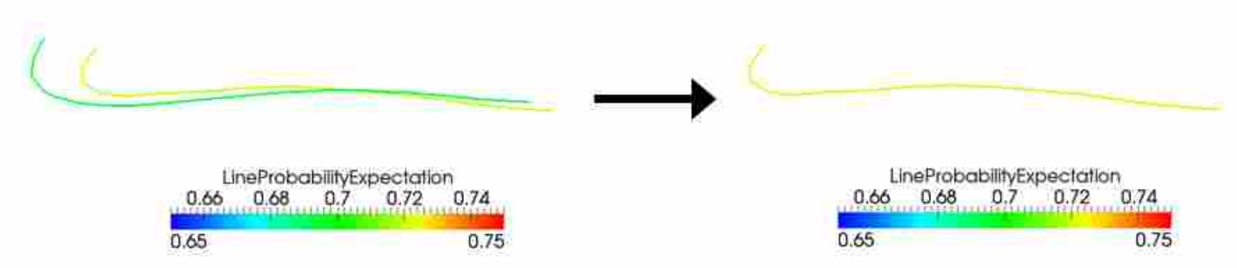


Figure 4.6: Example of two vortex core lines which are automatically verified as duplicates. The vortex core with high average probability expectation is kept and the other is removed.

vortex cores in the exact same location, so tolerances are used to match two lines. To reduce computational demands, only features within one line length of the feature under consideration are inspected, similar to the process presented in Section 3.3.3. The length and position tolerances employed in duplicate feature matching are shown in Eqs. 4.31 and 4.32. These equations are formed so that they are in a range from 0 to 1, where lower function values denote highly duplicate lines. When both Eq. 4.31 and 4.32 are less than 0.1, the lines are considered duplicates. The duplicate lines are compared, and the vortex core with a lower average E is removed. An example of this operation can be seen in Figure 4.6. Though it is apparent that the two lines are very similar, the automatic method created here removes the lower belief line without any user input.

$$f_{length} = \frac{|L_1 - L_2|}{max(L_1, L_2)} \tag{4.31}$$

$$f_{position} = \frac{\|P_1 - P_2\|}{L_1} \tag{4.32}$$

The automated method created here currently only operates on line-type features. For shell features such as shock waves, surface area might replace line length in Eq. 4.31, and for volume features, volume may be successful in place of length for finding duplicate features. Eq. 4.32 would also require modification to work for other types of features.

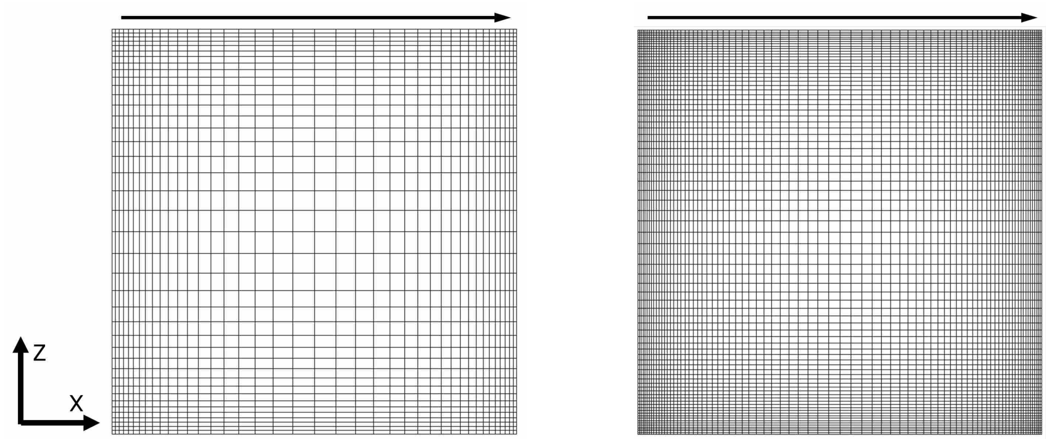
CHAPTER 5. RESULTS AND DISCUSSION

Two benchmark simulations were run on different geometries in order to test the time-dependent feature extraction and tracking framework described in Chapters 3 and 4. The three-dimensional cubic lid-driven cavity has been extensively studied and contains well-defined vortex core lines. It is a simple data set which is simple to set up and run quickly. The three-dimensional cylinder in cross flow, another classical unsteady flow problem which exhibits the famous von Kármán vortex street, was also used in this research. The cylinder case has a more complex unsteady flow field and was selected to validate the unsteady vortex visualization method.

A massive simulation of a wind turbine was obtained in order to test the method on a large data set. The results of vortex core line extraction from the two benchmark simulations as well as the massive data set are shown below.

5.1 Lid-Driven Cavity

A CFD simulation of a cubic lid-driven cavity [66] was run using the unsteady laminar Navier-Stokes equations and was solved in Fluent 12. The Pressure Implicit Splitting of Operators (PISO) algorithm was used for pressure-velocity coupling with second-order implicit stepping through time. For each time step, 40 Newton sub-iterations were computed to attain convergence. The lid of the cavity was impulsively started at $t = 0\ s$ in order to view the development of the vortex cores. The Reynolds number based on cavity side length and lid velocity was 1000, at which the flow was laminar and became steady after a period of time. Two structured grids were created ($40 \times 40 \times 40$ and $80 \times 80 \times 80$) to find the influence of grid density on vortex core extraction. A slice of each grid can be seen in Figure 5.1. Grid clustering was employed near the walls to account for wall effects, and all boundaries were set with a no-slip boundary condition. The lid was also modeled as a no-slip wall, but with a constant velocity in the x-direction. Other properties of the simulation may be seen in Table 5.1.



(a) Coarse mesh – 64,000 nodes (b) Fine mesh – 512,000 nodes

Figure 5.1: Slices of the computational meshes created for the lid-driven cavity simulation. The lid, denoted by the side with an arrow over it, is moved at a constant velocity in the $+x$-direction.

Table 5.1: Cavity simulation parameters.

Time step (s)	Lid velocity (m/s)	Side length (m)	Reynolds number
0.01	1	0.1	1000

The simulation was run for a total time of 10.0 *s* and was saved at each time step. A visualization of the flow evolution through time may be seen in Figure 5.2, where the fine mesh simulation was used. At early time steps, the central vortex moved from the top right corner to the center and grew in size. At later time steps, secondary corner vortices developed and also grew in strength, though they were much weaker than the primary vortex. Little change occurred to the flow domain after 5.5 *s*, or the equivalent of 55 lid passings. Though the full domain was modeled, the data set showed a high degree of symmetry around the *xz*-midplane of the cavity.

5.1.1 Vortex Cores Extracted from Data Set

Vortex cores were extracted with the time derivative modification from the lid-driven cavity data set. The vortex cores extracted by the SH and RP algorithms at the steady state condition can be seen in Figure 5.3. As shown in Figure 5.3(a), the SH algorithm extracted the main vortex cores from the data set: the primary, secondary, and corner vortex cores. They were disconnected near the *xz*-midplane, and the vortex cores around the *xz*-midplane were quite symmetric. As shown

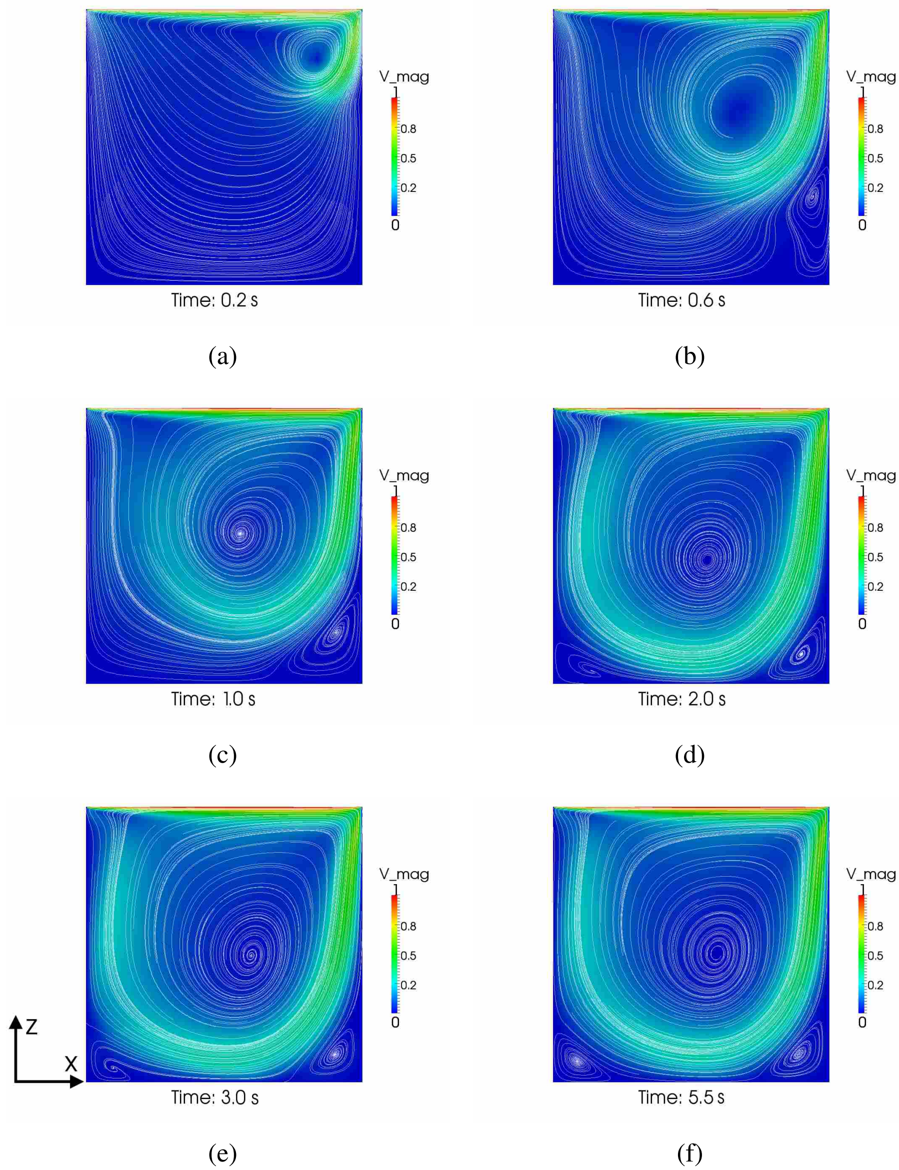
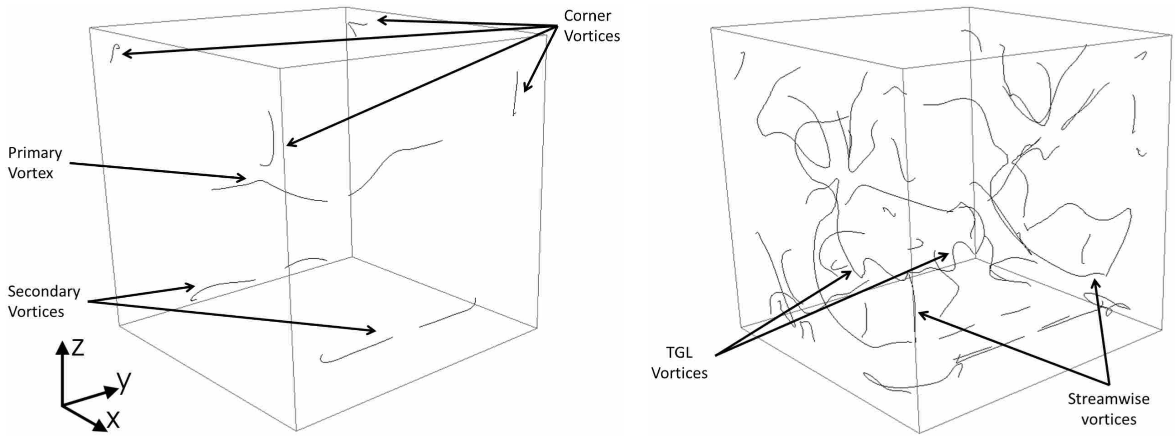


Figure 5.2: Visualization of the lid-driven cavity data set. Streamlines are traced in the y-midplane, and the slice is colored by velocity magnitude. The lid moves in the $+x$ direction and the velocity is in m/s.

in Figure 5.3(b), the RP algorithm extracted many more vortex cores which would be difficult to differentiate without vortex core line extraction because they were fairly close to each other. By visual inspection in the CFD data set, some of the vortex cores extracted by the RP algorithm were confirmed to be spurious, while others in the RP data set were similar to those extracted by the SH algorithm and were verified to be correct vortex cores. Taylor-Görtler-Like (TGL) vortices – streamwise vortices along the wall of the cavity – were also discernible in the RP data set and have



(a) SH vortex cores.

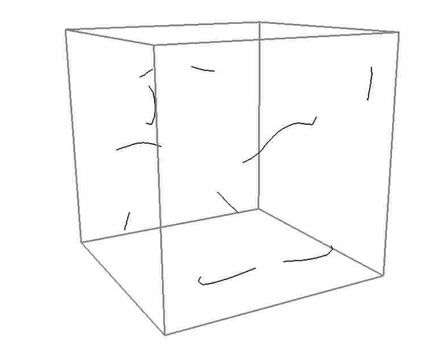
(b) RP vortex cores.

Figure 5.3: Vortex cores extracted by the SH and RP algorithms. Key vortex structures are listed. The lid moves in the $+x$-direction.

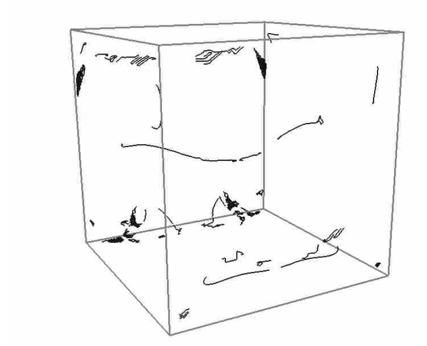
been verified by Albensoeder at this flow regime [66]. Other vortex cores of interest which were extracted by the RP algorithm were the long stream-wise vortex cores which were extracted near the walls in the $+$ and $-y$-directions.

Grid density was investigated in the lid-driven cavity data set to understand its effect on unsteady vortex core extraction. Figure 5.4 shows both extraction algorithm outputs for the two grids. All extractions shown were from the same time ($5.0s$), when the vortex cores were still moving into their steady positions. It can clearly be seen that the existing vortex cores were refined as the grid was refined, which can be seen especially in the case of the primary vortex. Both algorithms extracted a clear primary vortex in the fine grid, whereas both failed to extract contiguous primary core lines from the coarse mesh. Also, both algorithms detected new vortex cores in the fine mesh case that were not found in the coarse mesh. Some of the new vortex cores in the fine mesh were found to be true vortex cores, while others were verified to be false, especially some of the shorter extracted vortex cores in the data set. With clearer true vortex cores in the fine mesh came a cost – many small, intertwining vortex cores were extracted in the corners at certain time steps, which appeared as vortex regions and were generally false detections of the extraction algorithms.

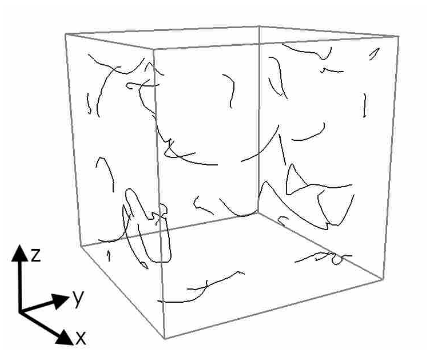
In order to correctly extract the main vortex cores, the cost of finding more vortex cores was acceptable in this data set. In a larger, more complex data set, the trade-off of CFD data size and correctness of extracted vortex cores would need to be taken into consideration. The process



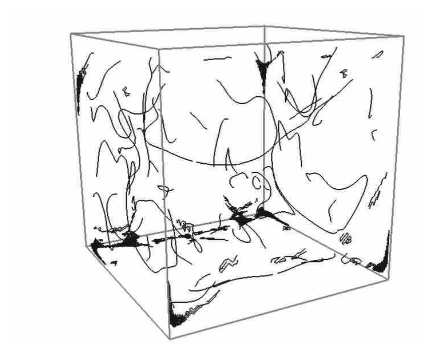
(a) SH, coarse mesh.



(b) SH, fine mesh.



(c) RP, coarse mesh.



(d) RP, fine mesh.

Figure 5.4: Effect of grid density on vortex core extraction in the lid-driven cavity data set. The lid moves in the $+x$-direction.

of manually verifying the many vortex cores in this data set was a laborious task, which increased the attractiveness of applying subjective logic to automatically detect the true vortex cores in the data set.

5.1.2 Influence of Time Derivatives on Extracted Vortex Cores

Time derivatives were computed and added to the feature extraction process in order to more correctly extract vortex core lines from time-dependent flows. The difference between cores extracted with and without time derivatives can be seen in Figure 5.5. For ease of visualization, the coarse data set was used to investigate the influence of time derivatives. At early time steps ($t = 0.2 - 0.6\ s$), the primary vortex moved significantly through the data set, and this movement was shown by the noticeable difference in vortex cores extracted with and without velocity time derivatives. At intermediate time steps ($t = 1 - 3\ s$), the secondary vortices were still developing, though the top corner vortex cores had become fully developed. As the simulation reached a steady-state condition ($t = 5.5$ s), the core lines extracted with and without the time derivatives were identical.

Though the effect of time derivatives on vortex core extraction appeared somewhat minimal in this data set, it was due to the fact that velocity and length scales are small. In larger data sets with higher Reynolds numbers, the addition of time derivatives will likely result in vortex cores which are shifted further from those extracted under a steady-state assumption. Even in this low flow situation, the difference in extracted cores was visually noticeable as the vortex cores moved. It was also shown that many vortex cores extracted under the steady-state assumption were spurious, so the extra computational cost of computing time derivatives was seen as a favorable step in unsteady vortex core extraction.

5.1.3 Vortex Cores Processed by Agents

The vortex cores extracted by both algorithms were processed by the agent-based trust network to determine the belief tuple of the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$. This was performed in all time steps of the cavity data set, but only one time step will be considered here (3.0 s). Figure 5.6 shows the belief and disbelief values of SH and RP vortex cores and highlights some of the strengths and weaknesses of each algorithm. When looking at the belief values for the SH vortex cores in Figure 5.6(a), one can see that high belief ($\sim 0.75-1$) was calculated for the primary and top near corner vortex cores, with low belief in the top far corner vortex cores. Similarly, in Figure 5.6(c), it can be seen that the opposite occurs in the disbelief values in the vortex cores. Since the SH algorithm was designed to extract strong, straight vortex cores, only the vortex cores which were straighter and had higher vortex strength contained higher belief values. The local curvature calculation was also seen to be successful, since the highest belief occurred in portions of the vortex cores with the lowest local curvature. In the RP data set the belief values as seen in Figure 5.6(b) were around 0.5 for the longer vortex cores, with low belief calculated for short vortex cores near the far wall of the cavity. The RP algorithms strengths included curved and weaker vortex cores, which was why higher belief was given to the curved, weaker corner vortex core lines. The effects of imparting low disbelief to highly curved vortex cores can be seen in Figure 5.6(d); the lowest disbelief values occurred in areas of the vortex cores where curvature was highest.

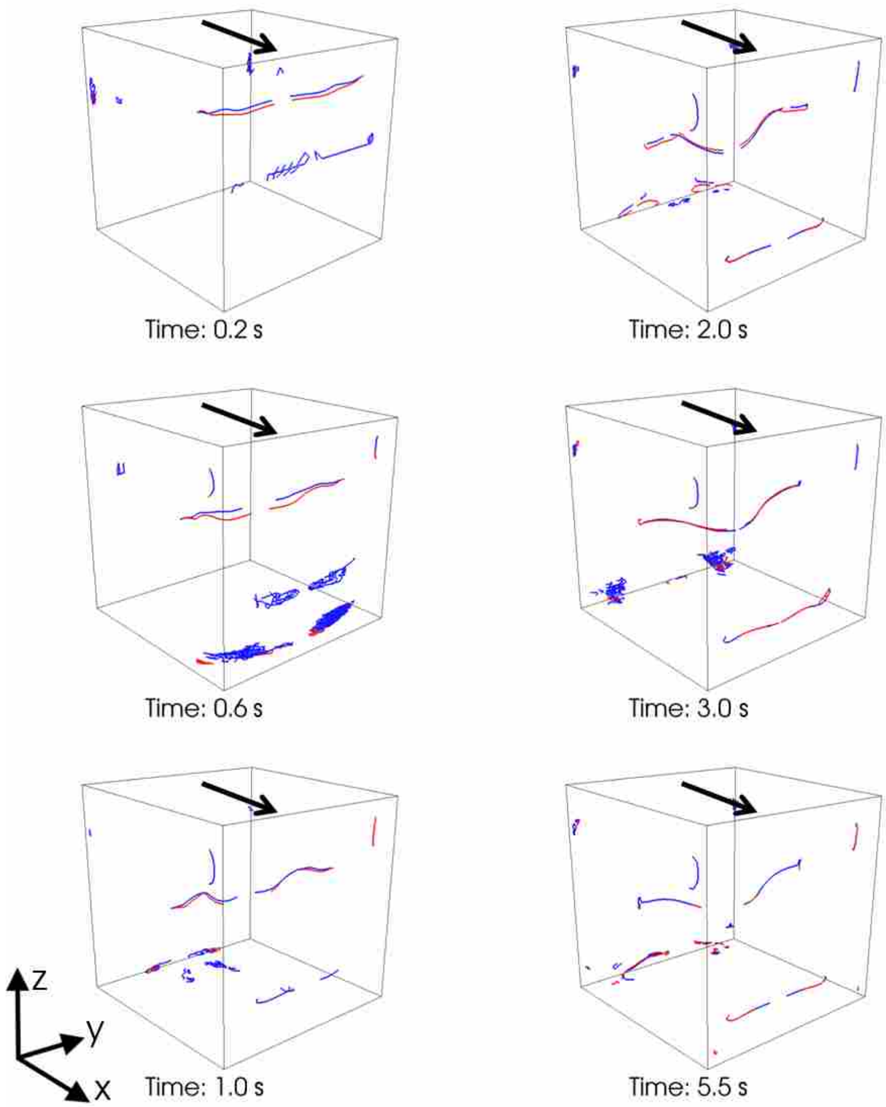
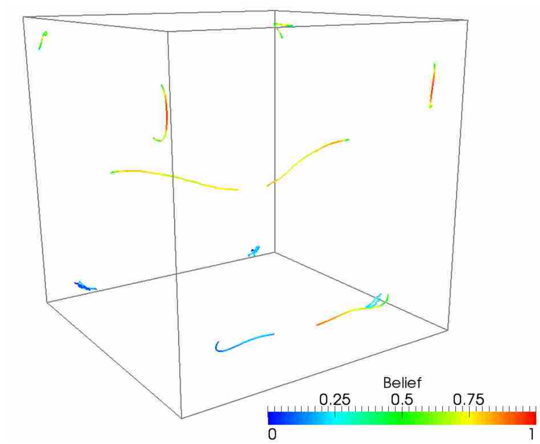
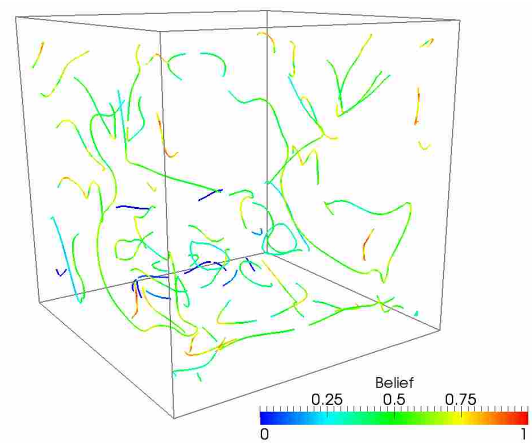


Figure 5.5: Vortex cores extracted from the lid-driven cavity case using Sujudi-Haimes: red cores – time derivatives included, blue cores – no time derivatives (steady-state assumption).

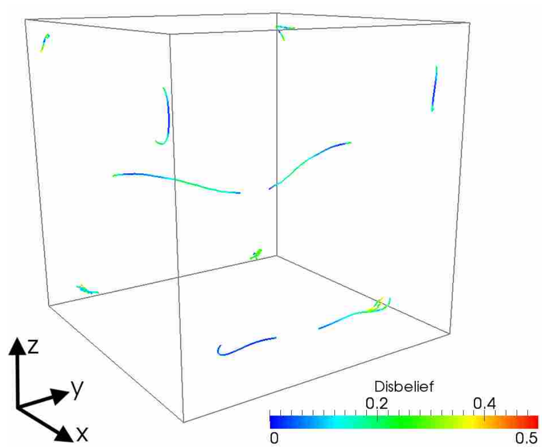
A comparison of uncertainty values of both agents' vortex cores revealed how closely the output agreed with the parameters used to define uncertainty. Figures 5.7(a) and 5.7(b) show the uncertainty values for the SH and RP vortex cores, respectively. The SH vortex cores which had low uncertainty calculated were both well tracked through time and had a λ_2 value of less than zero for most of the cores. The cores with higher uncertainty, especially the short vortex cores extracted in the far bottom corner of the cavity, were poorly tracked since they would often "flash" in and out of consecutive time steps, which greatly increased the difficulty of tracking. In the RP data set,



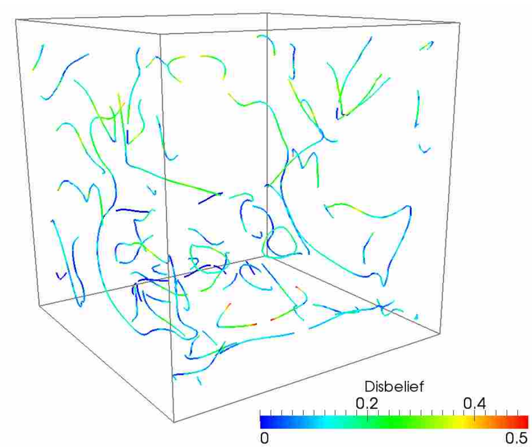
(a) SH belief values.



(b) RP belief values.



(c) SH disbelief values.

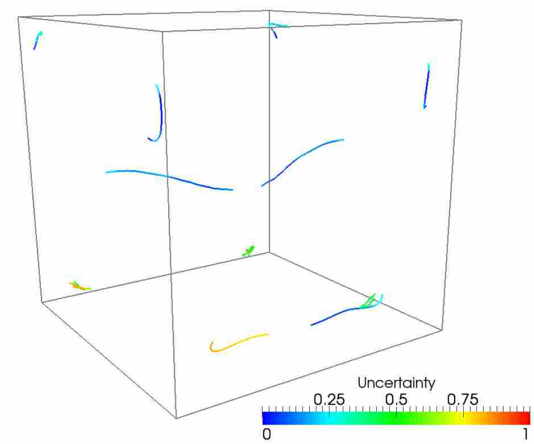


(d) RP disbelief values.

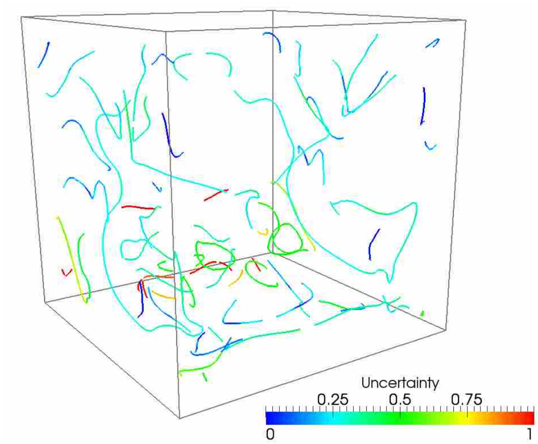
Figure 5.6: Comparison of the belief and disbelief values from the final opinion ω_{R}^{MA} of the vortex cores extracted by the SH and RP algorithms from the lid-driven cavity data set. The lid moves in the $+x$-direction.

a large range of uncertainty values was calculated for the vortex cores. Very low uncertainty was calculated for the top corner vortex cores, with uncertainties of roughly 0.25 obtained for the bulk of the longer vortex cores. The highest uncertainties were calculated where the vortex cores were not tracked at all. In general, it was observed that feature tracking had a larger effect on the RP vortex core uncertainty than the λ_2 criterion.

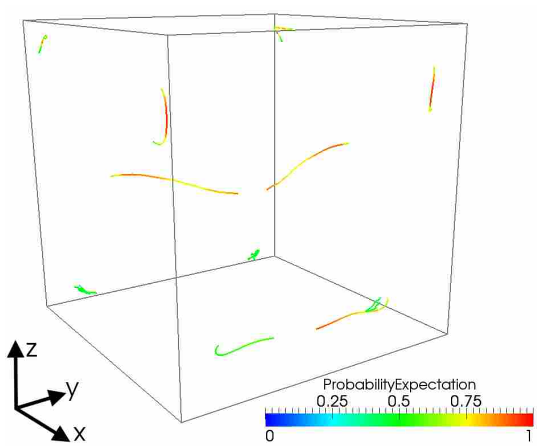
The probability expectation (E) of the vortex core data sets revealed the most believable vortex cores and which algorithm extracted them. Recall that E is calculated using Eq. 2.24, which takes into account the final belief and uncertainty and gives what one would expect the probability of a feature to be. The vortex cores colored by E may be seen in Figures 5.7(c) and 5.7(d) for the



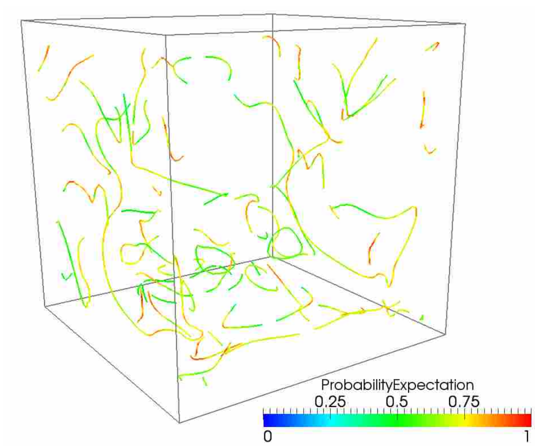
(a) SH uncertainty values.



(b) RP uncertainty values.



(c) SH probability expectation values.



(d) RP probability expectation values.

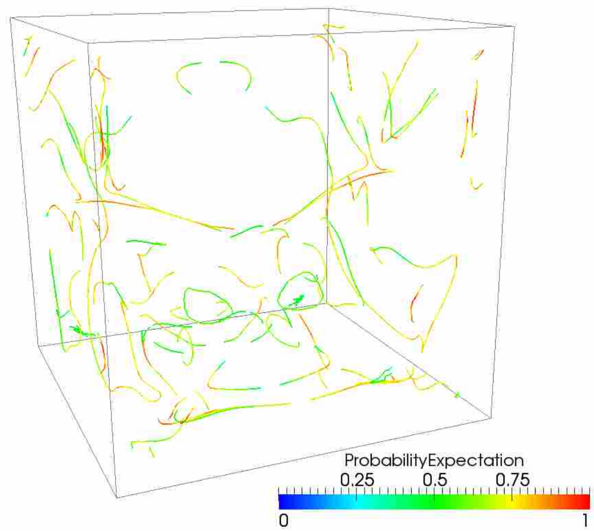
Figure 5.7: Comparison of the uncertainty value and probability expectation from the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$ of the vortex cores extracted by the SH and RP algorithms from the lid-driven cavity data set. The lid moves in the $+x$-direction.

SH and RP algorithms, respectively. The SH algorithm clearly extracted more believable primary and near top corner vortex cores, since the values of E in these vortex cores were greater than 0.75. The RP algorithm had higher E in the weaker, more curved vortex cores, which included the long stream-wise vortex cores and the smaller TGL vortex cores near the far back wall of the cavity. However, the RP data set was too cluttered with vortex cores with low E to clearly visualize some of the vortex cores with high probability expectation.

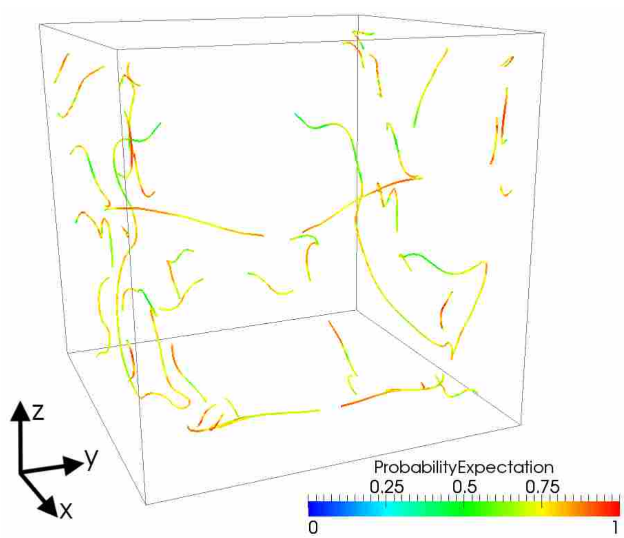
5.1.4 Automatic Combination of Data Sets

The vortex cores extracted by the SH and RP algorithms were combined using the method outlined in Section 4.4. The results of this automatic operation can be seen in Figure 5.8, where the two unfiltered data sets are shown in Figure 5.8(a) and the final data set is shown in Figure 5.8(b). Many of the vortex cores with low average probability expectation were removed from the data set, which clearly reduced the amount of visual clutter. In the final data set, many of the expected vortex cores (primary, secondary, corner, stream-wise, and TGL) had a high probability expectation and were included in the final feature set. The second step of the feature set combination method was to find vortex cores which had been extracted by both algorithms and select the more believable vortex core. This step was required to select the most believable primary and secondary cores, and the SH algorithm was generally found to be more successful through time in extracting both of these vortex cores. However, the check for duplicate cores failed in the case of the near top corner vortex cores, where both algorithms extracted similar vortex cores with high probability expectation. They were not similar enough in size or location to be automatically detected. However, the method performed well in most cases and created a data set with believable vortex cores from both algorithms.

Verification was performed on all of the expected vortex cores as well as some of the spurious vortex cores. This was accomplished by use of streamlines and cutting planes of the CFD data set. The use of subjective logic was also proven to be effective at finding the correct vortex cores in the data set. For example, streamlines seeded around the primary vortex core showed that the SH algorithm was more successful at correctly extracting the primary core, which was also shown by use of subjective logic. Other vortex cores with high expected probability also agreed with the swirling flow definition by use of streamlines. Cutting planes of the CFD data set colored by vortex strength also showed the success of subjective logic in detecting spurious vortex cores. Vortex cores with low E also agreed with regions of very low vortex strength, though this was only one of the characteristics used in subjective logic. Visualization of the verification of these vortex cores may be seen in Appendix A.



(a) Unfiltered output of both vortex core extraction algorithms.



(b) Final feature set, which includes only believable vortex cores.

Figure 5.8: Automatic combination of two different algorithm outputs shown in the cavity data set. The lid moves in the $+x$-direction.

5.2 Cylinder in Cross Flow

The second CFD simulation used in this research to validate the unsteady feature extraction method was the case of a cylinder in cross flow [67]. This simulation was chosen as a validation case for the method because of the more complex flow field in the cylinder wake and the convection of the vortex cores through the domain. The Reynolds number based on cylinder diameter and

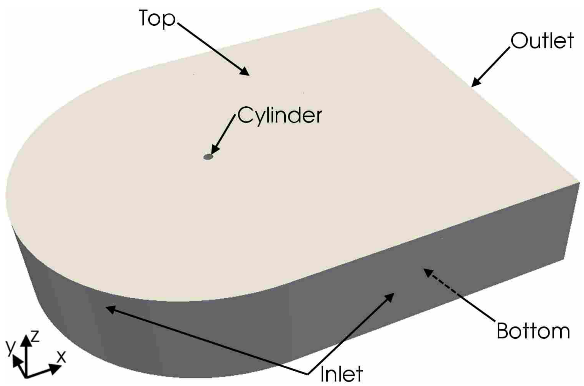
Table 5.2: Cylinder mesh details.

Mesh type	Node count	Number of points			
		x	y	z	Cylinder
Unstructured	$1,691,412$	100	50	100	80
Structured	$1,026,000$	68	86	75	85
Structured, fine	$4,222,773$	156	140	115	200

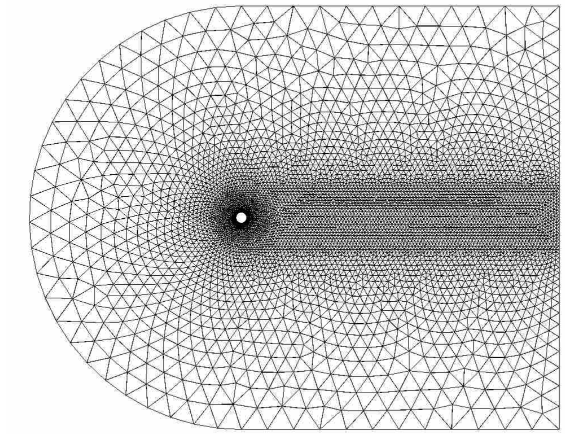
freestream velocity was 300, at which three-dimensional mode B shedding has been documented in both experimental and numerical results. Mode B vortex shedding, according to Williamson [68], "comprises finer-scale streamwise vortices, with a spanwise length scale of around one diameter. The large intermittent low-frequency wake velocity fluctuations, originally monitored by Roshko [69] and then by Bloor [70], have been shown to be due to the presence of large-scale spot-like 'vortex dislocations' in this transition regime. These are caused by local shedding-phase dislocations along the span."

The unsteady, incompressible Navier-Stokes equations were solved in Fluent 12. The PISO algorithm was used for pressure-velocity coupling with second-order implicit stepping through time. Three different meshes were created to view the effectiveness of the method in differing grid types: unstructured, structured, and very fine structured. These three meshes, as well as a view of the computational domain, can be seen in Figure 5.9. As seen in the 3 slices, the wake was refined in order to capture the vortical flow structures that were shed from the cylinder. The domain extended 20 cylinder diameters upstream of the cylinder, 30 diameters downstream of the cylinder, and 10 cylinder diameters in the spanwise direction. A no-slip wall boundary condition was applied to the cylinder wall, a velocity inlet with a prescribed x-velocity was used for the inlet, and a pressure outlet boundary condition was used for the domain outlet. Symmetry boundary conditions were used for the top and bottom of the domain in order to recreate the conditions of the simulations run by Zhang et al. [67]. Table 5.2 give more details about the meshes.

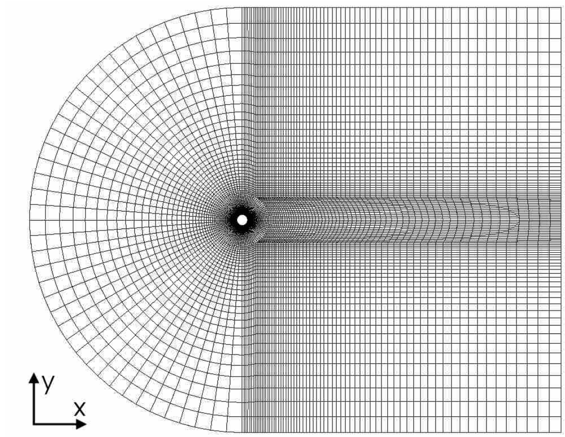
The simulations were run until the flow became quasi-steady. This was determined by investigation of the drag coefficient history. After 2000 time steps, the drag coefficient oscillated around the same mean value and the flow was deemed to be fully developed. The drag coefficient values were compared to the Direct Numerical Simulation (DNS) results of Zhang et al. [67] and can be seen in Table 5.3.



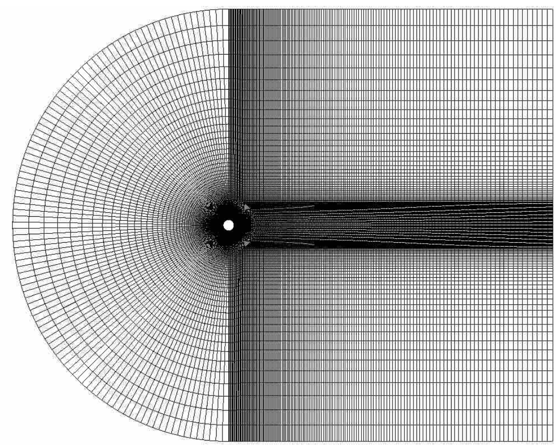
(a) Full 3-D domain.



(b) Slice of the unstructured mesh.



(c) Slice of the structured mesh.



(d) Slice of the fine structured mesh.

Figure 5.9: Computational meshes used in the simulation of a cylinder in cross flow. Flow moves in the $+x$-direction.

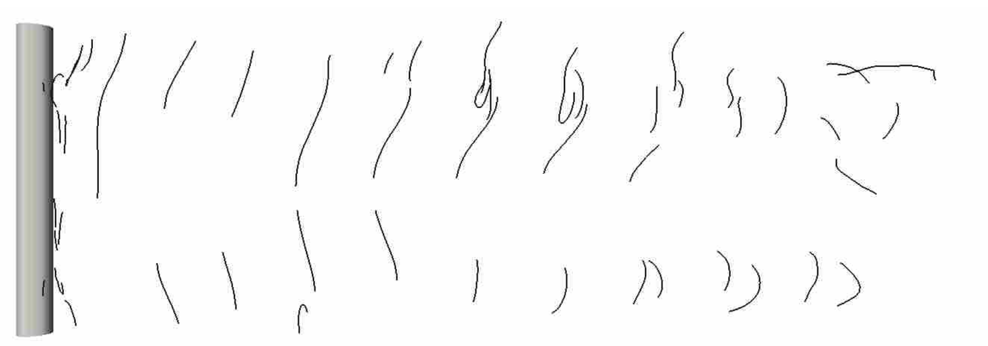
Table 5.3: Cylinder data set drag coefficient study.

Mesh	Time Step (s)	C_D	$C_{D,\text{DNS}}$	Error (%)
Unstructured	0.05	1.132	1.278	11.4
Structured	0.05	1.242	1.278	2.85
Structured, fine	0.01	1.298	1.278	1.54

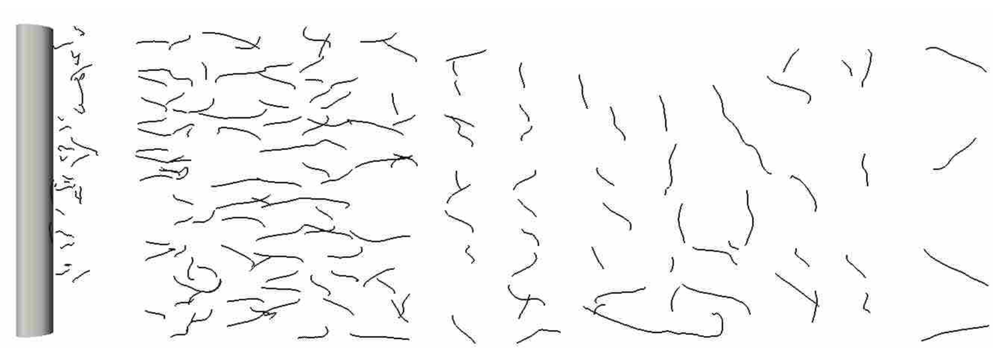
5.2.1 Comparison of Vortex Cores Extracted from Different Grids

Vortex cores were extracted from each of the grids of the cylinder data set using the RP and SH algorithms. Figure 5.10 shows the representative results obtained by the RP algorithm from the three grids studied. Note that the vortex cores shown here were not extracted from the same time

(a) Unstructured mesh results.



(b) Structured mesh results.



(c) Fine structured mesh results.

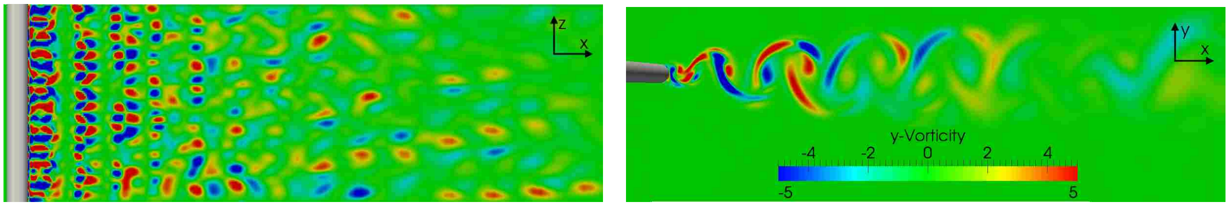
Figure 5.10: Vortex cores detected by the RP algorithm from the three different types of grids. Flow moves from left to right.

step in the simulation, so the vortex core locations were not exactly the same. It can be seen that the type of mesh and grid resolution had a significant impact on the vortex core extraction process.

The vortex cores extracted from the unstructured mesh appeared very jagged and unphysical, as seen in Figure 5.10(a). The jaggedness of the vortex cores was due to the due to the nature of the vortex point detection and the line connection algorithm used. Both vortex core extraction algorithms used the PV operator to investigate each cell edge in the domain and determine if two vector fields were parallel at the node points of the edge. Linear interpolation was then used to find the point on the edge where the PV operator was satisfied. The line connection algorithm then connected points which were extracted from cells with a common edge. In the unstructured data set, cell neighbors were not well ordered, and thus the interpolation and connection process resulted in jagged core lines. It was also observed that the vortex cores extracted were not parallel to the cylinder, nor were they continuous through the domain. One important note was that the drag coefficient in the unstructured mesh simulation was quite different from DNS result, which meant that the under-resolved simulation may have also had an effect on the extracted vortex cores.

The vortex core lines extracted from the structured mesh, as seen in Figure 5.10(b), were much more smooth but still exhibited similar characteristics to the unstructured mesh vortex cores. While some of the core lines were continuous through most of the spanwise domain, there were none that extended the whole length of the cylinder. Also, in the far downstream wake of the cylinder, the vortex cores were very disconnected and curved, which may be attributed both to mesh coarsening and breaking up of the vortex cores as they were convected downstream. Using a particle trace, some of the cores extracted by the RP algorithm near the cylinder were verified to follow swirling particle flow, though the short vortex cores far downstream of the cylinder were not in the centers of swirling flow. The SH algorithm generally failed to extract correct vortex cores, which was due to the fact that the vortex cores in the far wake were quite curved, which was a weakness of the SH algorithm. One of the weaknesses of both algorithms as noted by Roth [7] was that a vortex core with a non-constant acceleration along the core was poorly extracted. From Figure 5.10(b), it can be seen that the vortex cores were stretched as they were convected downstream, which introduced a non-constant acceleration along the core lines. Visualizations of the vortex cores extracted on this grid as compared to the CFD data set may be seen in Appendix A.

The results from the fine structured mesh may be seen in Figure 5.10(c) and differed dramatically from both other grid results. Near the cylinder, streamwise vortex cores dominated the flow, while the expected spanwise vortex core lines which were conspicuous in the other data sets were missing. One reason that the extraction algorithms failed to extract the spanwise vortex cores was because their strength was much lower than that of the mode B vortex cores. Figure 5.11



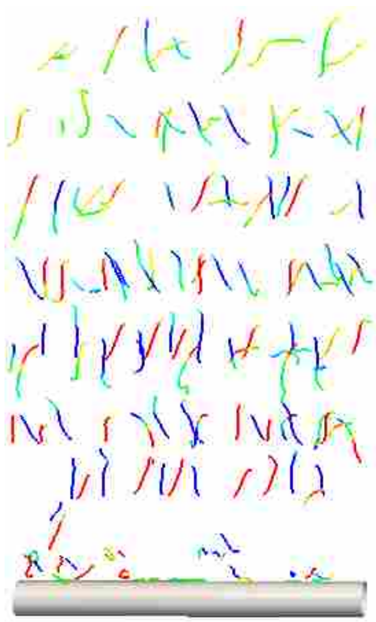
(a) ζ_y in the *xz*-plane shows the strong streamwise mode B vortex cores.

(b) ζ_y in the *xy*-plane shows that the spanwise vortex cores dissipate more quickly.

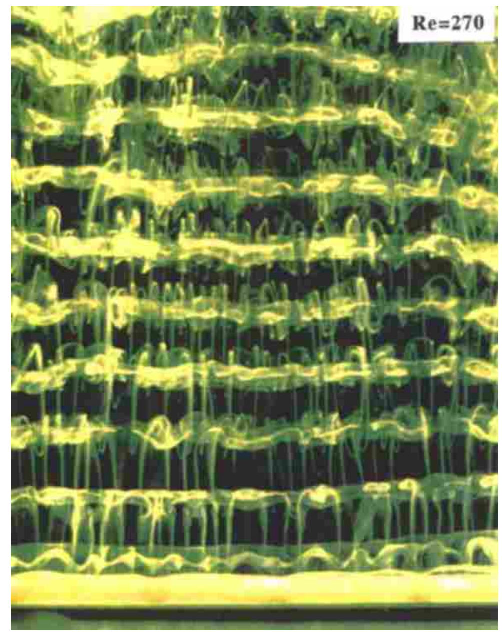
Figure 5.11: Comparative slices of the structured fine CFD data set for the case of cylinder in cross flow. Slices are colored by y-vorticity (ζ_y) on the same scale as shown in (b). Flow is in the $+x$-direction.

shows comparative slices for the structured fine data set. It can be seen in Figure 5.11(a) that the streamwise mode B vortex cores had a high y-vorticity and showed that the strength of the vortex cores was high because of the large regions of ζ_y in the vortex cores. In Figure 5.11(b), the spanwise vortex cores had a lower ζ_y than the streamwise cores and dissipated quickly in the wake. Another reason that the spanwise vortex cores were not extracted was the formulation of the two extractions algorithms: the RP algorithm was designed to detect curved vortex cores, and the SH algorithm was formulated to detect strong vortex cores. Since the spanwise vortex cores were weak and straight, neither extraction algorithm successfully detected the spanwise vortex cores. Last, the success of the coarse structured grid in extracting the spanwise vortex cores may have been due to the fact that the time step was less fine than the fine simulation, which may have resulted in more coherent spanwise vortex cores in the domain.

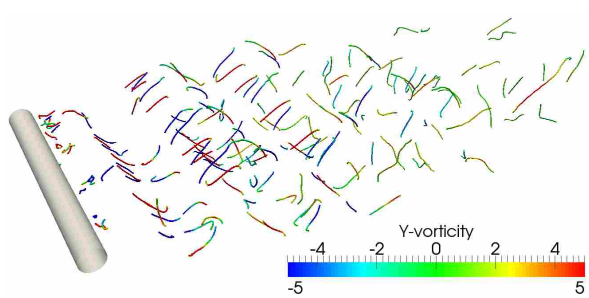
A comparison of the extracted streamwise vortex cores to experimental and DNS flow visualization, as seen in Figure 5.12, verified that the mode B vortex cores extracted from the fine data set agreed well with the physics of the flow. At a Reynolds number of $Re_D = 300$, mode B vortex shedding has been shown to dominate the flow as the wake transitions to a 3-dimensional flow, which can be seen by the extracted vortex cores. The vortex cores in Figures 5.12(c) and 5.12(a) were comprised mostly of the mode B vortex structures, with a lack of the longer spanwise vortex structures that can be seen in Figure 5.12(b). Another aspect of the correctness of the extracted vortex cores was the distance between counter-rotating vortex pairs. The ζ_y of the vortex core lines in Figures 5.12(c) and 5.12(a) showed that the immediately adjacent vortex cores in the near-wake region had opposite vorticity, meaning they were counter-rotating vortex pairs. Williamson [68]



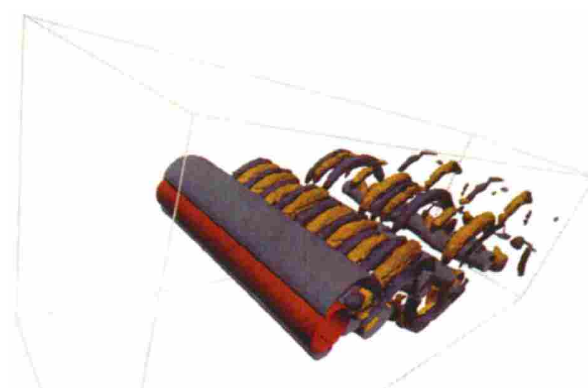
(a) Vortex cores extracted by RP algorithm.



(b) Experimental results from Williamson [68].



(c) Vortex cores extracted by RP algorithm.



(d) DNS results from Thompson et al. [71]. Reynolds number is 285.

Figure 5.12: Comparison of Mode B vortex cores extracted from fine mesh to DNS and experimental results. Extracted vortex cores are colored by ζ_y.

reported that the wavelength between streamwise vortex core pairs was roughly $1D$, which was qualitatively shown by the vortex cores in Figure 5.12(a). The curvature and breakup distance of the vortex cores shown in Figure 5.12(c) also compared well to the ζ_y isosurfaces from the DNS results presented in Figure 5.12(d).

The remainder of the vortex core analysis for the cylinder data set will be made with the fine mesh data set, since it was felt to reflect the physics of the CFD data set most correctly.

Table 5.4: Results of extracting vortex cores from the cylinder data set using different time step widths.

Algorithm	Δt	Number of Vortex Cores
Roth-Peikert	0.01	175
	0.02	169
	0.05	159
	0.10	135
Sujudi-Haimes	0.01	30
	0.02	30
	0.05	34
	0.10	65

5.2.2 Effect of Time Step Width on Vortex Core Extraction

The effect of time step width between data sets on vortex core extraction was investigated. Different time step widths were used to extract vortex cores: 0.01, 0.02, 0.05, and 0.10 seconds. For example, with a time step width of 0.05 s, feature extraction was performed every 0.05 s, with a corresponding time derivative computation for CFD data sets with a spacing of 0.05 s. Table 5.4 shows the total number of vortex core lines using the different time step widths for the same time in the data set so that vortex cores at the same time step could be compared. At this time step, 156 vortex cores were expected in the simulation. This number was found by computing a ζ_y isosurface and counting the number of expected vortex cores. The RP algorithm acted as expected – as the time step width increased, the number of extracted vortex cores decreased, with the largest decrease between time steps widths of 0.05 and 0.10. It was also observed that the vortex cores eliminated as time step width increased were those which were most believable – the mode B vortex cores in the cylinder near-wake. Fuchs et al. [42] reported a similar result that as time step width increased between data sets, the time derivative computation became less accurate, thus reducing the number of believable vortex cores while increasing the number of spurious cores. However, the SH algorithm behaved in the opposite of the RP algorithm, where as the time step width increased, the number of extracted vortex cores increased.

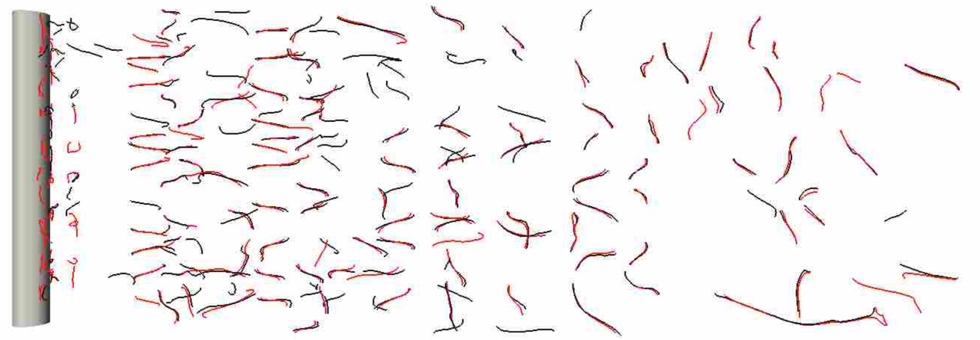
To determine why the RP and SH algorithms acted so differently, the vortex cores extracted using different time step widths were overlaid and visualized. The vortex cores extracted using time step widths of 0.01 and 0.10 seconds can be seen in Figure 5.13. The vortex cores extracted by the RP algorithm using different time step width, as shown in Figure 5.13(a), were quite similar, with only a small shift in the extraction between many of the vortex cores. Using a larger time step width did result in the extraction of more vortex cores near the cylinder surface, which were verified to be spurious. Also, a significant number of $\Delta t = 0.01s$ vortex cores (black) were not detected with the larger time step width.

Viewing the results of the time step width study with regard to the SH algorithm revealed why it extracted more vortex cores as time step width increased. In Figure 5.13(b), it is much easier to tell the difference between $\Delta t = 0.01s$ vortex cores (black) and $\Delta t = 0.10s$ cores (red) than in the RP data set. The $\Delta t = 0.10s$ vortex cores extracted by the SH algorithm exhibited much less curvature and were generally longer than the $\Delta t = 0.01s$ cores. After considering the assumption made by the SH algorithm of a linear flow field, it made sense that increasing time step width would increase the success of the SH algorithm in detecting vortex core lines. As time step width increased, the temporal resolution and thus the curvature of the wake vortex street decreased; therefore, the SH algorithm detected more vortex cores in the lower curvature velocity field. However, though this may lead to the conclusion that data needs to be saved less frequently, the longer, low curvature vortex cores extracted by the SH algorithm using $\Delta t = 0.10s$ were spurious.

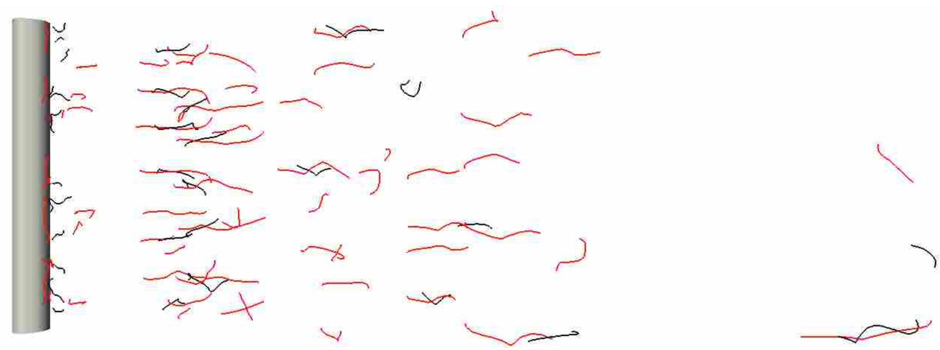
Both extraction algorithms showed in different ways that time step width was an important factor when extracting unsteady vortex core lines. When saving unsteady CFD data sets for use in feature extraction, the number of time steps between saved CFD data sets must be chosen carefully and tailored to each simulation. In simulations where features are expected to move significantly or the flow moves at a high velocity, the time step width will likely become more important when extracting unsteady vortex core lines.

5.2.3 Feature Tracking Results

Feature tracking was performed on the cylinder data set and cores were tracked through time. 502 time steps were selected for analysis, so 500 extraction steps were performed due to the computation of central-differenced time derivatives. The key results from both Roth-Peikert and Sujudi-Haimes can be seen in Table 5.5. As seen, the RP algorithm extracted almost six times more vortex cores than the SH algorithm. The RP vortex cores were also tracked better than the SH vortex cores and had longer average path length. One interesting note was the effect of increasing



(a) Vortex cores extracted by RP algorithm.



(b) Vortex cores extracted by SH algorithm.

Figure 5.13: Comparison of vortex cores extracted with time step widths of $\Delta t = 0.01$ (black) and $\Delta t = 0.10$ (red). Flow is from left to right.

the number of passes through the data set while relaxing tracking tolerances as presented in Section 3.3.2: few new paths were found, as seen by the small increase in the the total tracking paths. However, increasing the number of passes did have a significant effect on the average path length. Performing multiple passes through the data set helped to extend previously created tracking paths, thus increasing the average feature life.

Though there were features extracted by both algorithms that only existed for 2 or less frames, most of the paths lasted much longer, and some existed for more than 100 time steps, or 20% of the entire data set. Vortex cores were observed to convect from the cylinder to the domain exit in roughly 300 time steps, so some vortex cores were tracked for 30% or more of a vortex core's life in the domain.

A 200 time step portion of the full RP data set was considered, and vortex cores which existed for more than 100 time steps in the smaller data set are shown in Figure 5.14. The longest-tracked vortex core existed for 178 time steps and was tracked from the mid-wake of the cylinder

Table 5.5: Vortex core extraction & tracking results from the cylinder data set.

Tracking parameters	Roth-Peikert		Sujudi-Haimes	
	1 Pass	10 Passes	1 Pass	10 Passes
Vortex cores	83,582		14,180	
Tracking paths	12,346	12,362	2,393	2,413
Untracked features	6,985	3,116	1,582	860
% features tracked	91.6	96.3	88.8	93.9
Average path length	12.2	19.1	6.2	12.2

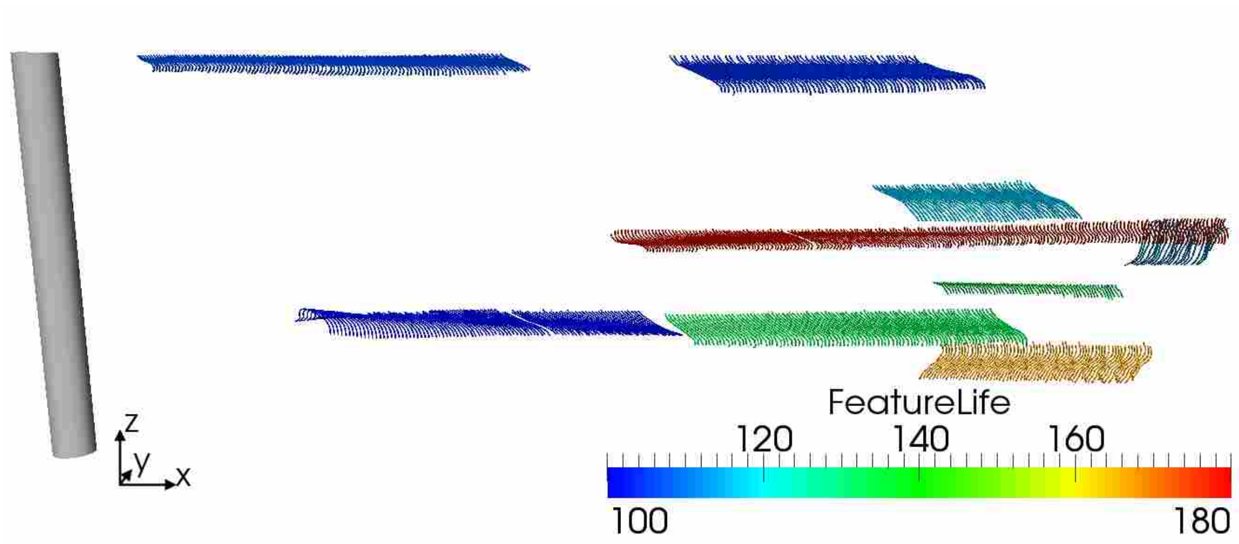


Figure 5.14: Paths of RP vortex cores which existed for more than 100 time steps of a 200-time step portion of the data set.

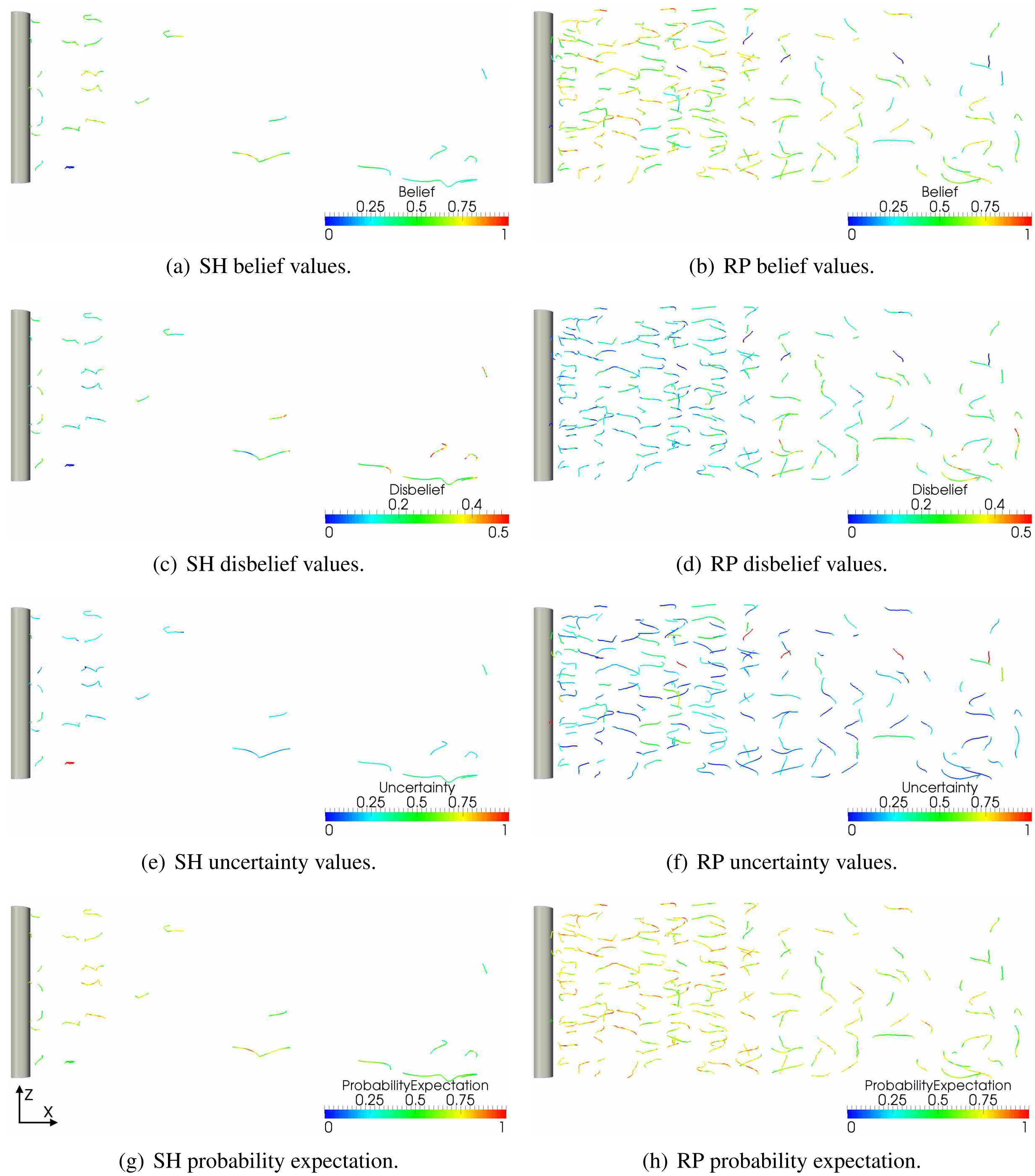
to the domain exit. In this data set, where the time step was quite small, many vortex cores were quite predictable, which increased the success of the feature tracking step. The attribute tracking method allowed for quick correlation and viewing of a relatively complex data set which would have been difficult to follow without the tracking paths to understand feature movement.

5.2.4 Vortex Cores Processed by Intelligent Agents

After feature extraction and tracking were accomplished, subjective logic was applied to define the opinion of the data set. The belief, disbelief, uncertainty, and probability expectation of the vortex cores were computed for each time step, and one representative time step can be seen in Figure 5.15. A comparison of the vortex cores extracted by the SH and RP algorithms showed that the SH algorithm performed poorly in this data set, extracting only some of the mode B vortex cores and other spurious cores, while the RP algorithm extracted most of correct near-wake vortex street as well as a number of other spurious vortex cores in the far wake.

The belief and disbelief values of the two data sets confirmed the RP algorithm's suitability in this flow situation and the SH algorithm's weakness in curved wakes. As shown in Figure 5.15(a), higher belief was calculated for the SH vortex cores near the cylinder, which was due to the higher vortex strength of the cores and relatively long feature life. However, most of the SH vortex cores had low belief due to the higher curvature of the vortex cores in the cylinder wake. The vortex cores extracted by the RP algorithm had a large range of belief values, as shown in Figure 5.15(b). This simulation was well-suited to the strengths of the RP algorithm, which include high curvature, moderate vortex strength, and low quality. As expected, the vortex cores in the near wake had higher belief ($0.7-1$) than those in the far wake, which had computed belief values of approximately 0.25 to 0.5. The disbelief of the the RP vortex cores, as seen in Figure 5.15(d), acted similarly, with low disbelief in the near wake with increasing disbelief for the vortex cores in the far wake.

An analysis of the various characteristics was made to understand why the AA belief tuple was computed as it was. A visualization of the characteristics that defined AA belief, disbelief, and uncertainty of the vortex cores can be seen in Figure 5.16. As expected, $VortexStrength$, shown in Figures 5.16(b) and 5.16(a), was high for both algorithms near the cylinder and declined further in the cylinder wake. The vortex cores near the cylinder had curvature higher than $CurvatureNorm = 25$, as seen in Figures 5.16(d) and 5.16(c). This in effect increased the RP vortex cores' belief while decreasing the SH vortex core belief near the cylinder. In other areas of the wake, segments of the SH vortex cores had lower curvature, which increased the expected probability in those small segments of the vortex cores, as seen in Figure 5.15(g). The quality values shown in Figures 5.16(f) and 5.16(e) demonstrated that calculation of quality in unsteady data sets generally resulted in higher quality values than was acceptable in steady-state data sets. This was due to the calculated vortex convection velocity, which was taken as the average velocity at which the vortex core line was moving and may not have been representative of the true vortex convection velocity. When looking at the λ_2 criterion in Figures 5.16(h) and 5.16(g), which influenced AA uncertainty, it was



(a) SH belief values.

(b) RP belief values.

(c) SH disbelief values.

(d) RP disbelief values.

(e) SH uncertainty values.

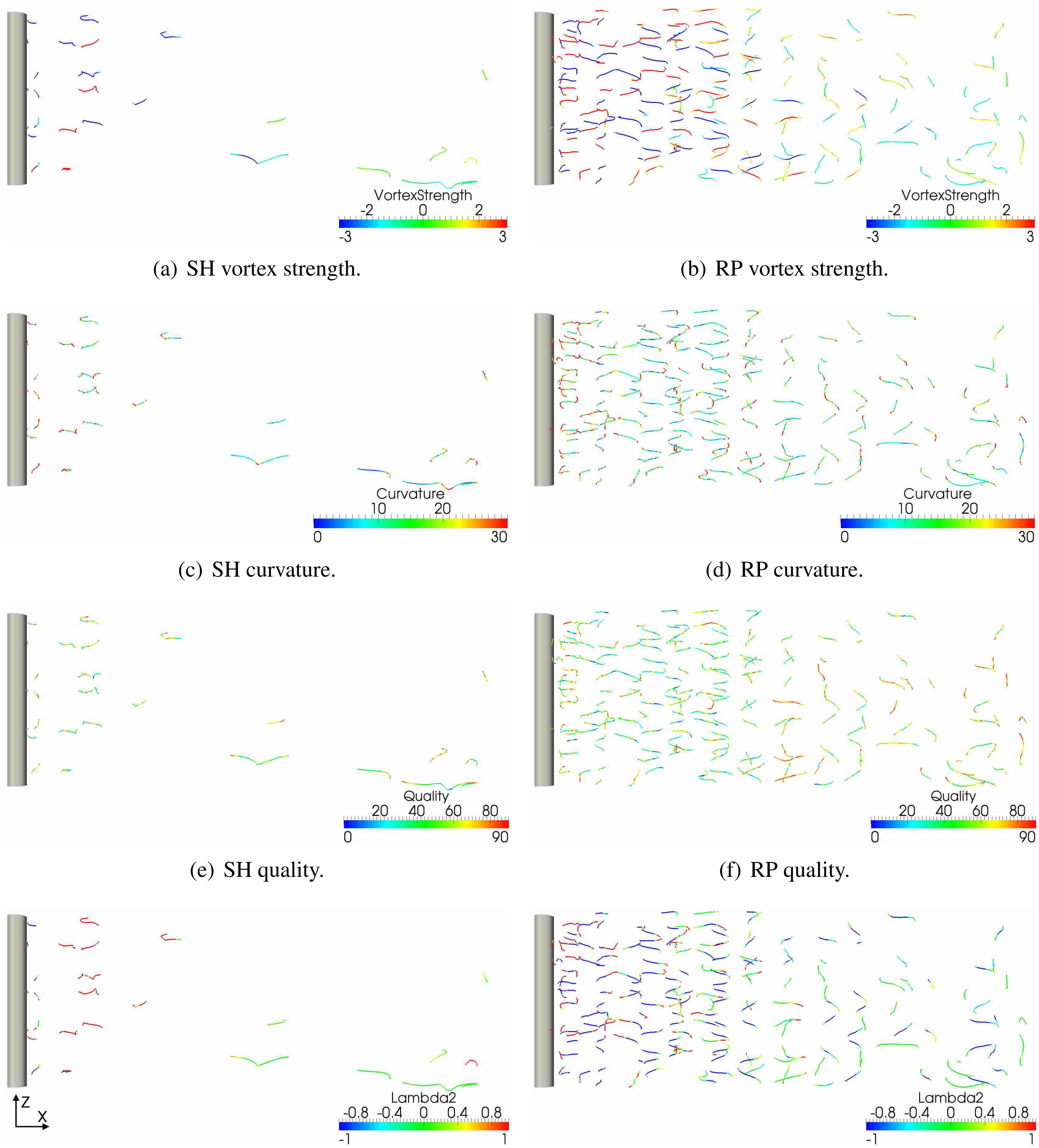
(f) RP uncertainty values.

(g) SH probability expectation.

(h) RP probability expectation.

Figure 5.15: Opinion calculated on vortex cores extracted from one time step of the cylinder data set. Flow moves from left to right.

observed that the RP algorithm extracted vortex cores which more closely agreed to the criterion, while the SH algorithm generally failed to extract vortex cores which satisfied $\lambda_2 < 0$. This resulted in a lower uncertainty for the RP vortex cores and a higher uncertainty for the SH vortex cores.



(a) SH vortex strength. (b) RP vortex strength.

(c) SH curvature. (d) RP curvature.

(e) SH quality. (f) RP quality.

(g) SH λ_2. (h) RP λ_2.

Figure 5.16: Cylinder data set vortex cores colored by characteristics defining the belief tuple. Flow moves from left to right.

The uncertainty of the vortex cores was seen to be a strong function of the feature tracking method, with a lesser influence from the λ_2 criterion. The vortex cores with $u = 1$ in Figures 5.15(e) and 5.15(f) had high uncertainty because the lines were not tracked at all in either direction, which resulted in a line correspondence, *Corr*, of less than -1. The MA uncertainty was based on *Corr*, as shown in Section 4.3, so the final uncertainty of the vortex core was 1 when a line was untracked in both directions in time.

Feature tracking was observed to have a greater impact overall than any of the other individual characteristics which were used to define the agent opinions. This was due to the fact that the MA opinion, which has the largest agent influence on the final opinion, was formulated using feature tracking parameters. When a feature was poorly tracked, it contributed to a generally lower opinion for the vortex core. This situation occurred even if a certain attribute, such as vortex strength, contributed to a high belief in the vortex core. However, when vortex cores exhibited all the strengths of a certain algorithm other than feature tracking, the final opinion was not as dependent on the feature tracking results.

The automated feature set combination method was applied to the cylinder data set and helped reduce some of the spurious and weak vortex cores from both data sets. The combined feature set may be seen in Figure 5.17. The RP algorithm was the dominant extraction algorithm in the cylinder data set, which was reflected in the vortex cores of the final data set – only 2 vortex cores at the time step shown in Figure 5.17 were extracted by the SH algorithm. Also, since the two algorithms did not extract vortex cores in the same location in most cases, the duplicate check did not result in the removal of vortex cores from either of the vortex core data sets. Another item of note was that in most time steps, many of the vortex cores that had been extracted by the RP algorithm in the far wake were eliminated due to the fact that they had very low vortex strength, λ_2 values greater than 0, and were mostly poorly tracked.

5.2.5 Visualization of CFD Data Set Vortex Physics

The main goal of feature extraction is to provide a clear and simple representation of the flow domain which also allows for visualization of massive data sets on a local workstation. The agent-based method presented here is also a good tool for visualizing the vortex physics of a CFD data set. One common vortex visualization method is finding isosurfaces of ζ_y, as shown in

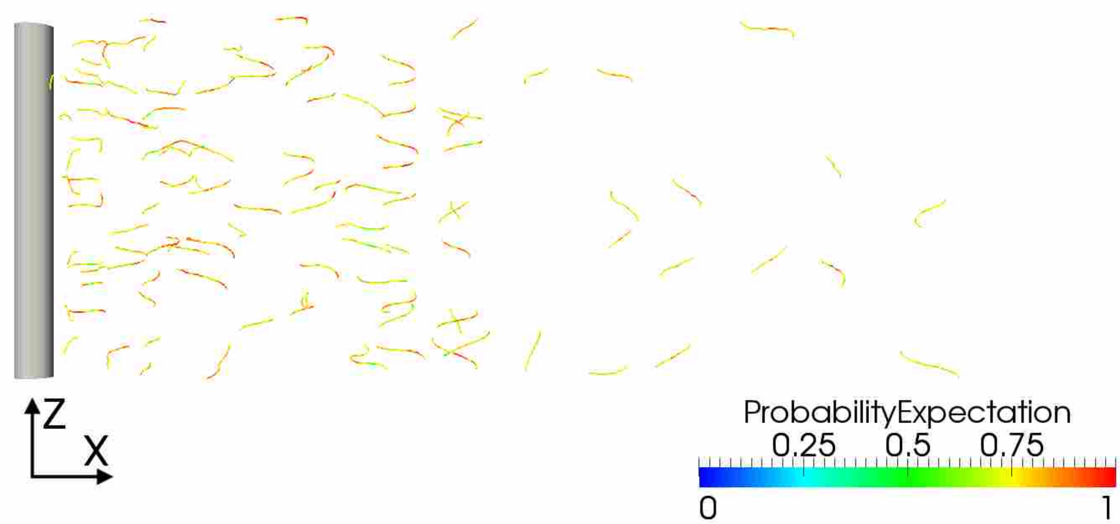
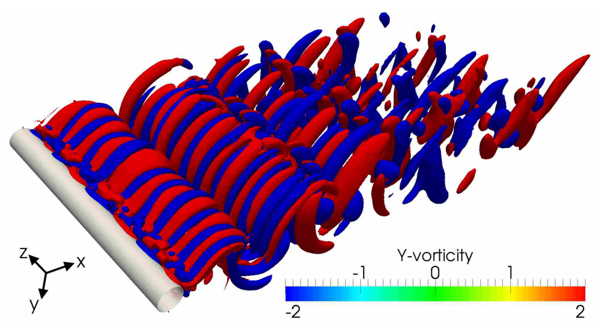
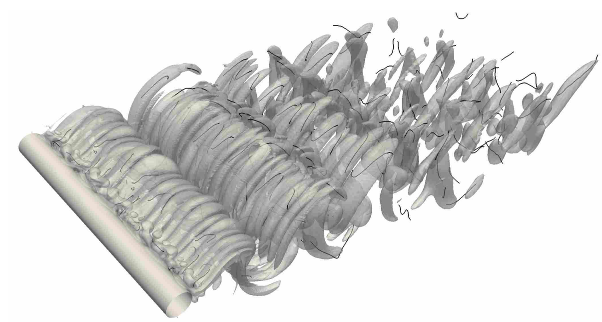


Figure 5.17: Final vortex core data set which was generated using the feature set combination method.



(a) Isosurfaces of y-vorticity ($\zeta_y = -2$ and 2).



(b) RP vortex cores shown with ζ_y isosurface.

Figure 5.18: Visualization of the wake in the cylinder data set. The visualization of vortex core lines provides a clear method for understanding the physics of the flow.

Figure 5.18. The ζ_y isosurface in Figure 5.18(a) clearly shows the mode B vortex shedding and the 3-dimensional vortex breakup in the far wake, but it is difficult to visualize some of the finer details because of the visual clutter produced by isosurfaces. Other issues with isosurfaces include choosing the correct isosurface value as well as indistinct delineation between vortex regions. In Figure 5.18(b), the addition of the vortex cores extracted by the RP algorithm showed that feature extraction agreed with the ζ_y isosurface and resulted in a simpler visualization of the physics in the wake of the cylinder.

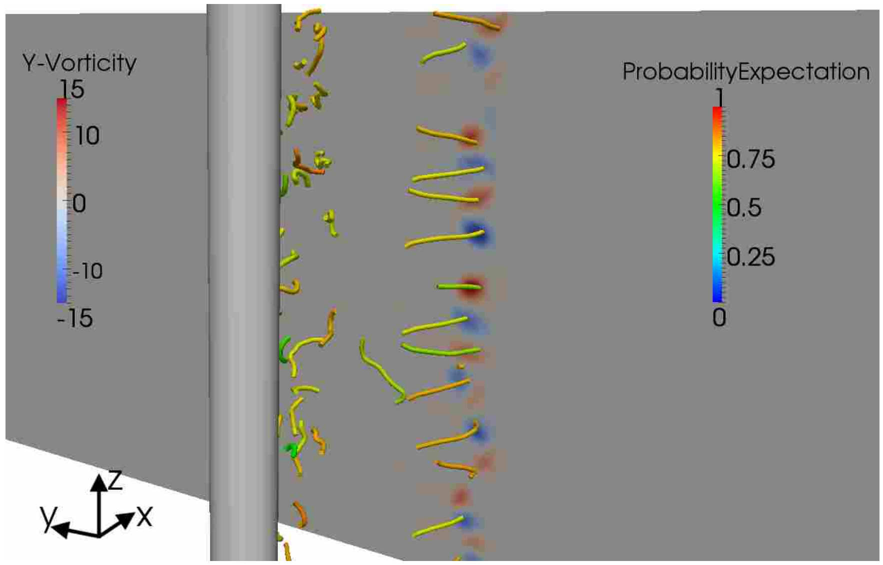


Figure 5.19: Slice of the CFD data set colored by ζ_y along with vortex cores from the RP data set colored by probability expectation.

The use of subjective logic to find the opinion of the extracted vortex cores further assisted in a determination of the vortex physics of CFD data sets. A slice of the CFD data set which bisects a row of vortex cores can be seen in Figure 5.19. Most of the vortex cores bisected by the slice had high probability expectation and agreed well with the centers of high ζ_y. The vortex cores that did have low probability expectation were shown to be shifted from the centers of the swirling flow. One key application of this tool is to find vortex cores in the data set with high expected probability, then utilize other visualization methods such as slices or isosurfaces to explore the flow physics in that region in further depth.

5.2.6 Effects of Changing Subjective Logic Equation Constants

In Chapter 4, the equations defining agent belief, disbelief, and uncertainty were shown to be a first-order model with two constants defining the line in the form of Eq. 4.7. Many of the constants m_1 and m_2 were created by Mortensen [17] for the steady-state feature extraction method and were also used in the unsteady method. To determine the effect of the constants on the final opinion of a data set, the constants were changed and the subjective logic was calculated for the same vortex core data set. One time step from the cylinder data set was used to calculate the average probability expectation of all vortex cores in the time step ($\overline{E}$) due to the change in constants.

Table 5.6: Original constants in subjective logic b,d,u equations.

	MA			RP_E			RP_{NE}			SH_E			SH_{NE}		
	b	d	u	b	d	u	b	d	u	b	d	u	b	d	u
m_1	0.5	-0.5	1.0	0.6	-0.4	0.5	0.8	-0.8	1.0	0.6	-0.4	0.5	0.8	-0.8	1.0
m_2	0.5	0.5	5.0	0.4	0.4	-10	0.2	0.8	0.0	0.4	0.4	-10	0.2	0.8	0.0

Many runs were conducted where m_1 and m_2 for each equation were changed simultaneously. For example, in one run the two constants that defined the Roth-Peikert extracting agent (RP_E) belief, m_{1,b,RP_E} and m_{2,b,RP_E}, were changed while all other constants were kept as the original values and the new $\overline{E}$ was calculated. The constants were changed to reflect the behavior of the original constants, i.e. if the sum of m_1 and m_2 was 1, the new constants also summed to 1.

32 runs were conducted in which the constants of each equation used in subjective logic were changed from the original values shown in Table 5.6. The constants in each run were chosen to provide a wide range of values in order to find the effects of the constants for different extremes. For example, the original constants $m_{1,b,MA}$ and $m_{2,b,MA}$ were 0.5 and 0.5, respectively, so two runs were made which changed the constants to 0.9 and 0.1, and 0.1 and 0.9. The results of the study can be seen in Table 5.7 and Figure 5.20. Generally, changes in the constants which defined belief resulted in more change in probability expectation than did those that defined disbelief and uncertainty.

Changes in the constants used in the MA opinion resulted in a similar change for the opinions of SH and RP vortex cores. In Run 1, changing the MA belief constants resulted in a 20% and 15% change in the RP and SH $\overline{E}$, respectively. The MA disbelief constants in Run 4 also resulted in a significant $\Delta\overline{E}$ of 4% and 8% for RP and SH, respectively. Changes in the MA uncertainty constants resulted in the most significant change in terms of uncertainty equation constants, with $\Delta\overline{E}$ as high a 7% recorded.

The belief constants for the AA_E had the most impact on the vortex cores which the AA_E extracted. This can be seen by the results of Runs 9, 21, and 22 in Figure 5.20, where $\Delta\overline{E}$ was as high as 16.5%. The disbelief and uncertainty constants in the AA_E equations had a negligible effect on the final opinion of the vortex cores.

Table 5.7: Subjective logic b, d, u equation constants study.

Run	Agent	Tuple	m_1	m_2	$\overline{E}_{\text{RP}}$	$\Delta\overline{E}_{\text{RP}}$ (%)	$\overline{E}_{\text{SH}}$	$\Delta\overline{E}_{\text{SH}}$ (%)
0	**Original**	–	–	–	**0.658**	–	**0.571**	–
1		*b*	0.9	0.1	0.526	19.99	0.484	15.14
2			0.1	0.9	0.693	5.26	0.565	0.91
3		*d*	-0.9	0.9	0.621	5.65	0.547	4.09
4	MA		-0.1	0.1	0.628	4.52	0.523	8.41
5		*u*	0.5	5.0	0.679	3.26	0.572	0.26
6			0.1	5.0	0.707	7.47	0.575	0.80
7			1.0	10.0	0.656	0.36	0.559	2.07
8		*b*	0.9	0.1	0.634	3.58	0.570	0.15
9			0.1	0.9	0.723	9.94	0.579	1.40
10		*d*	-0.9	0.9	0.651	1.09	0.569	0.20
11	RP_{E}		-0.1	0.1	0.651	1.01	0.562	1.48
12		*u*	0.5	-5.0	0.646	1.82	0.569	0.35
13			0.9	-5.0	0.661	0.44	0.555	2.66
14			0.1	5.0	0.669	1.72	0.578	1.30
15		*b*	0.5	0.5	0.654	0.67	0.592	3.80
16			0.2	0.8	0.659	0.18	0.585	2.44
17			0.4	0.6	0.662	0.66	0.576	0.99
18	RP_{NE}	*d*	-0.1	0.1	0.661	0.51	0.674	18.07
19			-0.5	0.5	0.659	0.10	0.612	7.33
20		*u*	0.5	0.5	0.668	1.53	0.582	2.01
21		*b*	0.9	0.1	0.672	2.06	0.512	10.27
22			0.1	0.9	0.667	1.31	0.665	16.52
23		*d*	-0.9	0.9	0.655	0.48	0.553	3.11
24	SH_{E}		-0.1	0.1	0.667	1.35	0.573	0.38
25		*u*	0.5	-5.0	0.651	1.09	0.575	0.77
26			0.9	-5.0	0.673	2.22	0.571	0.02
27			0.1	-5.0	0.671	2.02	0.551	3.44
28		*b*	0.1	0.9	0.673	2.31	0.559	2.07
29			0.5	0.5	0.657	0.09	0.574	0.66
30	SH_{NE}	*d*	-0.5	0.5	0.699	6.19	0.571	0.01
31			-0.1	0.1	0.725	10.21	0.576	1.03
32		*u*	0.5	0.5	0.664	0.92	0.569	0.26

For the AA_{NE}, the constants in the disbelief equations affected the vortex cores which the AA_{NE} did not extract. This was most shown in Runs 18 and 31, where $\overline{E}$ changed by 10% and 18% for the RP and SH vortex cores, respectively. For the AA_{NE}, the belief and uncertainty constants

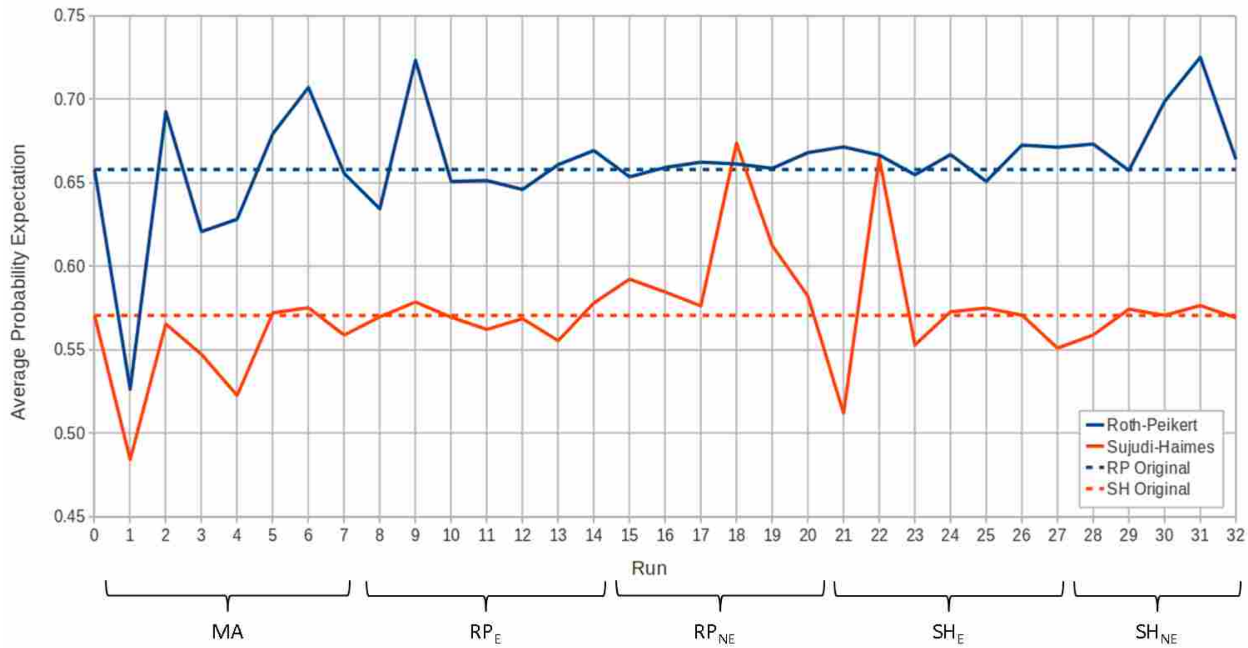


Figure 5.20: Results from the subjective logic equation constants study.

were observed to have a negligible effect on the vortex core opinion, similar to what was observed for the AA_E.

In summary, the final opinion of extracted vortex cores was most sensitive to changes in the following constants: MA belief and disbelief, AA_E belief, and AA_{NE} disbelief. Changes in other constants, especially the AA_E and AA_{NE} uncertainty, showed the insensitivity of the opinion by changing these constants. An understanding of the sensitivity of certain constants helps for future improvement of the equations defining subjective logic. One idea for improving the opinion computation is to optimize the constants described here so that the opinion more correctly reflects the belief through all data sets. This might be accomplished by applying the method on data sets with known vortex cores and altering the constants until a correct opinion has been computed through the different data sets.

5.3 Wind Turbine

A simulation of a wind turbine was obtained to test the method's effectiveness in massive CFD data sets. The data set was a simulation of the NREL Phase VI two-blade wind turbine [72]. The simulation was run in OVERFLOW-D [73], a NASA CFD flow solver which utilizes overset

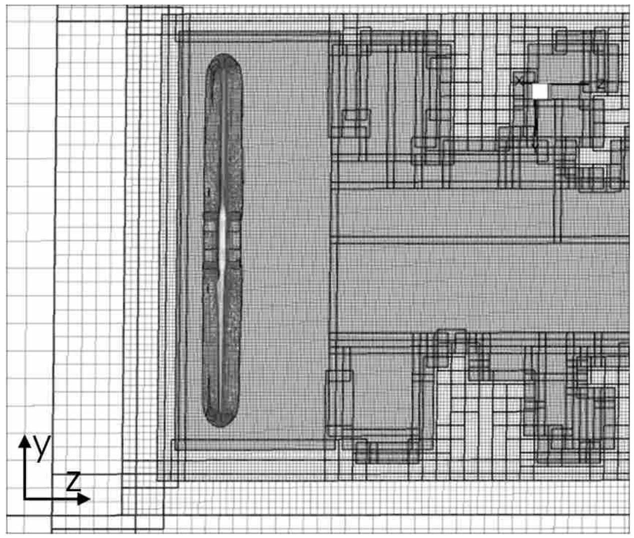


Figure 5.21: Near-wake slice of the computational mesh used in the wind turbine simulation.

grids to solve the Navier-Stokes equations. An adaptive overset mesh, which contained roughly 30 million mesh nodes per time step, was used where refined blocks were inserted in areas of interest. A representative slice of the overset mesh near the turbine blade is shown in Figure 5.21. More details on the simulation are presented by Duque et al. [74].

The simulation was run to convergence and 360 time steps were saved, which corresponded to 1 time step per degree of blade revolution. In order to operate on a massive data set, the feature extraction and tracking method was compiled on the local BYU supercomputer to fulfill the memory requirements of the method. Feature extraction and tracking were performed on each time step, and subjective logic was applied to compute the opinion of the vortex cores.

5.3.1 Computational Requirements of Method

Table 5.8 shows the results of the different steps taken on the wind turbine data set in terms of data size, memory, and processing time. Here it can be seen that the method developed reduced the data size by 2 to 3 orders of magnitude, which allowed for quick visualization on a desktop workstation instead of a computationally expensive visualization cluster. However, the amount of required memory and processing time per processor for the extraction step was roughly 11 and 80 times greater than that of the actual CFD simulation, respectively, because it could only be run on one processor. Feature extraction required a large amount of memory because 3 different time steps

Table 5.8: Vortex core extraction and tracking results from the wind turbine data set. Memory and time requirements are shown per processor.

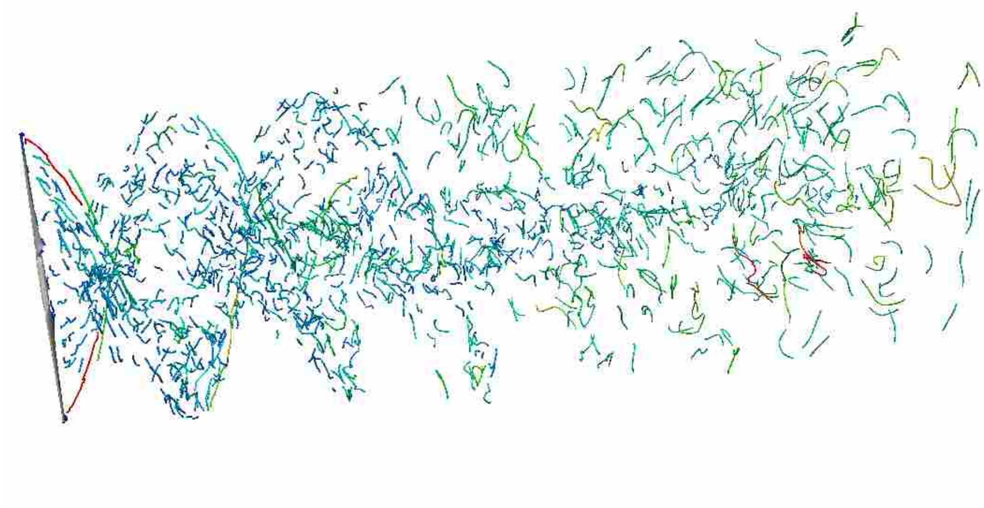
	Data size (MB)	Processors	Memory (MB)	Wall time (hr)
CFD data set (per time step)	3000	162	2000.0	0.03
Extraction (per time step)	2.2-20.0	1	23000	2.5
Tracking (5 passes)	N/A	1	90	10.0
Opinion (full data set)	2.5-27.0	1	50	4.0

were read into memory simultaneously for computation of time derivatives. The requirements of feature extraction showed the need for this step to become parallelized so that feature extraction could be run on the same processors as the CFD simulation and thus keep up with the simulation as it runs.

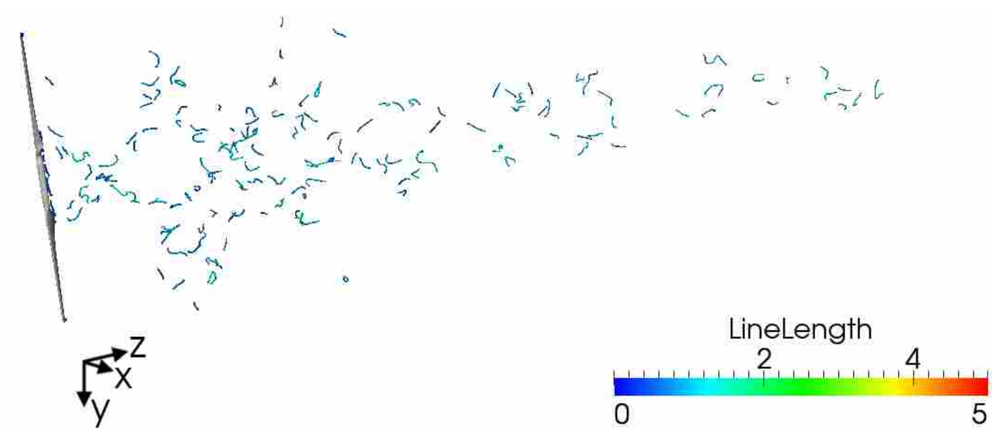
Because of the nature of the different steps of the method, feature extraction was the only step that could be performed in real time as the simulation was running. The other steps of feature tracking and subjective logic required a series of extracted feature sets to be able to work. However, those steps were negligible in terms of memory and processing time when compared to the feature extraction step. To reduce the post-processing time, one could extract vortex cores as soon as three consecutive CFD time steps have been written out, then track and compute the opinion of the vortex cores as soon as the desired number of feature sets have been obtained. For example, consider a simulation which will be run for 5000 time steps. When the simulation reaches the 1000$^{\text{th}}$ time step, feature extraction could be run for 100 time steps concurrent to the simulation. After the vortex cores have been extracted, the feature tracking and agent opinion steps would then be run so that the analyst could view the results of the simulation while the simulation is still running.

5.3.2 Discussion of Extracted and Tracked Vortex Cores

The vortex cores extracted by the SH and RP algorithms displayed a similar trend as in the cylinder data set, as seen in Figure 5.22. The RP algorithm again extracted many more vortex cores than the SH algorithm – over 360 time steps, the RP algorithm extracted 455,000 vortex cores, while the SH algorithm extracted 56,000 vortex cores. The RP algorithm extracted the noticeable tip vortices on both blades as well as many other vortex cores in the turbine wake. The SH algorithm mainly extracted short vortex cores which were mostly confined to the root of the



(a) Vortex cores extracted by RP algorithm.



(b) Vortex cores extracted by SH algorithm.

Figure 5.22: Vortex cores extracted from the wind turbine data set at 1 time step. Both data sets are colored by vortex core line length. Flow moves in the $+z$-direction.

wind turbine. The location where the tip vortices dissipated and broke up into less coherent vortex cores was shown to be roughly 1.5 blade diameters downstream of the wind turbine, as seen in Figure 5.22(a).

Some challenges were encountered while extracting vortex cores from unsteady data sets with an adaptive mesh such as the wind turbine data set. Because of the adaptive mesh utilized in the CFD simulation, time derivatives were not calculated for most of the domain. Recall that time derivatives were computed using the i^{th} node point in a mesh in three separate time steps. However, in an adaptive mesh, the i^{th} node point in a certain time step does not correspond to the

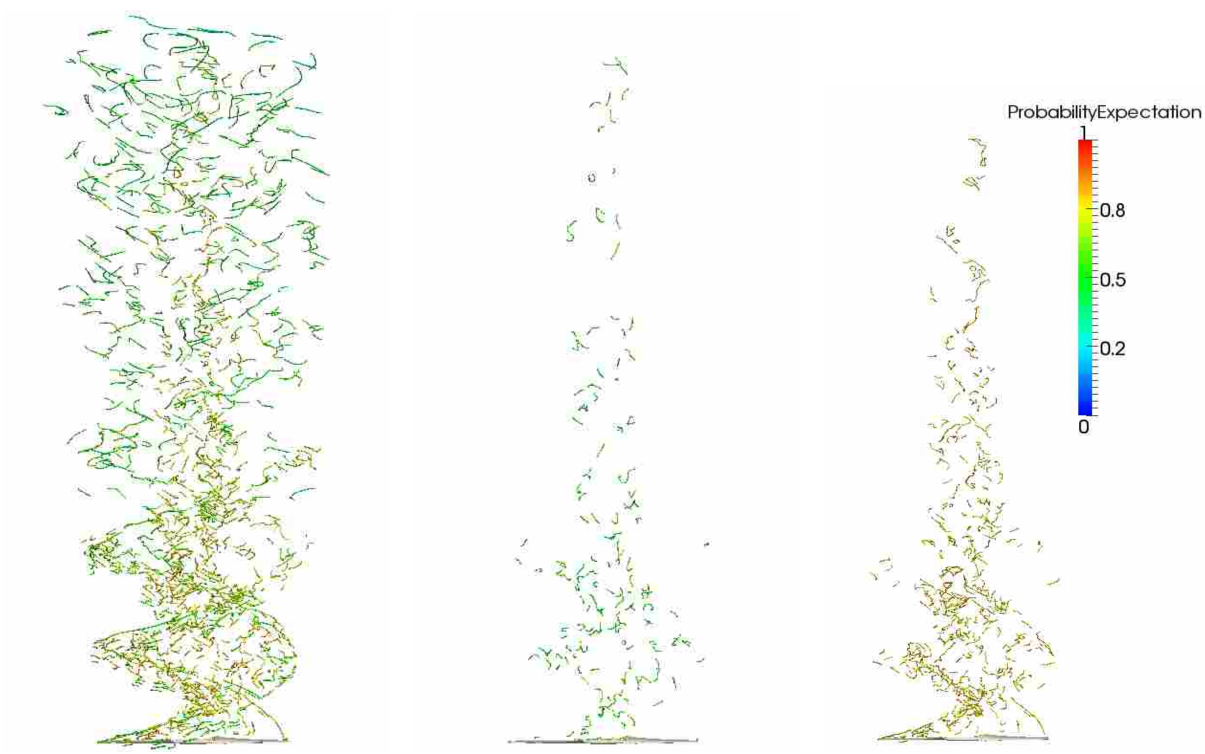
i^{th} node point in another time step, so time derivatives were not computed and the vortex cores were extracted using a steady-state assumption. In the wind turbine data set, the mesh blocks near the turbine blade were not adapted over time, so time derivatives were computed in these blocks, where most of the tip vortex cores were contained. The same difficulty would be encountered in a data set with a moving mesh. One option for computing time derivatives from these types of data sets would be to calculate time derivatives at physical coordinates in the domain instead of at mesh nodes. The time derivative field could then be interpolated onto the mesh nodes so that unsteady extraction might be accomplished.

Another challenge in feature extraction was due to the overset mesh of the wind turbine data set. Overset meshes were created in such a way that most of the domain in the wake of the wind turbine was a combination of overlapping coarse and fine meshes. Vortex cores were extracted from each block of the data set, where there were roughly 1,500 blocks in each time step. The vortex cores from each block were then combined into one final set. It was observed that some vortex cores which had been extracted through multiple blocks were disconnected at the block edges, and some vortex cores were duplicated because they had been found in overlapping blocks. To fix this, one could convert the whole data set into one unstructured mesh and remove duplicate mesh nodes, then perform feature extraction. This was performed and it was observed that the feature extraction from the unstructured mesh required much more time and memory than from the blocks of the structured data set.

Feature tracking in the wind turbine data set showed the success of the efficient search method outlined in Section 3.3.3. In this data set, there were roughly 1,200 vortex cores per time step, a number at which exhaustive search through the data set became prohibitive. Without use of the search method, one pass of the feature tracking was incomplete after 100 hours of run time. With the efficient search method in place, five passes of feature tracking were performed in roughly 10 hours. With a faster tracking time, the post-processing of the data set was expedited in a more timely manner.

5.3.3 Vortex Cores Processed by Agents

The opinion of the extracted and tracked vortex cores was calculated and is shown in Figure 5.23. As seen, the RP algorithm was the dominant extraction algorithm in the wind turbine data set. This made sense because the incoming flow was fairly low speed (13 m/s) and the wake of the turbine was highly curved. This resulted in curved, low strength vortex cores. However, the RP algorithm also extracted many spurious vortex cores in the far wake of the data set, similar to what happened in the cylinder data set. These spurious vortex cores were assigned low belief and therefore also had low expected probability. The vortex cores extracted by the SH algorithm were assigned low belief due to the nature of the data set, though some of the SH vortex cores near the root of the turbine blade were passed into the final feature set. The final feature set, as shown in Figure 5.23(c), showed the tip vortex cores near the turbine blade as well as the root vortex wake which extended further downstream. The creation of the final feature set allowed for viewing of the key vortex structures in the wake of the wind turbine without the noise created by the RP algorithm in the far wake.



(a) RP vortex cores. (b) SH vortex cores. (c) Final vortex core data set.

Figure 5.23: Probability expectation of wind turbine vortex core data sets. Flow moves from bottom to top.

CHAPTER 6. RECOMMENDATIONS FOR FUTURE WORK

This chapter gives general recommendations regarding the extension of unsteady feature extraction and tracking to features other than vortex core lines. Also presented are topics for future research regarding vortex core lines and the application of subjective logic to CFD data mining.

6.1 General Unsteady Feature Extraction & Tracking

Currently, only vortex core line extraction algorithms have been modified to correctly extract vortex core lines from time-dependent CFD data sets. Two other features which were researched by Lively [75] were shock waves and separation and attachment lines. These features were extracted from steady-state data sets and subjective logic was applied to compute the opinion of the features. Future work should investigate transient modifications to these extraction algorithms and the effect of the modifications on the extracted features.

Feature tracking is another aspect of the unsteady trust network that that would require attention in different types of features. The attribute-based feature tracking implemented here required line-type features as input, and different attributes would be required for different types of features. For example, if it was desired to track a volume-type feature, attributes such as volume and orientation might be used, as suggested by Reinders et al. [11]. Different tracking methods have been created for specific types of features and might be implemented in the general unsteady feature extraction method so that different features might be successfully tracked.

It was shown in Section 5.3.1 that feature extraction took much longer and required more memory per processor than the actual CFD simulation. Due to the architecture of the feature extraction method it was not possible to run feature extraction on multiple processors, which increased the difficulty of running the extraction on large data sets. In order to reduce extraction time, the code would need to be parallelized so that feature extraction could be run on the same processors as a CFD simulation while the simulation is running.

6.2 Vortex Core Line Extraction & Tracking

Two feature extraction algorithms were used in this research to show the feasibility of utilizing a trust network to detect believable features in unsteady CFD data sets. As shown in Chapter 2, many vortex core extraction algorithms have been and continue to be developed, especially for use in unsteady flow situations. Any extraction algorithm could be utilized into the trust network with a knowledge of its strengths and weaknesses. Future research should look into the effect of employing multiple vortex core extraction algorithms in the trust network.

The attribute-based tracking method used in this research was shown to be quite effective at tracking vortex core lines through an unsteady CFD data set, but would require additional work to become more robust throughout data sets. The $Position$ tolerance was very simulation-dependent, since length scales vary widely in different CFD simulations. One idea for future work would be to create a parameter which finds an appropriate $Position$ tolerance, perhaps based on a characteristic length of the flow such as hydraulic diameter. Feature event detection was also not implemented in the tracking method. Finding events such as split and merge as shown in Figure 2.7 serves two purposes: increase the feature life of an extracted vortex core and view additional aspects of vortex cores which might aid in a greater understanding of flow physics. Birth and death events may also be found by marking vortex cores which have only been tracked in one direction in time. For the vortex core line extracting and tracking step of the method, this is should be addressed first in order to improve vortex core line tracking through time.

Grid density and mesh type was shown to be an important factor in extracting vortex cores that agreed with the physics of the flow domain. Extraneous vortex cores extracted from coarse meshes sometimes had a high calculated probability expectation because the vortex core satisfied the strengths of the extraction algorithm. Some parameter which describes the grid density, perhaps related to the reference length of the simulation or wall y^+ in turbulent flows, could be created and used to help define the opinion of an extracted vortex core. Another improvement that could be made to the vortex core extraction methods would be a technique to extract smooth vortex core lines from data sets with unstructured meshes. One possible application of the method might be to find vortex cores with high expected probability and use them as input for adaptive mesh refinement.

The data sets considered here were incompressible flows in the laminar or turbulent range, where the turbulence was modeled using RANS. Another area of research would look at the results of feature extraction and tracking from unsteady LES and DNS simulations of different flow domains. Turbulent eddies are partially or fully resolved in these simulations, so future work would determine whether feature extraction methods extract these flow structures as vortex core lines. Also, with the fine meshes and small time steps associated with such simulations, the data reduction would need to be investigated to ascertain whether the method actually helps to detect key vortex structures in such refined simulations. Compressible flows should also be considered, since the study of vortex-shock interactions is one of key interest in many industries, and an understanding of vortex physics in compressible simulations would lend to improved engineering designs.

6.3 Subjective Logic Framework

In Section 2.2.2, both vortex core extraction algorithms used had the same weakness of incorrectly extracting vortex core lines with a non-constant acceleration. This weakness was not implemented in the subjective logic computations, and a better calculation of vortex core belief would likely result from the addition of an acceleration check along vortex core lines extracted by the SH and RP algorithms. Future research would investigate the magnitudes of acceleration along these vortex core lines and the effect of an acceleration parameter in subjective logic.

The λ_2 criterion was used to define vortex core uncertainty in this research. This vortex identification method has been extensively used in a variety of CFD data sets with success, but it has its shortcomings. It can fail to find vortices in rotating frames of reference, was not formulated to be useful in compressible flow, and can declare the whole domain to be a vortex in certain simulations. Some of the criteria presented in Section 2.2.1 may be used in tandem with the λ_2 criterion to define AA_E uncertainty, or other methods such as particle tracing may be used in unsteady CFD data sets to find areas of swirling flow.

The MA opinion was calculated based on a normalized feature life $FeatureLifeNorm$, which was very simulation-dependent and required user input based on the number of time steps a believable feature was expected to exist. This required analysis of the data set in order to select a proper value of $FeatureLifeNorm$, and some method of automation for this parameter would increase the generality of the unsteady trust network. Because feature tracking is closely related to time step, some parameter might be created which correlates the time step to vortex convection velocity or shedding frequency to find an appropriate $FeatureLifeNorm$ for individual data sets without the user's input.

In Chapter 4, the equations defining agent belief, disbelief, and uncertainty were shown to be first-order equations with user-defined constants which were chosen to satisfy $b+d+u=1$. However, in most situations, this requirement was not satisfied, which resulted in a less robust implementation of subjective logic and incorrect values of belief, disbelief, and uncertainty, especially in situations where $b+d+u>2$. This result reflected the need for a better set of equations which define the agent belief tuples and is the most important aspect of the subjective logic framework that should be addressed. Future work would look at improving the agent b,d,u equations so that the condition of $b+d+u=1$ is satisfied more consistently.

The automated feature set combination was shown to effectively combine two feature sets and detect many duplicate lines between data sets. However, in some instances, vortex core lines which were visually verified to be duplicate lines were not detected by the automated method. One idea for finding believable vortex core lines is to find all believable points in vortex core lines and place the disconnected points into a new data set. A new line connection method could then be used to connect the believable points into a final set of vortex core lines.

CHAPTER 7. SUMMARY AND CONCLUSIONS

7.1 Summary

This thesis has presented a method for extracting and tracking vortex core lines from unsteady CFD data sets using subjective logic in a trust network. The method comprises five steps which may be applied to any unsteady CFD data set:

1. Extract vortex core lines from the CFD data set using unsteady feature extraction algorithms.
2. Track extracted vortex cores through time.
3. Create agent opinions for each vortex core line.
4. Combine agent opinions to form final opinions of vortex core lines.
5. Aggregate believable vortex cores from separate data sets into one final feature set.

The SH and RP algorithms were used to extract vortex core lines from unsteady data sets. Both algorithms were selected because they are well known and have documented strengths and weaknesses which complement each other. The algorithms and parameters which defined the strengths and weaknesses of the algorithms were modified for unsteady data sets. An efficient feature tracking method was also created for use with line-type features and was shown to successfully track vortex core lines through a time series of data. The opinion of the extracted and tracked vortex cores was computed using subjective logic in a trust network. The MA opinion was formulated using feature tracking parameters, while the AA opinions were computed using algorithm strengths and weaknesses as well as the λ_2 criterion. After the final opinion of the vortex core lines was determined, the believable features from both algorithm data sets were automatically combined into one final believable vortex core line data set.

7.2 Conclusions

The addition of time derivatives to the feature extraction algorithms had a noticeable effect on the vortex cores extracted. The computational cost of simultaneously loading 3 time steps into memory was felt to be necessary for correct extraction of vortex cores from unsteady CFD data sets. The vortex cores extracted with time derivatives from the lid-driven cavity data set were shifted towards the center of rotation. Also, there were many more spurious vortex cores which were extracted without time derivatives.

The automated feature set combination showed that subjective logic could be used to successfully find the believable vortex core lines in a flow simulation and to remove spurious vortex cores. A critical line-average probability expectation of $E = 0.75$ was found to be most successful at automatically removing spurious vortex core lines from the simulations and leaving only the highly believable vortex cores for visualization. In the lid-driven cavity, application of the feature set combination showed that the SH algorithm extracted the most believable primary, secondary, and corner vortex core lines and removed the corresponding vortex core lines extracted by the RP algorithm. In the cylinder data set, the vortex core lines in the far wake were marked as mostly spurious which moved the focus of the visualization on the stronger mode B vortex cores in the near wake of the cylinder.

The type of grid from which vortex cores were extracted was shown to have a significant effect on the quality of the extracted vortex core lines. Grid density in the cylinder data set had a significant effect on the quality of extracted vortex core lines. The vortex cores extracted from the structured mesh of the cylinder data set were segmented and did not generally follow the swirling flow of the data set. In the fine structured mesh, the mode B vortex core lines, as expected at the simulation flow regime, were extracted and were tracked well through time.

The RP algorithm was determined to be the dominant extraction algorithm in simulations of wake flows. The RP algorithm extracted roughly six times as many vortex cores as the SH algorithm from the cylinder and wind turbine data sets since the RP algorithm was designed to extract the ideal semi-circular vortex core line. The vortex core line opinions computed with subjective corroborated this conclusion, with higher expected probability in most of the vortex cores extracted by the RP algorithm than those extracted by the SH algorithm. The effect of increasing time step width was also shown to be very important as it decreased the flow curvature in wake simulations, which decreased the effectiveness of the RP algorithm while allowing the SH algorithm to detect more vortex core lines. In either case, increasing time step width resulted in poorer results for both algorithms.

Feature tracking was shown to have a greater effect on the final opinion of the vortex cores than any other individual characteristic of vortex cores because of its use in computing the MA opinion. When a vortex core line was untracked in both directions in time, the final uncertainty was usually $u = 1$, which resulted in a probability expectation of $E = 0.5$. The addition of using more tracking passes through the data set with increasing tolerances resulted in significantly longer tracking path lengths, which increased the belief of well-tracked vortex core lines.

Analysis of the constants used in the agent b, d, u equations showed that the most important constants were those defining MA belief and disbelief, AA_E belief, and AA_{NE} disbelief. In general, changing the belief and disbelief constants in the MA opinion resulted in the most change in opinion for both the vortex core data sets from the cylinder data set, with changes of up to 20% in $\overline{E}$ reported. Change in the belief constants of the AA_E resulted in considerable $\Delta\overline{E}$ of up to 16% for the vortex cores which the AA_E extracted. The last significant change occurred when the AA_{NE} disbelief constants were altered, with $\Delta\overline{E}$ of up to 18% in the vortex cores which the AA_{NE} did not extract.

This method allows for a clear and simple visualization of the flow physics of unsteady CFD data sets. In the lid-driven cavity simulation, the RP algorithm extracted several vortex core lines which were not expected but had high expected probability and were then verified to be centers of swirling flow. In the cylinder data set, mode B vortex cores were extracted and tracked through time and corresponded well to findings made by others. The vortex breakup in the far wake of the cylinder data set was also observed. The vortex cores extracted from the wind turbine data set showed the extent of the tip vortex cores as well as the length at which the turbine wake broke up into more random vortex cores. By use of the method, a researcher can find vortex cores with high expected probability and investigate the region from which the vortex core was extracted in greater depth as well as following the vortex core as it travels through the data set.

This method contains certain weaknesses which increase the difficulty of using it in unsteady data sets. Feature extraction and tracking results in a significant data size reduction from the CFD data set, but there is still a large amount of data to analyze, especially when the method is performed on large CFD data sets. With such a large amount of data, application of subjective logic results in incorrect opinions for certain vortex core lines. Subjective logic is also by definition uncertain, meaning that there is no clear true or false when it comes to defining the opinion of a feature, so the opinion of weaker vortex core lines may be inconclusive. One of the biggest weaknesses of the method presented here is the numerous values which define the b,d,u equations. There are three opinions with three belief tuple equations each, where each belief tuple component contains two constants, which results in 18 variables that can be changed to find the final opinion of features. Last, a good knowledge of an algorithm's strengths and weaknesses must be known to form the opinion, so algorithms which are new or not well understood cannot be used in this method.

Even with these weaknesses, the application of this method in large unsteady data sets provides a way to remove a considerable amount of spurious features and allows for clear analysis of the most believable features in a data set. Since there is no clear true or false result from subjective logic opinions, this allows some flexibility for the researcher to decide what is believable and what is not. The method also aids in the search for features in areas of a simulation that may not have been apparent and points the researcher to areas where features are most believable. In unsteady data sets, these believable features can then be followed through time to watch the interactions and evolution of features in time.

The novel application of intelligent agents to extract and track vortex core lines from unsteady CFD data set aids in the search for all relevant flow features in a time-dependent flow field. By use of subjective logic in a trust network, the belief and expected probability of features may be found if knowledge of the algorithm and flow feature physics are known. Feature tracking in unsteady data sets is also used to find the belief of a feature as it exists through time. Features with high expected probabilities from different data set are then combined into one final feature set, which simplifies the analysis of the flow domain into one simple data set. This new CFD visualization method will enable an analyst to focus on key regions of a CFD simulation and quickly analyze the physics of massive time-dependent data sets.

REFERENCES

[1] List, M., Gorrell, S., and Turner, M., 2008. "Investigation of Loss Generation in an Embedded Transonic Fan Stage at Several Gaps using High Fidelity, Time-accurate CFD." In *Proceedings of ASME Turbo Expo 2008: Power for Land, Sea and Air*.

[2] Yao, J., Wadia, A., and Gorrell, S., 2008. "High-Fidelity Numerical Analysis of Per-Rev-Type Inlet Distortion Transfer in Multistage Fans–Part II: Entire Component Simulation and Investigation." *ASME Paper GT2008-50813*, June.

[3] TECPLOT, INC, 2011. *Tecplot 360 User's Manual Release 2*. P.O. Box 52708, Bellevue, WA 98015-2708, U.S.A.

[4] COMPUTATIONAL ENGINEERING INTERNATIONAL, INC., 2008. *EnSight User Manual for Version 9.0*. 2166 N. Salem Street, Suite 101, Apex, NC 27523.

[5] Post, F., Vrolijk, B., Hauser, H., Laramee, R., and Doleisch, H., 2003. "The State of the Art in Flow Visualization: Feature Extraction and Tracking." *Computer Graphics Forum,* **22**(4), December, pp. 775–792.

[6] Ma, K.-L., van Rosendale, J., and Vermeer, W., 1996. "3D Shock Wave Visualization on Unstructured Grids." In *Proceedings of the 1996 Symposium on Volume Visualization*, pp. 87–94,104.

[7] Roth, M., 2000. "Automatic Extraction of Vortex Core Lines and Other Line-Type Features for Scientific Visualization." PhD dissertation, Swiss Federal Institute of Technology.

[8] Meadows, K. R., Kumar, A., and Hussaini, M., 1991. "Computational Study on the Interaction Between a Vortex and a Shock Wave." *AIAA Journal,* **29**(2), pp. 174–179.

[9] Inoue, O., and Hattori, Y., 1999. "Sound Generation by Shock–Vortex Interactions." *Journal of Fluid Mechanics,* **380**, pp. 81–116.

[10] Kalivas, D. S., and Sawchuk, A. A., 1991. "A Region Matching Motion Estimation Algorithm." *CVGIP: Image Understanding,* **54**(2), pp. 275–288.

[11] Reinders, F., Post, F. H., and Spoelder, H. J., 2001. "Visualization of Time-Dependent Data with Feature Tracking and Event Detection." *The Visual Computer,* **17**, pp. 55–71.

[12] Weinkauf, T., Sahner, J., Theisel, H., and Hege, H.-C., 2007. "Cores of Swirling Particle Motion in Unsteady Flows." *IEEE Transactions on Visualization and Computer Graphics,* **13**(6), November/December, pp. 1759–1766.

[13] Bauer, D., and Peikert, R., 2002. "Vortex Tracking in Scale-Space." In *Proceedings of the Symposium on Data Visualization 2002*, VISSYM '02, Eurographics Association, pp. 233–ff.

[14] Jøsang, A., 2001. "A Logic for Uncertain Probabilities." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems,* **9**(3), June, pp. 279–311.

[15] Jøsang, A., 2002. "The Consensus Operator for Combining Beliefs." *Artificial Intelligence Journal,* **141**(1-2), October, pp. 157–170.

[16] McAnally, D., and Jøsang, A., 2004. "Addition and Subtraction of Beliefs." In *Proceedings of Information Processing and Management of Uncertainty in Knowledge-Based Systems*.

[17] Mortensen, C. H., 2010. "A Computational Fluid Dynamics Feature Extraction Method Using Subjective Logic." Master's thesis, Brigham Young University, August.

[18] Jøsang, A., Hayward, R., and Pope, S., 2006. "Trust Network Analysis with Subjective Logic." In *Proceedings of the 29th Australasian Computer Science Conference*, Vol. 48, pp. 85–94.

[19] Robinson, S., 1991. "Coherent Motions in the Turbulent Boundary Layer." *Annual Reviw of Fluid Mechanics,* **23**, pp. 601–639.

[20] Nguyen, E., 2004. Mulvane, KS Tornado `http://http://www.mesoscale.ws/pic2004/040612-13.jpg`, June.

[21] eFluids, 2010. NASA Wake Vortex Study at Wallops Island `http://media.efluids.com/galleries/vortex?medium=191`, May.

[22] Villasenor, J., and Vincent, A., 1992. "An Algorithm for Space Recognition and Time Tracking of Vorticity Tubes in Turbulence." *Computer Vision, Graphics, and Image Processing: Image Understanding,* **55**(1), pp. 27–35.

[23] Hunt, J., Wray, A., and Moin, P., 1988. Eddies, Stream, and Convergence Zones in Turbulent Flows Tech. Rep. CTR-S88, Center for Turbulence Research Report.

[24] Chong, M., Perry, A., and Cantwell, B., 1990. "A General Classification of Three-Dimensional Flow Fields.." *Physics of Fluids A,* **2**, pp. 765–777.

[25] Jeong, J., and Hussain, F., 1995. "On the Identification of a Vortex." *Journal of Fluid Mechanics,* **285**, pp. 69–94.

[26] Haller, G., 2003. "An Objective Definition of a Vortex." *Journal of Fluid Mechanics,* **525**, pp. 1–26.

[27] Peikert, R., and Roth, M., 1999. "The 'Parallel Vectors' Operator – A Vector Field Visualization Primitive." In *Proceedings of IEEE Visualization '99*, pp. 263–270.

[28] Banks, D., and Singer, B., 1995. "A Predictor-Corrector Technique for Visualizing Unsteady Flow." *IEEE Transactions on Visualization and Computer Graphics,* **1**, pp. 151–163.

[29] Strawn, R. C., Ahmad, J., and Kenwright, D. N., 1999. "Computer Visualization of Vortex Wake Systems." *AIAA Journal,* **37**, apr, pp. 511–512.

[30] Sujudi, D., and Haimes, R., 1995. "Identification of Swirling Flow in 3-D Vector Fields." *AIAA Paper 95-1715*, June.

[31] Haimes, R., 1994. "pV3: A Distributed System for Large-Scale Unsteady CFD Visualization." *AIAA Paper 94-0321*.

[32] Roth, M., and Peikert, R., 1996. "Flow Visualization for Turbomachinery Design." In *Proceedings of Visualization '96*, pp. 381–384.

[33] Roth, M., and Peikert, R., 1998. "A Higher-order Method for Finding Vortex Core Lines." In *Proceedings of IEEE Visualization*, pp. 143–150.

[34] Sahner, J., Weinkauf, T., and Hege, H.-C., 2005. "Galilean Invariant Extraction and Iconic Representation of Vortex Core Lines." In *EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, K. Brodlie, D. Duke, and K. Joy, eds., pp. 151–160.

[35] Jiang, M., Machiraju, R., and Thompson, D., 2002. "Geometric Verification of Swirling Features in Flow Fields." In *Visualization, 2002. VIS 2002. IEEE*, pp. 307–314.

[36] Globus, A., Levit, C., and Lasinski, T., 1991. "A Tool for Visualizing the Topology of Three-Dimensional Vector Fields." In *VIS '91: Proceedings of the 2nd Conference on Visualization '91*, pp. 33–40.

[37] Pagendarm, H., Henne, B., and Rütten, M., 1999. "Detecting Vortical Phenomena in Vector Data by Medium-Scale Correlation." In *VIS '99: Proceedings of the Conference on Visualization '99*, pp. 409–412.

[38] Miura, H., and Kida, S., 1996. "Identification of Central Lines of Swirling Motion in Turbulence." In *Proceedings of International Conference on Plasma Physics*, pp. 866–869.

[39] Helman, J., and Hesselink, L., 1989. "Representation and Display of Vector Field Topology in Fluid Flow Data Sets." *IEEE Computer,* **22**(8), pp. 27–36.

[40] Helman, J., and Hesselink, L., 1991. "Visualizing Vector Field Topology in Fluid Flows." *IEEE Computer Graphics and Applications,* **11**(3), pp. 36–46.

[41] Dörrie, H., 1965. *100 Great Problems of Elementary Mathematics*. Dover.

[42] Fuchs, R., Peikert, R., Hauser, H., Sadlo, F., and Muigg, P., 2008. "Parallel Vectors Criteria for Unsteady Flow Vortices." *IEEE Transactions on Visualization and Computer Graphics,* **14**(3), May/June, pp. 615–626.

[43] Schindler, B., Fuchs, R., Biddiscombe, J., and Peikert, R., 2009. "Predictor-Corrector Schemes for Visualization of Smoothed Particle Hydrodynamics Data." *IEEE Transactions on Visualization and Computer Graphics,* **15**(6), November/December, pp. 1243–1250.

[44] Thiesel, H., and Seidel, H.-P., 2003. "Feature Flow Fields." In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, G.-P. Bonneau, S. Hahmann, and C. Hansen, eds., The Eurographics Association, pp. 141–149.

[45] Theisel, H., Sahner, J., Weinkauf, T., Hege, H.-C., and Seidel, H.-P., 2005. "Extraction of Parallel Vector Surfaces in 3D Time-Dependent Fields and Application to Vortex Core Line Tracking." In *Proc. IEEE Visualization 2005*, pp. 631–638.

[46] Jiang, M., Machiraju, R., and Thompson, D., 2005. "Detection and Visualization of Vortices." In *The Visualization Handbook*. Academic Press, pp. 295–309.

[47] Peikert, R., Parkinson, E., Ait-Bouziadz, Y., Sicky, M., Sadlo, F., Favrex, J., Biddiscombex, J., and Jangx, Y., 2008. Physically Based Methods for Flow Visualization and Analysis and Interactive Exploration Techniques for Time-Dependent CFD Data Tech. Rep. 1, CTI.

[48] Marusic, I., Candler, G., Interrante, V., Subbareddy, P., and Moss, A., 2001. *Real Time Feature Extraction for the Analysis of Turbulent Flows*. Springer, ch. 13, pp. 223–238.

[49] Fuchs, R., Kemmler, J., Schindler, B., Sadlo, F., Hauser, H., and Peikert, R., 2010. "Toward a Lagrangian Vector Field Topology." *Computer Graphics Forum,* **29**(3), pp. 1163–1172.

[50] Kasten, J., Hotz, I., Noack, B., and Hege, H.-C., 2010. "On the Extraction of Long-Living Features in Unsteady Fluid Flows." In *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications (TopoInVis'09)*.

[51] Fuchs, R., Peikert, R., Sadlo, F., Alsallakh, B., and Gröller, M. E., 2008. "Delocalized Unsteady Vortex Region Detectors." In *Proceedings VMV 2008*, D. K. D. S. Oliver Deussen, ed., pp. 81–90.

[52] Shi, K., Theisel, H., Hauser, H., Weinkauf, T., Matkovic, K., Hege, H.-C., and Seidel, H.-P., 2009. "Path Line Attributes – An Information Visualization Approach to Analyzing the Dynamic Behavior of 3D Time-Dependent Flow Fields." In *Topology-Based Methods in Visualization II*, H.-C. Hege, K. Polthier, and G. Scheuermann, eds., Mathematics and Visualization, Springer, pp. 75–88.

[53] Weinkauf, T., Theisel, H., Gelder, A. V., and Pang, A., 2010. "Stable Feature Flow Fields." *IEEE Transactions on Visualization and Computer Graphics,* **17**(6), pp. 770–780.

[54] Samtaney, R., Silver, D., Zabusky, N., and Cao, J., 1994. "Visualizing Features and Tracking Their Evolution." *IEEE Computer,* **27**(7), July, pp. 20–27.

[55] Silver, D., and Wang, X., 1997. "Tracking and Visualizing Turbulent 3D Features." *IEEE Transactions on Visualization and Computer Graphics,* **3**(2), April-June, pp. 129–141.

[56] Chen, J., Silver, D., and Parashar, M., 2003. "Real Time Feature Extraction and Tracking in a Computational Steering Environment." In *Proceedings of the High Performance Computing Symposium, HPC2003, Society for Modeling and Simulation International*, pp. 155–160.

[57] Tzeng, F.-Y., and Ma, K.-L., 2005. "Intelligent Feature Extraction and Tracking for Visualizing Large-Scale 4D Flow Simulations." In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, p. 6.

[58] Muelder, C., and Ma, K.-L., 2009. "Interactive Feature Extraction and Tracking by Utilizing Region Coherency." In *Proceedings of the 2009 IEEE Pacific Visualization Symposium*, PACIFICVIS '09, IEEE Computer Society, pp. 17–24.

[59] Schafhitzel, T., Baysal, K., Rist, U., Weiskopf, D., and Ertl, T., 2008. "Particle-Based Vortex Core Line Tracking Taking into Account Vortex Dynamics." In *Proceedings of International Symposium on Flow Visualization '08*.

[60] Schafhitzel, T., Baysal, K., Vaaraniemi, M., Rist, U., and Weiskopf, D., 2011. "Visualizing the Evolution and Interaction of Vortices and Shear Layers in Time-Dependent 3D Flow." *IEEE Transactions on Visualization and Computer Graphics,* **17**(4), April, pp. 412–425.

[61] eFluids, 2012. Particle Image Velocimetry `http://www.efluids.com/efluids/pages/products/piv.htm`, March.

[62] Reinders, F., Post, F. H., and Spoelder, H. J. W., 1999. "Attribute-based Feature Tracking." In *Data Visualization '99*, Springer Verlag, pp. 63–72.

[63] Reinders, F., Sadarjoen, I. A., Vrolijk, B., and Post, F. H., 2002. "Vortex Tracking and Visualisation in a Flow Past a Tapered Cylinder." *Computer Graphics Forum,* **21**(4), Nov, pp. 675–682.

[64] Bauer, D., 2006. "Selective Visualization of Unsteady 3D Flow Using Scale-Space and Feature-Based Techniques." PhD dissertation no. 16640, Swiss Federal Institute of Technology (ETH).

[65] Bland, J. M., and Altman, D. G., 1996. "Transforming Data." *British Medical Journal,* **312**, March, p. 770.

[66] Albensoeder, S., and Kuhlmann, H., 2005. "Accurate Three-Dimensional Lid-Driven Cavity Flow." *Journal of Computational Physics,* **206**(2), July, pp. 536–558.

[67] Zhang, H.-Q., Fey, U., Noack, B. R., König, M., and Eckelmann, H., 1995. "On the Transition of the Cylinder Wake." *Physics of Fluids,* **7**(4), April, pp. 779–794.

[68] Williamson, C., 1996. "Vortex Dynamics in the Cylinder Wake." *Annual Review of Fluid Mechanics,* **28**, pp. 477–539.

[69] Roshko, A., 1952. "On the Development of Turbulent Wakes from Vortex Streets." PhD thesis, California Institute of Technology.

[70] Bloor, M., 1964. "The Transition to Turbulence in the Wake of a Circular Cylinder." *Journal of Fluid Mechanics,* **19**(290), pp. 290–304.

[71] Thompson, M., Hourigan, K., and Sheridan, J., 1996. "Three-Dimensional Instabilities in the Wake of a Circular Cylinder." *Experimental Thermal and Fluid Science,* **12**, pp. 190–196.

[72] Hand, M., Simms, D., Fingersh, L., Jager, D., Cotrell, J., Schreck, S., and Larwood, S., 2001. Unsteady Aerodynamics Experiment Phase VI: Wind Tunnel Test Configurations and Available Data Campaigns Tech. Rep. NREL/TP-500-29955, National Renewable Energy Laboratory, 1617 Cole Boulevard, Golden, Colorado 80401-3393, December.

[73] Nichols, R. H., and Buning, P. G., 2008. *User's Manual for OVERFLOW 2.1.*, 2.1t ed. NASA, August.

[74] Duque, E. P., Burklund, M. D., and Johnson, W., 2003. "Navier-Stokes and Comprehensive Analysis Performance Predictions of the NREL Phase VI Experiment." *ASME Journal of Solar Enery Engineering,* **125**, November, pp. 457–467.

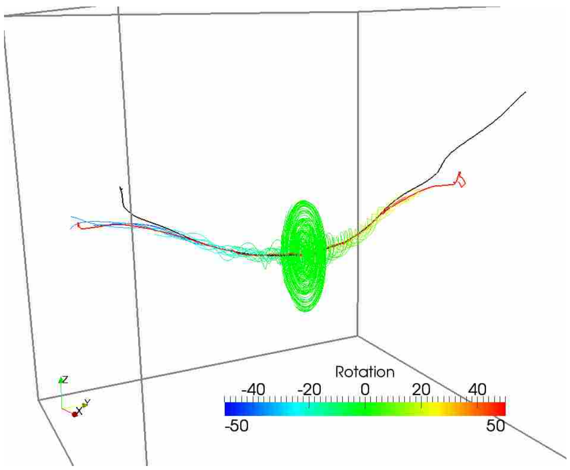
[75] Lively, M. C., 2012. "Extraction of Shock Waves and Separation and Attachment Lines From Computational Fluid Dynamics Simulations Using Subjective Logic." Master's thesis, Brigham Young University, June.

[76] Kitware, 2006. *The VTK User's Guide*., 5 ed. Kitware, Inc.

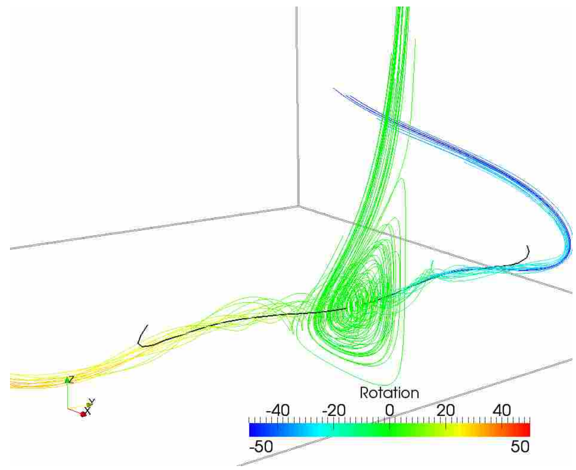
[77] Kitware, 2006. *The Visualization Toolkit*., 4 ed. Kitware, Inc.

APPENDIX A. FLOW VISUALIZATION IMAGES

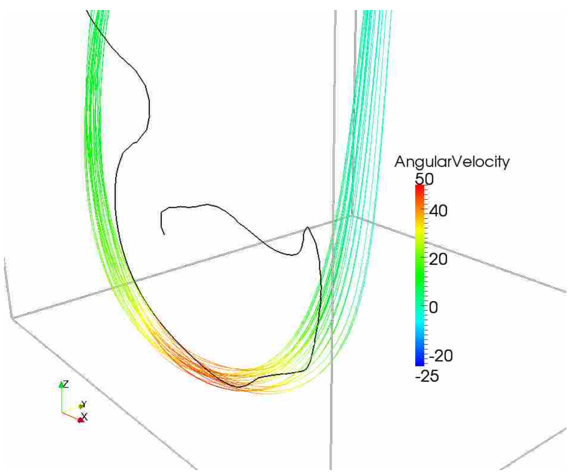
This appendix contains additional flow visualization conducted for the lid-driven cavity and the cylinder in cross flow.



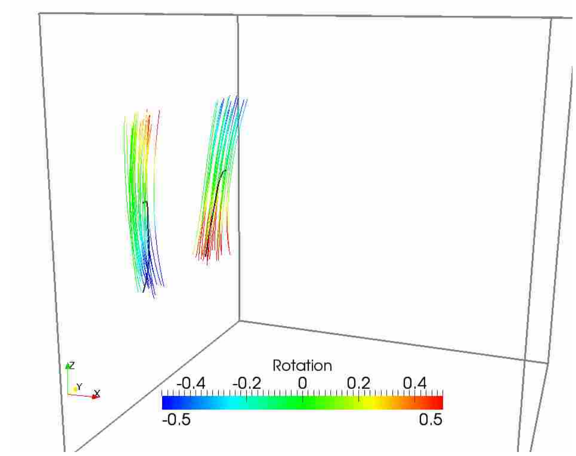
(a) Streamlines seeded around the primary vortex core shows that SH (red) was better than RP (blue) at extracting the primary core.



(b) Streamline rotation verifies extents of secondary vortex core.

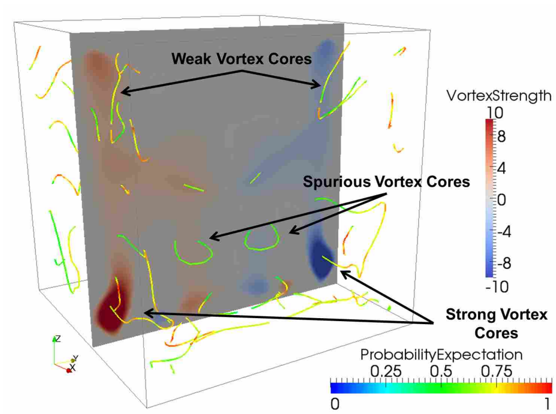


(c) Swirling flow is verified for RP stream-wise vortex cores.

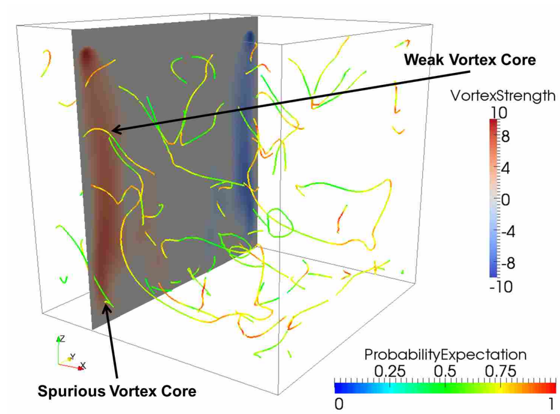


(d) Weak swirling flow is detected in the RP Taylor-Görtler-Like vortex cores.

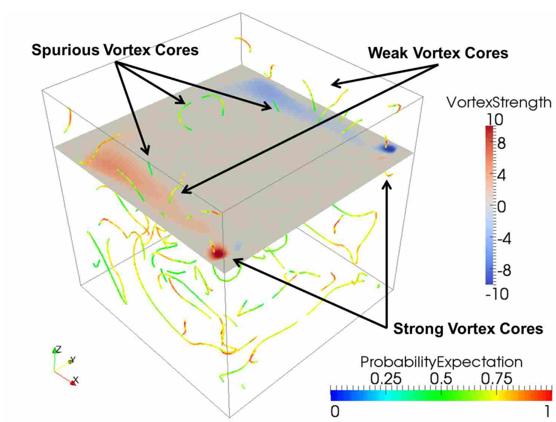
Figure A.1: Verification of the main vortex core lines in the lid-driven cavity set. Streamlines are used to show swirling strength and vortex extents. Lid moves in the $+x$-direction.



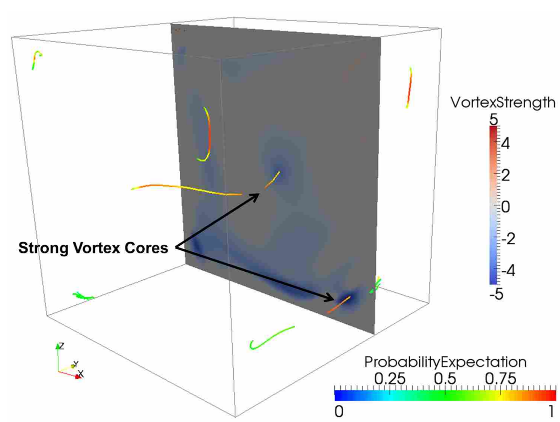
(a) *yz*-slice of CFD data set with RP vortex cores.



(b) *yz*-slice of CFD data set with RP vortex cores.



(c) *xy*-slice of CFD data set with RP vortex cores.



(d) *xz*-slice of CFD data set with SH vortex cores.

Figure A.2: Verification of vortex core lines in the lid-driven cavity set. Cutting planes of the CFD data set colored by vortex strength show the correct and spurious vortex cores and that the computed subjective logic of the vortex cores agrees with the manual visualization. Lid moves in the $+x$-direction.

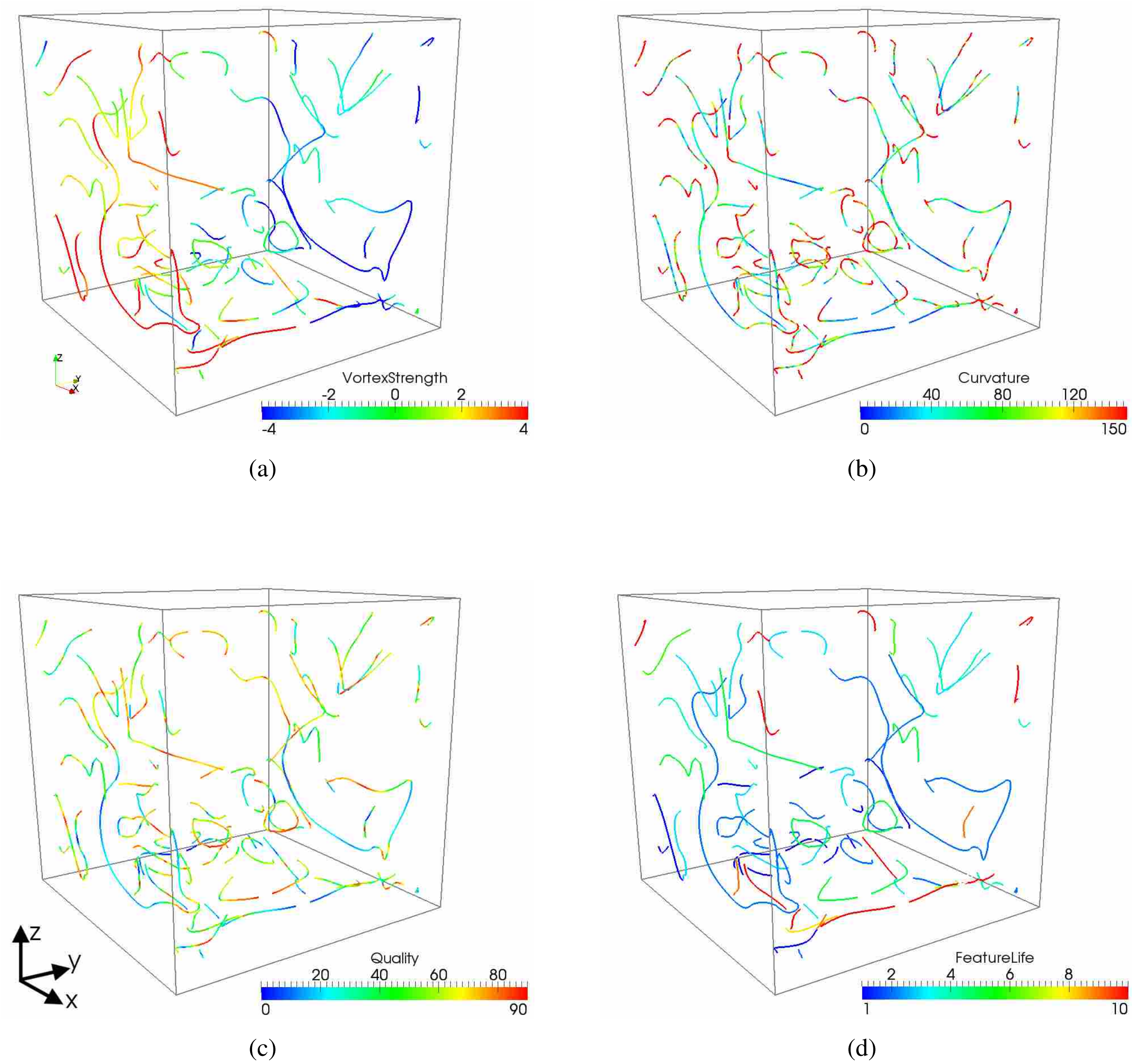


Figure A.3: Values for the RP vortex cores at $t = 3.0s$. Lid moves in the $+x$-direction.

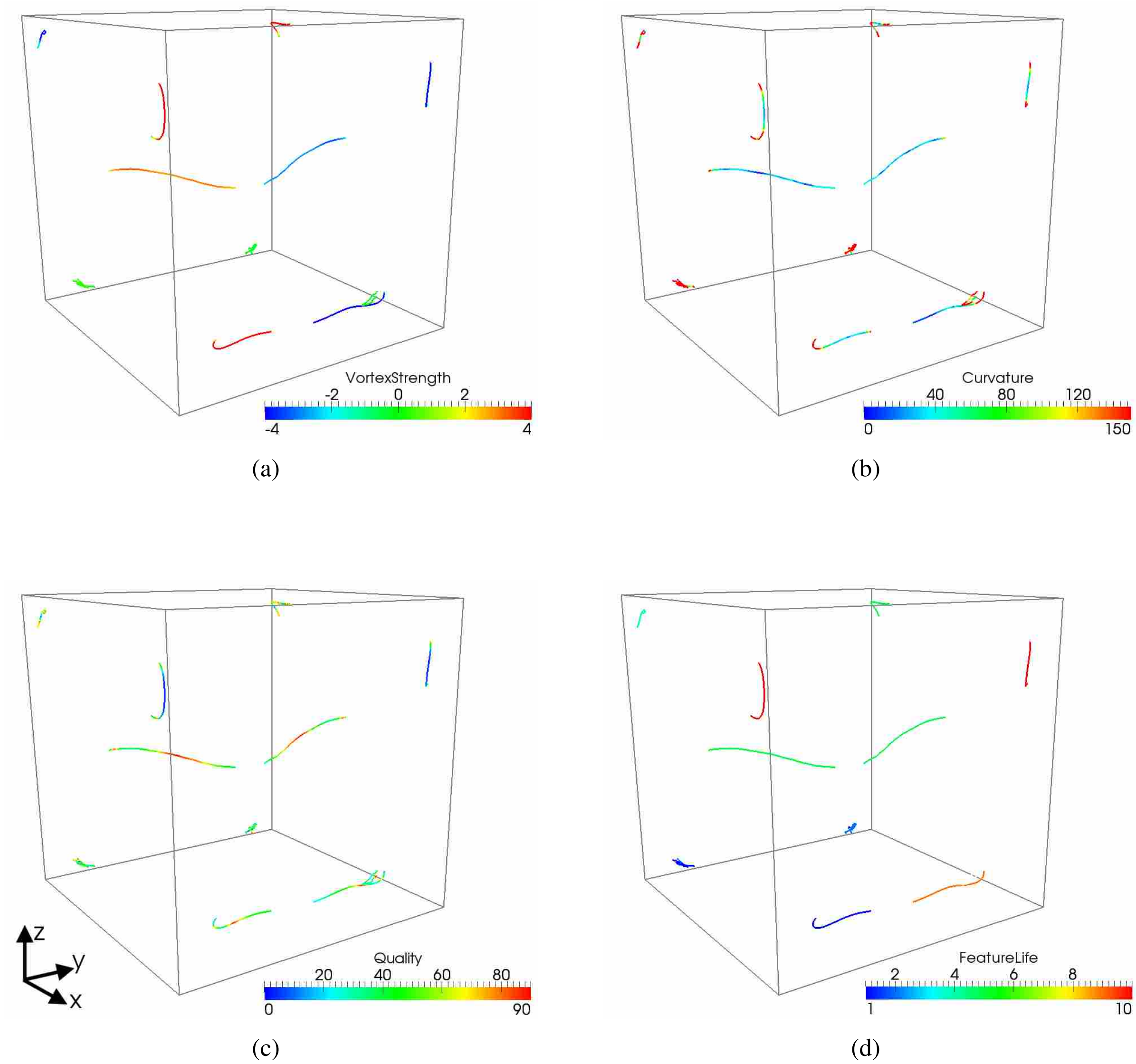
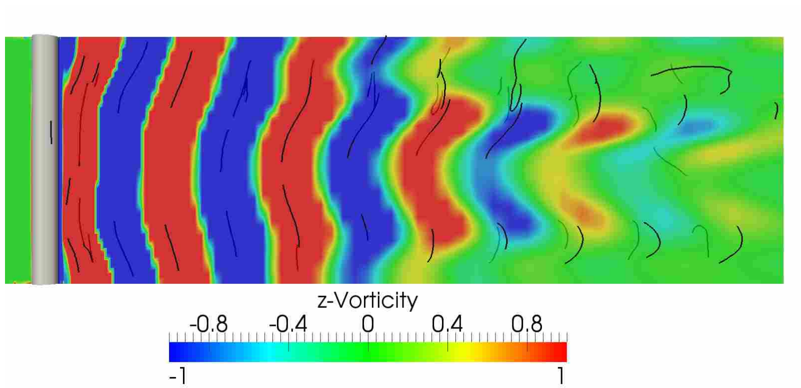
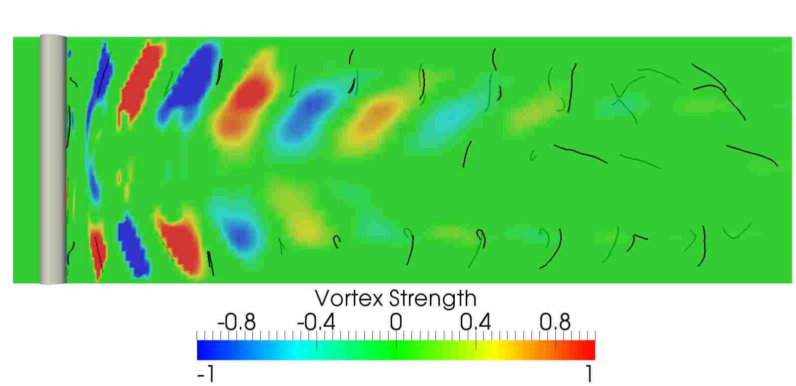


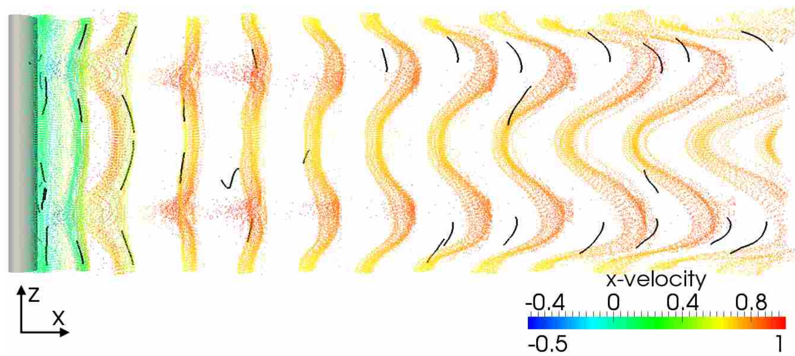
Figure A.4: Values for the SH vortex cores at $t = 3.0s$. Lid moves in the $+x$-direction.



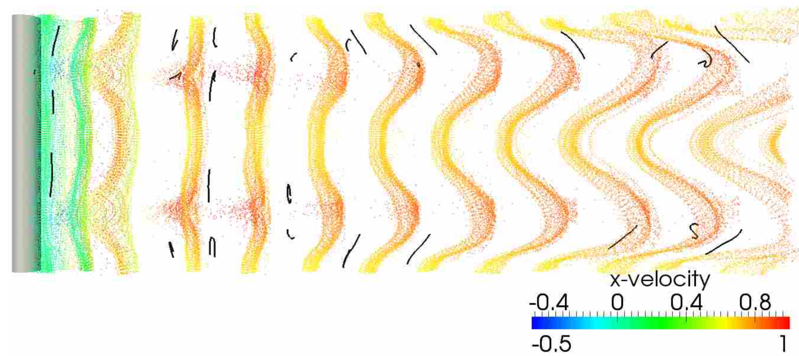
(a) Slice of CFD data set colored by z-vorticity with overlaid RP vortex cores.



(b) Slice of CFD data set colored by vortex strength with overlaid SH vortex cores.

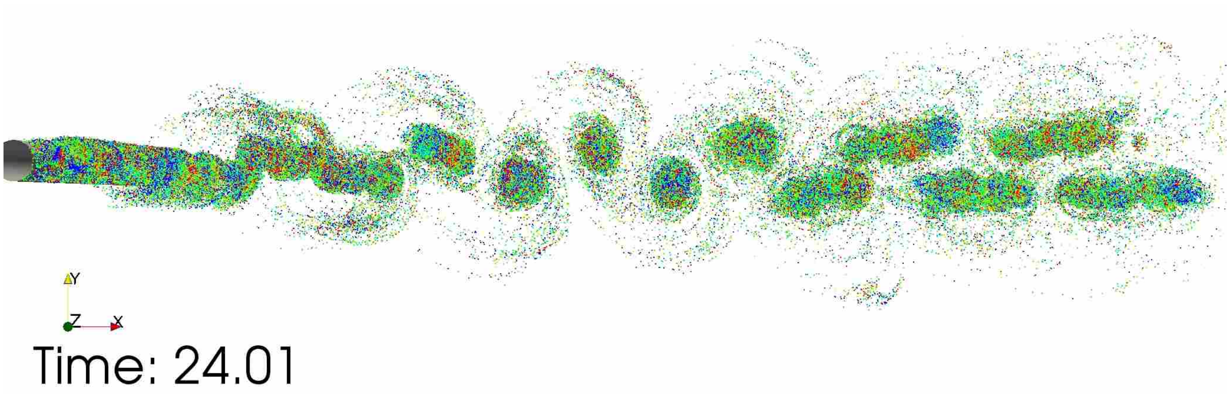


(c) Particles seeded in CFD data set and overlaid with RP vortex cores.

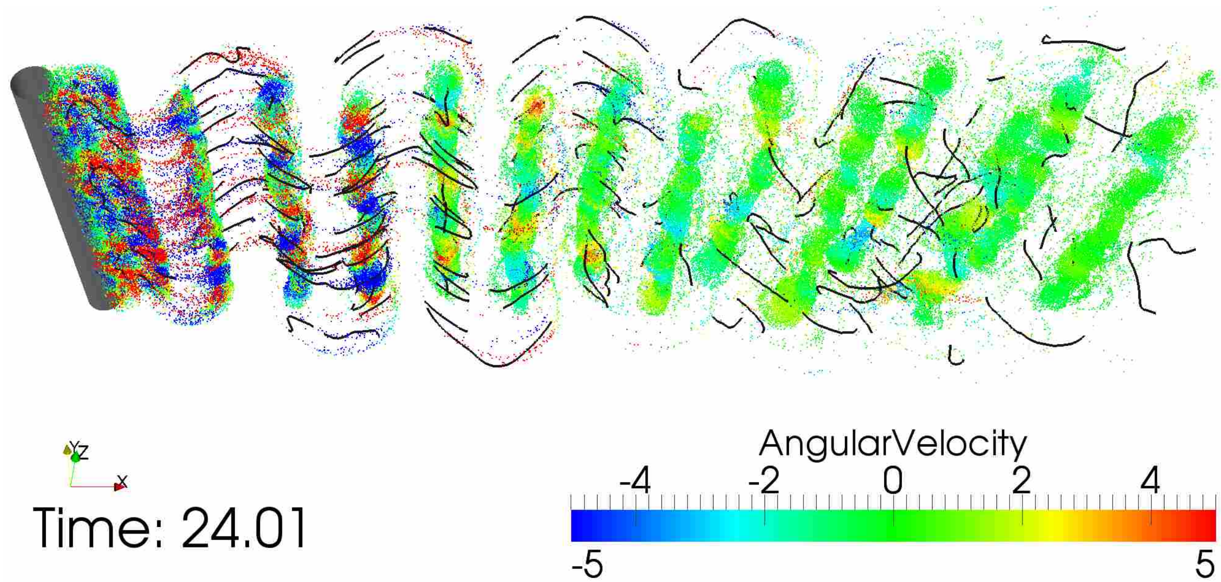


(d) Particles seeded in CFD data set and overlaid with SH vortex cores.

Figure A.5: Visualization of cylinder data set vortex cores extracted from the structured coarse mesh (Section 5.2.1). RP vortex cores agree with the simulation more than than SH vortex cores. Vortex stretching can be seen in the cylinder far wake.

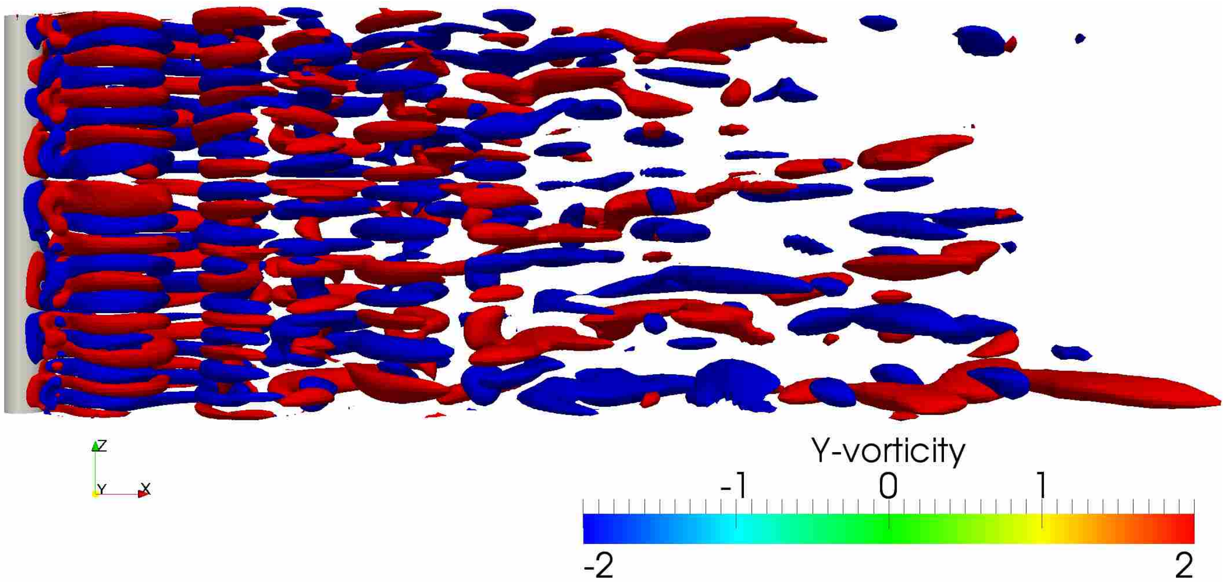


(a) Traced particles colored by rotation.

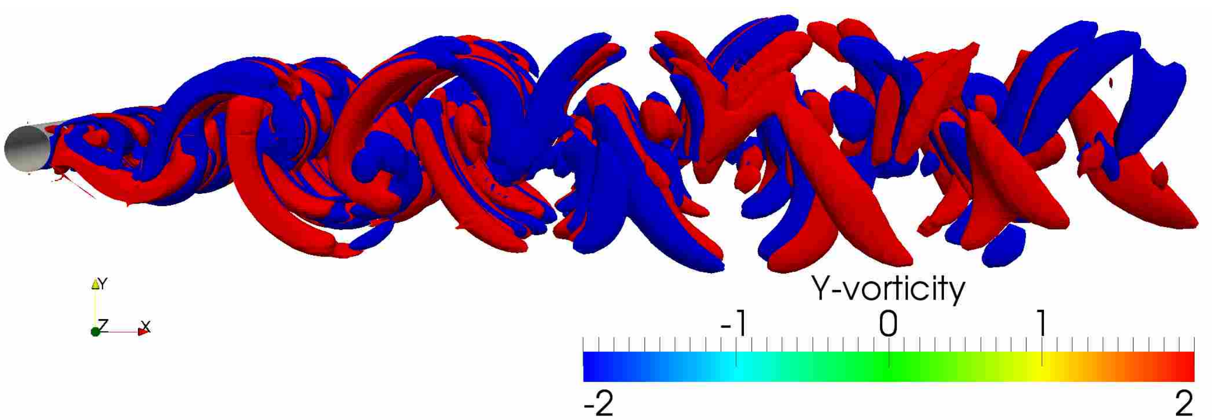


(b) View of traced particles and vortex core lines extracted by the RP algorithm.

Figure A.6: Visualization of a particle trace in the structured fine cylinder CFD data set.



(a) y-vorticity isosurface viewed from the xz-plane.



(b) y-vorticity isosurface viewed from the xy-plane.

Figure A.7: Visualization of y-vorticity isosurfaces in the structured fine cylinder CFD data set.

APPENDIX B. USER'S GUIDE TO VORTEX CORE EXTRACTION METHOD WITH SOURCE CODE

B.1 User's Guide

The code that runs the intelligent vortex core extraction and tracking is shown below. To run the code, Cafe_script.bash, which contains user inputs and is shown in Section B.2.1, is run from the command line. The 'main' program, which is contained in Section B.2.2, contains all the routines that are required to extract vortex core lines, track the vortex cores through time, and compute the opinion of the extracted and tracked vortex cores. Before this code will compile the VTK 5.8 libraries with parallel enabled must be compiled and working properly. All other linked libraries come from the C++ Standard Library. This code has been compiled on Ubuntu 10.04 LTS (Lucid Lynx) using g++ and cmake 2.8 to create make files. Each section of the code will be explained below.

B.1.1 Cafe_script.bash

The bash script created for this research contains many user inputs which are subsequently passed into the main routine. Lines 5–7 specify which types of features will be extracted. Line 10 specifies whether or not the data set under consideration is time-dependent. In line 13, the time step is input, and line 16 specifies the time value of the first data set under consideration. line 19 specifies the data set file type and supports Ensight, FLUENT, Plot3D, OVERFLOW, and VTK file types.

File paths and names are specified in Lines 22–43. In lines 22–26, the path to the executable is specified based on the time-dependence of the data set. Line 30 sets the file path to the CFD data sets and line 34 sets the file path where vortex core line data sets will be written. In line 39, the base name of the CFD data set is specified and line 40 sets the file extension of the CFD data set. Last, the number of CFD data sets under consideration is set in line 40.

Lines 45–70 run the actual intelligent extraction and tracking code. When ‘TRANSIENT’ is set to ‘true’, then the unsteady feature extraction and tracking code is run using line 46. Lines 48–68 are used when features are extracted from steady-state CFD data sets. In this research, only transient data sets were considered, so the steady-state section of the script was not used.

B.1.2 intelligentExtractionTransient.cxx

The # include statements on lines 1–2 call other files which include all the required C++ and VTK classes which are required for the code to work. Lines 10–21 contain additional user inputs for the entire code and include calls on which portion of the code will be run, i.e. extraction/tracking/opinion. Lines 24–48 contain additional inputs which are specific to vortex core line extraction. Lines 25–31 are specific to extraction, and lines 33–48 pertain to feature tracking.

Inputs from the bash script are instantiated in lines 52–69 for later use in the code. For transient data sets, each input and output file pertains to a certain time in the simulation, so an array of the times under consideration is created in lines 71–131. After the array is created, each time is converted to a string with the necessary leading and trailing zeros in order to create a time step file name appellation for the vortex core line files.

In order to handle the different input file types, the code has a different section for each input type, which is contained in lines 133–589. Velocity, pressure, and density array names are created specific to each file type in lines 133–172. File names and other variables are passed from the bash script in lines 174–182. Because some of the file types require a multi-block data set, each data set is read into a multi-block data set. Three multi-block data sets are created in lines 188-195, which correspond to the current, previous, and next time step of the data set. Three data sets are read in simultaneously for computation of time derivatives. Lines 205–290 contain the routine for reading in FLUENT files and calculating the velocity vector. Ensight offers the choice of transient and instantaneous file types, which are handled using lines 292–341 and lines 343–389, respectively. PLOT3D files are read in using lines 391–461. The OVERFLOW routine is contained is lines 463–541. OVERFLOW files are often multi-block data sets, so for-loops are used to assign each block of the data sets to the respective blocks of the VTK multi-block data set. Last, the VTK file reader is contained is lines 543–589.

Lines 591–855 is the section of the code where feature extraction is performed. Lines 591–601 set up the vortex core line output file names using the given output file prefix and the time step under consideration. The for-loop on line 608 starts the extraction for vortex cores from each block of the data set. The results of extraction from each block are appended onto a vtkPolyData structure, which is instantiated in lines 603–605. In lines 610–639, cell-centered data is converted to point data due to the requirements of the extraction algorithms. Velocity time derivatives are then computed in lines 641–657. Cells near walls are removed using a velocity magnitude threshold in lines 665–673, then the λ_2 criterion is calculated for all points in the domain in lines 675–680. If it is desired to write the full CFD data set with λ_2 and vortex strength at each point, then the code in lines 682–701 is used. Vortex core lines are then extracted using the Roth-Peikert algorithm in lines 703–717, which takes in the vtkUnstructuredGrid with a velocity vector field as input and outputs raw polylines. A similar process is conducted using the Sujudi-Haimes algorithm in lines 722–736. After all blocks in the data set have been inspected, the output from the two algorithms is cleaned in lines 743–747 to remove duplicate vortex cores. Vortex core attirbutes are then calculated for both data sets and the vortex core data is then written to file in lines 749–855.

Feature tracking is accomplished in lines 857–1434. Line-averaged vortex core attributes are first calculated in lines 862–956. Tracking is then begun in line 976 after instantiating some variables for tracking. The for-loop on line 976 sets up how many forward and backward passes are performed through the time series of vortex core lines. The for-loop on line 980 then enters the forward pass through the data set. Lines 980–1122 contain the forward tracking pass, where vortex cores are tracked in positive time. Lines 1124-1275 perform a similar function as the forward pass but now a backward pass is conducted through negative time. After performing n passes through the data set, the feature lifetime of the tracked vortex cores is measured in lines 1282–1348. Different tracking parameters such as average feature life are then calculated in lines 1350–1368. The calculated feature lifetimes are then set for each vortex core in lines 1371–1434.

The subjective logic portion of the code is contained in lines 1437–1589. Subjective logic is calculated starting at $i == 3$ due to feature tracking and time derivative constraints. The timing of the opinion calculation is performed in lines 1444–1450 and file names are instantiated in lines 1452–1465. The RP and SH vortex core lines are then read in lines 1467–1481, after which the minimum distance between both data sets is measured in lines 1483–1497. The final opinion of the data sets are then computed and unnecessary arrays are removed in lines 1499–1550. The last steps are to combine believable vortex cores into the final data set, which is performed in lines 1552–1562, and to write all the results to file in lines 1564–1583. Lines 1585–1587 deal with code timing, then the code exits on line 1592.

B.2 Source Code

B.2.1 Cafe_script.bash

```
#!/bin/bash

#Select which features to extract
#Specify "true" or "false"
SA=false
SHOCK=false
VORTEX=true

#Specify whether simulation is time-dependent
TRANSIENT=true

#Change to the iteration interval/time step between each saved dataset
DATASET_INTERVAL=0.01

#Change to the iteration/time step of the first saved dataset
CURRENT_DATASET=19.02

#Change to the type of saved datasets (ensight, ensighttransient, fluent, plot3d, overflow, vtk)
MODE=ensighttransient

#Specify the path to the executable and the executable name
if [ $TRANSIENT == 'true' ]; then
    IE_PATH=/home/rshaw/Workspace/finalIntelligentExtraction/runIntelligentExtractionTransient
else
    IE_PATH=/home/rshaw/Workspace/finalIntelligentExtraction/runIntelligentExtractionSteady
fi

#Specify the path to the directory where the files to be processed are
#Change to address of saved datasets to process
INPUT_PATH=/home/rshaw/Workspace/dataSets/CylinderFine/

#Specify the path to the directory where the extracted files go
#Change to address of where you want extracted files to be saved
OUTPUT_PATH=/home/rshaw/Workspace/dataSets/CylinderFine/

#Specify the base file name for your files to be processed
#for overflow the grid file must be named grid.in or this script will fail
#Also for overflow the FILE_BASE_NAME must be equal to q.
FILE_BASE_NAME='cyl'        #Change to the name of the datasets of interest
FILE_EXTENSION='.encas'        #Change to the suffix of the datasets of interest

#Set number of data sets to analyze
NUM_OF_DATASETS=502

if [ $TRANSIENT == 'true' ]; then
    $IE_PATH $DATASET_INTERVAL $CURRENT_DATASET $NUM_OF_DATASETS $SA $SHOCK $VORTEX $INPUT_PATH $FILE_BASE_NAME $OUTPUT_PATH $MODE
else
    i=1
```

```
    PREVIOUS_DATASET=$(($CURRENT_DATASET-$DATASET_INTERVAL))
    while [ $i -lt $NUM_OF_DATASETS ]
    do
        if [ $MODE == 'overflow' ]; then
            if [ $i == 0 ]; then
                $IE_PATH $DATASET_INTERVAL $i $SA $SHOCK $VORTEX $MODE $INPUT_PATH'grid.in'
                    $INPUT_PATH'q.'$CURRENT_DATASET $OUTPUT_PATH
            else
                $IE_PATH $DATASET_INTERVAL $i $SA $SHOCK $VORTEX $MODE $INPUT_PATH'grid.in'
                    $INPUT_PATH'q.'$CURRENT_DATASET $OUTPUT_PATH $OUTPUT_PATH'x_'
                    $PREVIOUS_DATASET
            fi
        else
            if [ $i == 0 ]; then
                $IE_PATH $DATASET_INTERVAL $i $SA $SHOCK $VORTEX $MODE
                    $INPUT_PATH$FILE_BASE_NAME$CURRENT_DATASET$FILE_EXTENSION $OUTPUT_PATH
            else
                $IE_PATH $DATASET_INTERVAL $i $SA $SHOCK $VORTEX $MODE
                    $INPUT_PATH$FILE_BASE_NAME$CURRENT_DATASET$FILE_EXTENSION $OUTPUT_PATH
                    $OUTPUT_PATH$FILE_BASE_NAME$PREVIOUS_DATASET
            fi
        fi

      i=$((1+$i))
      PREVIOUS_DATASET=$CURRENT_DATASET
      CURRENT_DATASET=$(($CURRENT_DATASET + $DATASET_INTERVAL))
    done
fi
```

B.2.2 intelligentExtractionTransient.cxx

```
#include <headers.h>
#include <classHeaders.h>

int main(int argc, char* argv[])
{
  //----------------------------------------------------------------------------
  // Extracting features from data sets

  //----------------------------------------------------------------------------
  // General user inputs
  int numLeadingZeros(0);          // Number of leading zeros in file name
  int numTrailingZeros(0);         // Number of trailing zeros in file name
  bool extract = false;             // Do you want to extract features?
  bool writeDataSet = false;        // Do you want to write out a copy of the CFD data set?
  bool track = true;              // Do you want to track features?
  bool logic = false;              // Do you want to perform subjective logic?
  bool verbose = false;            // Output to screen the percent complete
  int cpu = 1;                     // Number of cpus to use for vtkParallelVectors class
  double probExpThreshold = 0.7;   // Used for combining outputs
  double combLengthTol = 0.25;     // length tolerance for combining lines
  double combDistTol = 0.25;        // distance tolerance for combining lines

  //----------------------------------------------------------------------------
  // User inputs for vortex extraction / tracking
  double qualityThresholdValue = 45;  // Typically between 30 and 45 degrees.
  bool thresholdLines = false;        // Tells quality filter to threshold lines
  int minimumCorePoints = 20;          // Min value 5
  bool adaptiveMesh = false;          // Required for time derivatives
  int numBlocksToDerive = 14;        // Change to desired number for adaptive meshes
  bool timeStepPhys = false;          // True if physical time step is not file time step
  double dtPhys = 1000000;        // Physical time step for computing time derivatives
  int numberOfPasses = 10;
  double lengthTolerance = 0.15;
  double strengthTolerance = 0.2;
```

```
double curvatureTolerance = 0.2;
double qualityTolerance = 0.1;
double distanceTolerance = 0.1;
double lengthIncr = 0.1*lengthTolerance;
double strengthIncr = 0.1*strengthTolerance;
double curvatureIncr = 0.1*curvatureTolerance;
double qualityIncr = 0.1*qualityTolerance;
double distanceIncr = 0.1*distanceTolerance;
double lengthWeight = 0.20;
double strengthWeight = 0.20;
double curvatureWeight = 0.20;
double qualityWeight = 0.20;
double distanceWeight = 0.20;
int normFeatureLife = 10;          // IMPORTANT -- Change based on how long
                                  // vortex cores are expected to exist in time

//------------------------------------------------------------------------------
//Begin determining dataset type and use correct reader
string inputFileName, inputFileName_ext, inputFileName_noext, filePathName;
string outputLocation;
string outputFileNameSH, outputFileNameRP,
       outputFileNameDataSet;
string passiveResultsName;

// Time step between saved data sets
double timeStep;
sscanf(argv[1], "%lf",&timeStep);

// Start time of data sets
double startTime;
sscanf(argv[2], "%lf",&startTime);

// Start time of data sets
int numberOfDataSets;
sscanf(argv[3], "%d",&numberOfDataSets);

  // Naming variables
size_t decimalFound;
string intPart, decPart;
int numDecimals[numberOfDataSets], numDecimalsMax(0),
    numIntegers[numberOfDataSets], numIntegersMax(0);
double time;

// Creating double and string time arrays
double timeArray[numberOfDataSets];
string timeArrayString[numberOfDataSets];
for(int i = 0 ; i < numberOfDataSets ; ++i)
{
  // Setting time i
  stringstream out;
  if(i == 0)
    timeArray[0] = startTime;
  else
    timeArray[i] = timeArray[i-1] + timeStep;

  // Passing time to a string
  out << timeArray[i];
  timeArrayString[i] = out.str();

  // Parsing time by decimal point
  decimalFound = timeArrayString[i].find('.');
  if(decimalFound != string::npos)      // if a decimal exists
  {
    intPart = timeArrayString[i].substr(0,decimalFound);
    decPart = timeArrayString[i].substr(decimalFound+1);

    // Setting number of integer and decimal places
    numIntegers[i] = intPart.length();
```

```
      numDecimals[i] = decPart.length();
    }
    else                                    // if no decimal exists
    {
      numIntegers[i] = timeArrayString[i].length();
      numDecimals[i] = 0;
      intPart = timeArrayString[i];
      decPart = "";
    }

    // Setting max integer and decimal place counts
    if(numIntegers[i] > numIntegersMax)
      numIntegersMax = numIntegers[i];
    if(numDecimals[i] > numDecimalsMax)
      numDecimalsMax = numDecimals[i];

    // Omitting decimal point in string
    timeArrayString[i] = intPart + decPart;
  }

  cout << endl;

  // Adding necessary 0's to front and end of string
  for(int i = 0 ; i < numberOfDataSets ; ++i)
  {
    timeArrayString[i].insert(0,numIntegersMax-numIntegers[i]+numLeadingZeros,'0');
    timeArrayString[i].append(numDecimalsMax-numDecimals[i]+numTrailingZeros,'0');
    cout << timeArrayString[i] <<endl;
  }

  // Create velocity, pressure, and density array names
  const char* velocityArrayName;
  const char* pressureArrayName;
  const char* densityArrayName;
  if(strcmp(argv[10],"fluent") == 0)
  {
    velocityArrayName = "Velocity";
    pressureArrayName = "PRESSURE";
    densityArrayName  = "DENSITY";
  }
  else if(strcmp(argv[10],"ensighttransient") == 0)
  {
    velocityArrayName = "velocity";
    pressureArrayName = "pressure";
    densityArrayName  = "density";
  }
  else if(strcmp(argv[10],"ensight") == 0)
  {
    velocityArrayName = "velocity";
    pressureArrayName = "pressure";
    densityArrayName  = "density";
  }
  else if (strcmp(argv[10],"plot3d") == 0)
  {
    velocityArrayName = "Velocity";
    pressureArrayName = "Pressure";
    densityArrayName  = "Density";
  }
  else if (strcmp(argv[10],"overflow") == 0)
  {
    velocityArrayName = "Velocity";
    pressureArrayName = "Pressure";
    densityArrayName  = "Density";
  }
  else
  {
    velocityArrayName = "Velocity";
    pressureArrayName = "Pressure";
```

```
    densityArrayName  = "Density";
  }

  // File name structure
  string inputFilePath, fileBaseName, inputFilePrefix,
         inputFileSuffix, outputFilePath, outputFilePrefix,
         fullFileName, fullFileNameNext, fullFileNamePrev;
  inputFilePath = argv[7];
  fileBaseName = argv[8];
  inputFilePrefix = inputFilePath + fileBaseName;
  outputFilePath = argv[9];
  outputFilePrefix = outputFilePath + fileBaseName;

  cout << "Input file prefix:  " << inputFilePrefix  << endl;
  cout << "Output file prefix: " << outputFilePrefix << endl;
  cout << "File mode: " << argv[10] << endl;

  // Creating multi-block data sets for time steps
  vtkSmartPointer<vtkMultiBlockDataSet> multiBlock =
      vtkSmartPointer<vtkMultiBlockDataSet>::New();
  vtkSmartPointer<vtkMultiBlockDataSet> multiBlockNext =
      vtkSmartPointer<vtkMultiBlockDataSet>::New();
  vtkSmartPointer<vtkMultiBlockDataSet> multiBlockPrev =
      vtkSmartPointer<vtkMultiBlockDataSet>::New();
  int numberOfBlocks, numberOfBlocksNext, numberOfBlocksPrev;

  // Storing number of vortex core lines
  int numLinesRP(0), numLinesSH(0);

  /***********************PERFORMING FEATURE EXTRACTION***********************/
  if(extract)
  {
    for(int i = 1 ; i < numberOfDataSets-1 ; ++i)
    {
      // FLUENT Reader
      if(strcmp(argv[10],"fluent") == 0)
      {
        // Parsing file names
        inputFileSuffix = ".cas";
        fullFileName     = inputFilePrefix + timeArrayString[i]   + inputFileSuffix;
        fullFileNameNext = inputFilePrefix + timeArrayString[i+1] + inputFileSuffix;
        fullFileNamePrev = inputFilePrefix + timeArrayString[i-1] + inputFileSuffix;
        cout << "Full File Name: " << fullFileName << endl;

        cout << "Begin Reading File." << endl;
        cout << "File Format: Fluent" << endl;

        // Reading in the FLUENT 5/6 file to a vtkUnstructuredGrid
        vtkSmartPointer<vtkFLUENTReader> fluent =
            vtkSmartPointer<vtkFLUENTReader>::New();
        fluent->SetFileName(fullFileName.c_str());
        fluent->Update();
        cout << "End Reading File." << endl;

        vtkSmartPointer<vtkFLUENTReader> fluentNext =
            vtkSmartPointer<vtkFLUENTReader>::New();
        fluentNext->SetFileName(fullFileNameNext.c_str());
        fluentNext->Update();

        vtkSmartPointer<vtkFLUENTReader> fluentPrev =
            vtkSmartPointer<vtkFLUENTReader>::New();
        fluentPrev->SetFileName(fullFileNamePrev.c_str());
        fluentPrev->Update();

        // Creating the 'Velocity' array
        vtkSmartPointer<vtkArrayCalculator> arrayCalc =
            vtkSmartPointer<vtkArrayCalculator>::New();
        arrayCalc->AddScalarVariable("X_Velocity", "X_VELOCITY", 0);
```

```
    arrayCalc->AddScalarVariable("Y_Velocity", "Y_VELOCITY", 0);
    arrayCalc->AddScalarVariable("Z_Velocity", "Z_VELOCITY", 0);
    arrayCalc->SetResultArrayName(velocityArrayName);
    arrayCalc->SetFunction("iHat*(X_Velocity) +"
                           "jHat*(Y_Velocity) +"
                           "kHat*(Z_Velocity)");
    arrayCalc->SetInput(fluent->GetOutput()->GetBlock(0));
    arrayCalc->SetAttributeModeToUseCellData();
    arrayCalc->Update();

    vtkSmartPointer<vtkArrayCalculator> arrayCalcNext =
        vtkSmartPointer<vtkArrayCalculator>::New();
    arrayCalcNext->AddScalarVariable("X_Velocity", "X_VELOCITY", 0);
    arrayCalcNext->AddScalarVariable("Y_Velocity", "Y_VELOCITY", 0);
    arrayCalcNext->AddScalarVariable("Z_Velocity", "Z_VELOCITY", 0);
    arrayCalcNext->SetResultArrayName(velocityArrayName);
    arrayCalcNext->SetFunction("iHat*(X_Velocity) +"
                               "jHat*(Y_Velocity) +"
                               "kHat*(Z_Velocity)");
    arrayCalcNext->SetInput(fluentNext->GetOutput()->GetBlock(0));
    arrayCalcNext->SetAttributeModeToUseCellData();
    arrayCalcNext->Update();

    vtkSmartPointer<vtkArrayCalculator> arrayCalcPrev =
        vtkSmartPointer<vtkArrayCalculator>::New();
    arrayCalcPrev->AddScalarVariable("X_Velocity", "X_VELOCITY", 0);
    arrayCalcPrev->AddScalarVariable("Y_Velocity", "Y_VELOCITY", 0);
    arrayCalcPrev->AddScalarVariable("Z_Velocity", "Z_VELOCITY", 0);
    arrayCalcPrev->SetResultArrayName(velocityArrayName);
    arrayCalcPrev->SetFunction("iHat*(X_Velocity) +"
                               "jHat*(Y_Velocity) +"
                               "kHat*(Z_Velocity)");
    arrayCalcPrev->SetInput(fluentPrev->GetOutput()->GetBlock(0));
    arrayCalcPrev->SetAttributeModeToUseCellData();
    arrayCalcPrev->Update();

    // Passing multi-block data set to extraction algorithms
    numberOfBlocks = 1;
    multiBlock->SetNumberOfBlocks(numberOfBlocks);
    for(int j = 0; j < numberOfBlocks ; j++)
      multiBlock->SetBlock(j, arrayCalc->GetOutput());

    numberOfBlocksNext = 1;
    multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
    for(int j = 0; j < numberOfBlocksNext ; j++)
      multiBlockNext->SetBlock(j, arrayCalcNext->GetOutput());

    numberOfBlocksPrev = 1;
    multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
    for(int j = 0; j < numberOfBlocksPrev ; j++)
      multiBlockPrev->SetBlock(j, arrayCalcPrev->GetOutput());
  }

  // Ensight transient reader
  if(strcmp(argv[10],"ensighttransient") == 0)
  {
    // Parsing file names
    inputFileSuffix = ".encas";
    fullFileName     = inputFilePrefix + inputFileSuffix;
    fullFileNameNext = inputFilePrefix + inputFileSuffix;
    fullFileNamePrev = inputFilePrefix + inputFileSuffix;
    cout << "Full File Name: " << fullFileName << endl;

    cout << "Begin Reading File." << endl;
    cout << "File Format: Ensight Transient" << endl;

    // Reading in the ENSIGHT to a vtkUnstructuredGrid
    vtkSmartPointer<vtkGenericEnSightReader> ensightTransientReader =
```

```
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensightTransientReader->SetCaseFileName(fullFileName.c_str());
    ensightTransientReader->SetTimeValue(timeArray[i]);
    ensightTransientReader->Update();

    vtkSmartPointer<vtkGenericEnSightReader> ensightTransientReaderNext =
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensightTransientReaderNext->SetCaseFileName(fullFileNameNext.c_str());
    ensightTransientReaderNext->SetTimeValue(timeArray[i+1]);
    ensightTransientReaderNext->Update();

    vtkSmartPointer<vtkGenericEnSightReader> ensightTransientReaderPrev =
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensightTransientReaderPrev->SetCaseFileName(fullFileNamePrev.c_str());
    ensightTransientReaderPrev->SetTimeValue(timeArray[i-1]);
    ensightTransientReaderPrev->Update();

    cout << "End Reading File." << endl;

    // Passing multi-block data set to extraction algorithms
    numberOfBlocks = 1;
    multiBlock->SetNumberOfBlocks(numberOfBlocks);
    for(int j = 0; j < numberOfBlocks ; j++)
      multiBlock->SetBlock(j, ensightTransientReader->GetOutput()->GetBlock(0));

    numberOfBlocksNext = 1;
    multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
    for(int j = 0; j < numberOfBlocksNext ; j++)
      multiBlockNext->SetBlock(j, ensightTransientReaderNext->GetOutput()->GetBlock(0));

    numberOfBlocksPrev = 1;
    multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
    for(int j = 0; j < numberOfBlocksPrev ; j++)
      multiBlockPrev->SetBlock(j, ensightTransientReaderPrev->GetOutput()->GetBlock(0));
  }

  // Ensight Reader
  if (strcmp(argv[10],"ensight") == 0)
  {
    // Parsing file names
    inputFileSuffix = ".encas";
    fullFileName     = inputFilePrefix + timeArrayString[i]   + inputFileSuffix;
    fullFileNameNext = inputFilePrefix + timeArrayString[i+1] + inputFileSuffix;
    fullFileNamePrev = inputFilePrefix + timeArrayString[i-1] + inputFileSuffix;
    cout << "Full File Name: " << fullFileName << endl;

    cout << "Begin Reading File." << endl;
    cout << "File Format: Ensight" << endl;

    // Reading in the ENSIGHT to a vtkUnstructuredGrid
    vtkSmartPointer<vtkGenericEnSightReader> ensight =
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensight->SetCaseFileName(fullFileName.c_str());
    ensight->Update();

    vtkSmartPointer<vtkGenericEnSightReader> ensightNext =
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensightNext->SetCaseFileName(fullFileNameNext.c_str());
    ensightNext->Update();

    vtkSmartPointer<vtkGenericEnSightReader> ensightPrev =
        vtkSmartPointer<vtkGenericEnSightReader>::New();
    ensightPrev->SetCaseFileName(fullFileNamePrev.c_str());
    ensightPrev->Update();

    cout << "End Reading File." << endl;

    // Passing multi-block data set to extraction algorithms
```

```
      numberOfBlocks = 1;
      multiBlock->SetNumberOfBlocks(numberOfBlocks);
      for(int j = 0; j < numberOfBlocks ; j++)
        multiBlock->SetBlock(j, ensight->GetOutput()->GetBlock(0));

      numberOfBlocksNext = 1;
      multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
      for(int j = 0; j < numberOfBlocksNext ; j++)
        multiBlockNext->SetBlock(j, ensightNext->GetOutput()->GetBlock(0));

      numberOfBlocksPrev = 1;
      multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
      for(int j = 0; j < numberOfBlocksPrev ; j++)
        multiBlockPrev->SetBlock(j, ensightPrev->GetOutput()->GetBlock(0));
    }

    // PLOT3D Reader
    if (strcmp(argv[10],"plot3d") == 0)
    {
      string gridName;

      // Parsing file names
      inputFileSuffix = "q.";
      fullFileName     = inputFilePath + inputFileSuffix + timeArrayString[i];
      fullFileNameNext = inputFilePath + inputFileSuffix + timeArrayString[i+1];
      fullFileNamePrev = inputFilePath + inputFileSuffix + timeArrayString[i-1];
      gridName = inputFilePath + "grid.in";
      cout << "Full File Name: " << fullFileName << endl;

      cout << "Begin Reading File." << endl;
      cout << "File Format: Plot3D" << endl;

      // Converting PLOT3D data set to unstructured grid
      vtkSmartPointer<vtkPLOT3DReader> pl3d =
          vtkSmartPointer<vtkPLOT3DReader>::New();
      pl3d->SetXYZFileName(gridName.c_str());
      pl3d->SetQFileName(fullFileName.c_str());
      pl3d->BinaryFileOn();
      pl3d->IBlankingOn();
      pl3d->AddFunction(100);
      pl3d->AddFunction(110);
      pl3d->AddFunction(210);
      pl3d->AddFunction(200);
      pl3d->Update();

      vtkSmartPointer<vtkPLOT3DReader> pl3dNext =
          vtkSmartPointer<vtkPLOT3DReader>::New();
      pl3dNext->SetXYZFileName(gridName.c_str());
      pl3dNext->SetQFileName(fullFileNameNext.c_str());
      pl3dNext->BinaryFileOn();
      pl3dNext->IBlankingOn();
      pl3dNext->AddFunction(100);
      pl3dNext->AddFunction(110);
      pl3dNext->AddFunction(210);
      pl3dNext->AddFunction(200);
      pl3dNext->Update();

      vtkSmartPointer<vtkPLOT3DReader> pl3dPrev =
          vtkSmartPointer<vtkPLOT3DReader>::New();
      pl3dPrev->SetXYZFileName(gridName.c_str());
      pl3dPrev->SetQFileName(fullFileNamePrev.c_str());
      pl3dPrev->BinaryFileOn();
      pl3dPrev->IBlankingOn();
      pl3dPrev->AddFunction(100);
      pl3dPrev->AddFunction(110);
      pl3dPrev->AddFunction(210);
      pl3dPrev->AddFunction(200);
      pl3dPrev->Update();
```

```
      cout << "End Reading File." << endl;

      // Passing multi-block data set to extraction algorithms
      numberOfBlocks = 1;
      multiBlock->SetNumberOfBlocks(numberOfBlocks);
      for(int j = 0; j < numberOfBlocks ; j++)
        multiBlock->SetBlock(j, pl3d->GetOutput());

      numberOfBlocksNext = 1;
      multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
      for(int j = 0; j < numberOfBlocksNext ; j++)
        multiBlockNext->SetBlock(j, pl3dNext->GetOutput());

      numberOfBlocksPrev = 1;
      multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
      for(int j = 0; j < numberOfBlocksPrev ; j++)
        multiBlockPrev->SetBlock(j, pl3dPrev->GetOutput());
    }

    // OVERFLOW Reader
    if (strcmp(argv[10],"overflow") == 0)
    {
      string gridName, gridNameNext, gridNamePrev;

      // Parsing file names
      inputFileSuffix = "q.";
      fullFileName     = inputFilePath + inputFileSuffix + timeArrayString[i];
      fullFileNameNext = inputFilePath + inputFileSuffix + timeArrayString[i+1];
      fullFileNamePrev = inputFilePath + inputFileSuffix + timeArrayString[i-1];
      if(adaptiveMesh)
      {
        gridName     = inputFilePath + "x." + timeArrayString[i];
        gridNameNext = inputFilePath + "x." + timeArrayString[i+1];
        gridNamePrev = inputFilePath + "x." + timeArrayString[i-1];
      }
      else
      {
        gridName     = inputFilePath + "grid.in";
        gridNameNext = inputFilePath + "grid.in";
        gridNamePrev = inputFilePath + "grid.in";
      }

      cout << "Full File Name: " << fullFileName << endl;

      cout << "Begin Reading File." << endl;
      cout << "File Format: OverFlow" << endl;

      // Reading multi-block overflow data set
      vtkSmartPointer<vtkMultiBlockOVERFLOWReader> oReader =
          vtkSmartPointer<vtkMultiBlockOVERFLOWReader>::New();
      oReader->SetXYZFileName(gridName.c_str());
      oReader->SetQFileName(fullFileName.c_str());
      oReader->AddFunction(100);
      oReader->AddFunction(110);
      oReader->AddFunction(210);
      oReader->AddFunction(200);
      oReader->AutoSetFileProperties();
      oReader->Update();
      cout << "End Reading File." << endl;

      vtkSmartPointer<vtkMultiBlockOVERFLOWReader> oReaderNext =
          vtkSmartPointer<vtkMultiBlockOVERFLOWReader>::New();
      oReaderNext->SetXYZFileName(gridNameNext.c_str());
      oReaderNext->SetQFileName(fullFileNameNext.c_str());
      oReaderNext->AddFunction(100);
      oReaderNext->AddFunction(110);
      oReaderNext->AddFunction(210);
```

```
      oReaderNext->AddFunction(200);
      oReaderNext->AutoSetFileProperties();
      oReaderNext->Update();

      vtkSmartPointer<vtkMultiBlockOVERFLOWReader> oReaderPrev =
          vtkSmartPointer<vtkMultiBlockOVERFLOWReader>::New();
      oReaderPrev->SetXYZFileName(gridNamePrev.c_str());
      oReaderPrev->SetQFileName(fullFileNamePrev.c_str());
      oReaderPrev->AddFunction(100);
      oReaderPrev->AddFunction(110);
      oReaderPrev->AddFunction(210);
      oReaderPrev->AddFunction(200);
      oReaderPrev->AutoSetFileProperties();
      oReaderPrev->Update();

      // Passing multi-block data set to extraction algorithms
      numberOfBlocks = oReader->GetOutput()->GetNumberOfBlocks();
      multiBlock->SetNumberOfBlocks(numberOfBlocks);
      for(int j = 0; j < numberOfBlocks ; j++)
        multiBlock->SetBlock(j, oReader->GetOutput()->GetBlock(j));

      numberOfBlocksNext = oReaderNext->GetOutput()->GetNumberOfBlocks();
      multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
      for(int j = 0; j < numberOfBlocksNext ; j++)
        multiBlockNext->SetBlock(j, oReaderNext->GetOutput()->GetBlock(j));

      numberOfBlocksPrev = oReaderPrev->GetOutput()->GetNumberOfBlocks();
      multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
      for(int j = 0; j < numberOfBlocksPrev ; j++)
        multiBlockPrev->SetBlock(j, oReaderPrev->GetOutput()->GetBlock(j));
    }

    //VTK Reader
    if (strcmp(argv[10],"vtk") == 0)
    {
      // Parsing file names
      inputFileSuffix = ".vtk";
      fullFileName     = inputFilePrefix + timeArrayString[i]   + inputFileSuffix;
      fullFileNameNext = inputFilePrefix + timeArrayString[i+1] + inputFileSuffix;
      fullFileNamePrev = inputFilePrefix + timeArrayString[i-1] + inputFileSuffix;
      cout << "Full File Name: " << fullFileName << endl;

      cout << "Begin Reading File." << endl;
      cout << "File Format: VTK" << endl;

      // Reading in the vtk file to a vtkUnstructuredGrid
      vtkSmartPointer<vtkUnstructuredGridReader> vtkReader =
          vtkSmartPointer<vtkUnstructuredGridReader>::New();
      vtkReader->SetFileName(fullFileName.c_str());
      vtkReader->Update();

      vtkSmartPointer<vtkUnstructuredGridReader> vtkReaderNext =
          vtkSmartPointer<vtkUnstructuredGridReader>::New();
      vtkReaderNext->SetFileName(fullFileNameNext.c_str());
      vtkReaderNext->Update();

      vtkSmartPointer<vtkUnstructuredGridReader> vtkReaderPrev =
          vtkSmartPointer<vtkUnstructuredGridReader>::New();
      vtkReaderPrev->SetFileName(fullFileNamePrev.c_str());
      vtkReaderPrev->Update();

      cout << "End Reading File." << endl;

      // Passing multi-block data set to extraction algorithms
      numberOfBlocks = 1;
      multiBlock->SetNumberOfBlocks(numberOfBlocks);
      for(int j = 0; j < numberOfBlocks ; j++)
        multiBlock->SetBlock(j, vtkReader->GetOutput());
```

```
      numberOfBlocksNext = 1;
      multiBlockNext->SetNumberOfBlocks(numberOfBlocksNext);
      for(int j = 0; j < numberOfBlocksNext ; j++)
        multiBlockNext->SetBlock(j, vtkReaderNext->GetOutput());

      numberOfBlocksPrev = 1;
      multiBlockPrev->SetNumberOfBlocks(numberOfBlocksPrev);
      for(int j = 0; j < numberOfBlocksPrev ; j++)
        multiBlockPrev->SetBlock(j, vtkReaderPrev->GetOutput());
    }

    cout << "VORTEX CORE FILES:\n";
    outputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
    outputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
    cout << "\tSH output file: " << outputFileNameSH << endl;
    cout << "\tRP output file: " << outputFileNameRP << endl;
    if(writeDataSet)
    {
      outputFileNameDataSet = outputFilePrefix + "_" + timeArrayString[i] + "_DataSet.vtk";
      cout << "\tData set output file: " << outputFileNameDataSet << endl;
    }
    cout << endl;

    // Setting up append filters for each extraction type
    vtkSmartPointer<vtkAppendPolyData> appendSH = vtkSmartPointer<vtkAppendPolyData>::New();
    vtkSmartPointer<vtkAppendPolyData> appendRP = vtkSmartPointer<vtkAppendPolyData>::New();

    // Iterating through all blocks of data set
    for(int j = 0 ; j < numberOfBlocks ; j++)
    {
      // Converting cell data to point data
      vtkSmartPointer<vtkCellDataToPointData> c2p =
          vtkSmartPointer<vtkCellDataToPointData>::New();
      c2p->SetInput(multiBlock->GetBlock(j));
      c2p->Update();

      // Calculate velocity magnitude
      vtkSmartPointer<vtkArrayCalculator> velMagCalc =
          vtkSmartPointer<vtkArrayCalculator>::New();
      velMagCalc->AddVectorVariable("Velocity",velocityArrayName);
      velMagCalc->SetResultArrayName("VelocityMagnitude");
      velMagCalc->SetFunction("mag(Velocity)");

      // Computing time derivatives in areas where mesh adaption does not occur
      if(adaptiveMesh && j > numBlocksToDerive-1)
      {
        velMagCalc->SetInput(c2p->GetOutput());
        velMagCalc->Update();
      }
      else
      {
        vtkSmartPointer<vtkCellDataToPointData> c2pNext =
            vtkSmartPointer<vtkCellDataToPointData>::New();
        c2pNext->SetInput(multiBlockNext->GetBlock(j));
        c2pNext->Update();

        vtkSmartPointer<vtkCellDataToPointData> c2pPrev =
            vtkSmartPointer<vtkCellDataToPointData>::New();
        c2pPrev->SetInput(multiBlockPrev->GetBlock(j));
        c2pPrev->Update();

        cout << "Computing time derivatives." << endl;
        vtkSmartPointer<vtkTimeDerivatives> timeDer =
            vtkSmartPointer<vtkTimeDerivatives>::New();
        timeDer->AddInputConnection(c2p->GetOutputPort());
        timeDer->AddInputConnection(c2pNext->GetOutputPort());
        timeDer->AddInputConnection(c2pPrev->GetOutputPort());
```

```
    if(timeStepPhys)
      timeDer->SetTimeStep(dtPhys);
    else
      timeDer->SetTimeStep(timeStep);
    timeDer->SetVelocity1ArrayName(velocityArrayName);
    timeDer->SetVelocity2ArrayName(velocityArrayName);
    timeDer->SetVelocity3ArrayName(velocityArrayName);
    timeDer->ForwardDifferenceOff();
    timeDer->BackwardDifferenceOff();
    timeDer->CentralDifferenceOn();
    timeDer->Update();

    velMagCalc->SetInput(timeDer->GetOutput());
    velMagCalc->Update();
  }

  cout << "Extracting Vortex Core Lines.\n";

  // Thresholding to ignore low-velocity regions, i.e. walls
  cout << "\tThresholding out wall cells." << endl;
  vtkSmartPointer<vtkThreshold> threshWalls = vtkSmartPointer<vtkThreshold>::New();
  threshWalls->SetInput(velMagCalc->GetOutput());
  threshWalls->ThresholdByUpper(0.001);
  threshWalls->AllScalarsOff();
  threshWalls->SetInputArrayToProcess(0, 0, 0,
      vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityMagnitude");
  threshWalls->Update();

  // Computing lambda_2 at each point in the data set
  cout << "\tComputing lambda_2." << endl;
  vtkSmartPointer<vtkLambdaTwo> l2 = vtkSmartPointer<vtkLambdaTwo>::New();
  l2->SetInput(threshWalls->GetOutput());
  l2->SetVelocityArrayName(velocityArrayName);
  l2->Update();

  // Data set writing option
  if(writeDataSet)
  {
    // Compute vortex strength in data set
    vtkSmartPointer<vtkVortexStrength> strength1 =
      vtkSmartPointer<vtkVortexStrength>::New();
    strength1->SetInput(l2->GetOutput());
    strength1->SetInputArrayToProcess(0, 0, 0,
      vtkDataObject::FIELD_ASSOCIATION_POINTS, velocityArrayName);
    strength1->SetInputArrayToProcess(1, 0, 0,
      vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityGradients");
    strength1->Update();

    // Writing data set
    vtkSmartPointer<vtkUnstructuredGridWriter> writer1 =
      vtkSmartPointer<vtkUnstructuredGridWriter>::New();
    writer1->SetInput(strength1->GetOutput());
    writer1->SetFileName(outputFileNameDataSet.c_str());
    writer1->Write();
  }

  // Extracting corelines using vtkRothPeikert
  // need to have a data set with point data as input and a velocity vector not
  // velocity as three separate scalar components.
  cout << "\t***ROTH-PEIKERT***" << endl;
  vtkSmartPointer<vtkRothPeikert> rothPeikert =
      vtkSmartPointer<vtkRothPeikert>::New();
  rothPeikert->SetInput(l2->GetOutput());
  rothPeikert->SetVelocityArrayName(velocityArrayName);
  rothPeikert->SetMinimumNumberOfPoints(minimumCorePoints);
  if(adaptiveMesh && j > numBlocksToDerive-1)
    rothPeikert->SetTransient(false);
  else
```

```
        rothPeikert->SetTransient(true);
      rothPeikert->SetVerbose(verbose);
      rothPeikert->Update();

      // Appending results of current block to append filter
      appendRP->AddInput(rothPeikert->GetOutput());

      // Extracting corelines using vtkSujudiHaimes
      // need to have a data set with point data as input and a velocity vector not
      // velocity as three separate scalar components.
      cout << "\t***SUJUDI-HAIMES***" << endl;
      vtkSmartPointer<vtkSujudiHaimes> sujudiHaimes =
          vtkSmartPointer<vtkSujudiHaimes>::New();
      sujudiHaimes->SetInput(l2->GetOutput());
      sujudiHaimes->SetVelocityArrayName(velocityArrayName);
      sujudiHaimes->SetMinimumNumberOfPoints(minimumCorePoints);
      if(adaptiveMesh && j > numBlocksToDerive-1)
        sujudiHaimes->SetTransient(false);
      else
        sujudiHaimes->SetTransient(true);
      sujudiHaimes->SetVerbose(verbose);
      sujudiHaimes->Update();

      // Appending results of current block to append filter
      appendSH->AddInput(sujudiHaimes->GetOutput());

      cout << endl;

    // cleaning the input data set
    vtkSmartPointer<vtkCleanPolyData> clean1 =
        vtkSmartPointer<vtkCleanPolyData>::New();
    clean1->SetInput(appendRP->GetOutput());
    clean1->Update();

    // Calculating vortex strength
    vtkSmartPointer<vtkVortexStrength> vortexStrength1 =
        vtkSmartPointer<vtkVortexStrength>::New();
    vortexStrength1->SetInput(clean1->GetOutput());
    vortexStrength1->SetInputArrayToProcess(0, 0, 0,
        vtkDataObject::FIELD_ASSOCIATION_POINTS, velocityArrayName);
    vortexStrength1->SetInputArrayToProcess(1, 0, 0,
        vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityGradients");
    vortexStrength1->Update();

    // Computing the quality of the vortices
    vtkSmartPointer<vtkQuality> quality1 =
        vtkSmartPointer<vtkQuality>::New();
    quality1->SetInput(vortexStrength1->GetOutput());
    quality1->SetThresholdLines(thresholdLines);
    quality1->SetQualityThresholdValue(qualityThresholdValue);
    quality1->SetVelocityArrayName(velocityArrayName);
    quality1->SetConvectiveCorrection(true);
    quality1->Update();

    // Paramaterizing line segments
    // each line segment has an a,b,c,d,e,f and l associated value
    vtkSmartPointer<vtkParamaterizeLineFilter> plf1 =
        vtkSmartPointer<vtkParamaterizeLineFilter>::New();
    plf1->SetInput(quality1->GetOutput());
    plf1->Update();

    // calculating the curvature of the line
    vtkSmartPointer<vtkCurvature> curvature1 =
        vtkSmartPointer<vtkCurvature>::New();
    curvature1->SetInput(plf1->GetOutput());
    curvature1->MultiSegmentCurvatureOn();
    curvature1->VelocityFieldCurvatureOff();
    curvature1->PointwiseCurvatureOff();
```

```
curvature1->Update();

// Computing feature-averaged attributes
vtkSmartPointer<vtkFeatureAttributes> featureAttributes1 =
    vtkSmartPointer<vtkFeatureAttributes>::New();
featureAttributes1->SetInput(curvature1->GetOutput());
featureAttributes1->Update();

// writing the connected lines to RP.vtk
vtkSmartPointer<vtkPolyDataWriter> writer1 =
    vtkSmartPointer<vtkPolyDataWriter>::New();
writer1->SetInput(featureAttributes1->GetOutput());
writer1->SetFileName(outputFileNameRP.c_str());
writer1->Write();
cout << "Writing " << outputFileNameRP.c_str() << endl;

// cleaning the input data set
vtkSmartPointer<vtkCleanPolyData> clean2 =
    vtkSmartPointer<vtkCleanPolyData>::New();
clean2->SetInput(appendSH->GetOutput());
clean2->Update();

// Calculating vortex strength
vtkSmartPointer<vtkVortexStrength> vortexStrength2 =
    vtkSmartPointer<vtkVortexStrength>::New();
vortexStrength2->SetInput(clean2->GetOutput());
vortexStrength2->SetInputArrayToProcess(0, 0, 0,
    vtkDataObject::FIELD_ASSOCIATION_POINTS, velocityArrayName);
vortexStrength2->SetInputArrayToProcess(1, 0, 0,
    vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityGradients");
vortexStrength2->Update();

// Computing the quality of the vortices
vtkSmartPointer<vtkQuality> quality2 =
    vtkSmartPointer<vtkQuality>::New();
quality2->SetInput(vortexStrength2->GetOutput());
quality2->SetThresholdLines(thresholdLines);
quality2->SetQualityThresholdValue(qualityThresholdValue);
quality2->SetVelocityArrayName(velocityArrayName);
quality2->SetConvectiveCorrection(true);
quality2->Update();

// Paramaterizing line segments
// each line segment has an a,b,c,d,e,f and l associated value
vtkSmartPointer<vtkParamaterizeLineFilter> plf2 =
    vtkSmartPointer<vtkParamaterizeLineFilter>::New();
plf2->SetInput(quality2->GetOutput());
plf2->Update();

// calculating the curvature of the line
vtkSmartPointer<vtkCurvature> curvature2 =
    vtkSmartPointer<vtkCurvature>::New();
curvature2->SetInput(plf2->GetOutput());
curvature2->MultiSegmentCurvatureOn();
curvature2->VelocityFieldCurvatureOff();
curvature2->PointwiseCurvatureOff();
curvature2->Update();

// Computing feature-averaged attributes
vtkSmartPointer<vtkFeatureAttributes> featureAttributes2 =
    vtkSmartPointer<vtkFeatureAttributes>::New();
featureAttributes2->SetInput(curvature2->GetOutput());
featureAttributes2->Update();

// writing extracted lines from Sujudi-Haimes
vtkSmartPointer<vtkPolyDataWriter> writer2 =
    vtkSmartPointer<vtkPolyDataWriter>::New();
writer2->SetInput(featureAttributes2->GetOutput());
```

```
      writer2->SetFileName(outputFileNameSH.c_str());
      writer2->Write();
      cout << "Writing " << outputFileNameSH.c_str() << endl;
    }
  }

  /***********************PERFORMING FEATURE TRACKING***********************/
  if(track)
  {
    // Calculating feature attributes
    cout << "*******Calculating feature attributes.*************" << endl << endl;
    for(int i = 1 ; i < numberOfDataSets-1 ; ++i)
    {
      string inputFileNameSH, inputFileNameRP,
             outputFileNameSH, outputFileNameRP;

      // getting correct names for the input/output files
      inputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      outputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      inputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
      outputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";

      ////****Sujudi-Haimes Section****////
      // Reading in data set
      vtkSmartPointer<vtkPolyDataReader> reader1 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      reader1->SetFileName(inputFileNameSH.c_str());
      reader1->Update();

      // Finding number of SH core lines
      numLinesSH += reader1->GetOutput()->GetNumberOfLines();

      // Calculating the curvature of the line
      vtkSmartPointer<vtkCurvature> curvature1 =
          vtkSmartPointer<vtkCurvature>::New();
      curvature1->SetInput(reader1->GetOutput());
      curvature1->MultiSegmentCurvatureOn();
      curvature1->VelocityFieldCurvatureOff();
      curvature1->PointwiseCurvatureOff();
      curvature1->Update();

      // compute vortex strength
      vtkSmartPointer<vtkVortexStrength> vortexStrength1 =
          vtkSmartPointer<vtkVortexStrength>::New();
      vortexStrength1->SetInput(curvature1->GetOutput());
      vortexStrength1->SetInputArrayToProcess(0, 0, 0,
          vtkDataObject::FIELD_ASSOCIATION_POINTS, velocityArrayName);
      vortexStrength1->SetInputArrayToProcess(1, 0, 0,
          vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityGradients");
      vortexStrength1->Update();

      // Calculating feature attributes
      vtkSmartPointer<vtkFeatureAttributes> attributes1 =
          vtkSmartPointer<vtkFeatureAttributes>::New();
      attributes1->SetInput(vortexStrength1->GetOutput());
      attributes1->Update();

      // Writing data set
      vtkSmartPointer<vtkPolyDataWriter> writer1 =
          vtkSmartPointer<vtkPolyDataWriter>::New();
      writer1->SetInput(attributes1->GetOutput());
      writer1->SetFileName(outputFileNameSH.c_str());
      writer1->Write();

      ////****Roth-Peikert Section****////
      // Reading in data set
      vtkSmartPointer<vtkPolyDataReader> reader2 =
          vtkSmartPointer<vtkPolyDataReader>::New();
```

```
  reader2->SetFileName(inputFileNameRP.c_str());
  reader2->Update();

  // Finding number of RP core lines
  numLinesRP += reader2->GetOutput()->GetNumberOfLines();

  // Calculating the curvature of the line
  vtkSmartPointer<vtkCurvature> curvature2 =
      vtkSmartPointer<vtkCurvature>::New();
  curvature2->SetInput(reader2->GetOutput());
  curvature2->MultiSegmentCurvatureOn();
  curvature2->VelocityFieldCurvatureOff();
  curvature2->PointwiseCurvatureOff();
  curvature2->Update();

  // compute vortex strength
  vtkSmartPointer<vtkVortexStrength> vortexStrength2 =
      vtkSmartPointer<vtkVortexStrength>::New();
  vortexStrength2->SetInput(curvature2->GetOutput());
  vortexStrength2->SetInputArrayToProcess(0, 0, 0,
      vtkDataObject::FIELD_ASSOCIATION_POINTS, velocityArrayName);
  vortexStrength2->SetInputArrayToProcess(1, 0, 0,
      vtkDataObject::FIELD_ASSOCIATION_POINTS, "VelocityGradients");
  vortexStrength2->Update();

  // Calculating feature attributes
  vtkSmartPointer<vtkFeatureAttributes> attributes2 =
      vtkSmartPointer<vtkFeatureAttributes>::New();
  attributes2->SetInput(vortexStrength2->GetOutput());
  attributes2->Update();

  // Writing data set
  vtkSmartPointer<vtkPolyDataWriter> writer2 =
      vtkSmartPointer<vtkPolyDataWriter>::New();
  writer2->SetInput(attributes2->GetOutput());
  writer2->SetFileName(outputFileNameRP.c_str());
  writer2->Write();
}

//----------------------------------------------------------------------------
// Tracking features by attributes

// inputs for tracking
//----------------------------------------------------------------------------
int maxTrackingIdSH = 0;
int maxTrackingIdRP = 0;
//----------------------------------------------------------------------------

// getting correct names for the files
string currentFileNameSH, prevFileNameSH, nextFileNameSH,
       currentOutputFileNameSH, nextOutputFileNameSH,
       currentFileNameRP, prevFileNameRP, nextFileNameRP,
       currentOutputFileNameRP, nextOutputFileNameRP;

cout << "*************Tracking Features.*******************" << endl << endl;
cout << "Number of passes: " << numberOfPasses << endl;
// Multiple passes
for(int p = 0 ; p < numberOfPasses ; p++)
{
  cout << "Pass " << p << endl;
  // Forward pass through data sets
  for(int i = 1 ; i < numberOfDataSets-2 ; ++i)
  {
    cout << "\tTime = " << timeArrayString[i] << endl;
    currentFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
    if(i == 1)
      prevFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
    else
```

```
      prevFileNameSH = outputFilePrefix + "_" + timeArrayString[i-1] + "_SH.vtk";
    nextFileNameSH = outputFilePrefix + "_" + timeArrayString[i+1] + "_SH.vtk";
    currentOutputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
    nextOutputFileNameSH = outputFilePrefix + "_" + timeArrayString[i+1] + "_SH.vtk";

    currentFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
    if(i == 1)
      prevFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
    else
      prevFileNameRP = outputFilePrefix + "_" + timeArrayString[i-1] + "_RP.vtk";
    nextFileNameRP = outputFilePrefix + "_" + timeArrayString[i+1] + "_RP.vtk";
    currentOutputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
    nextOutputFileNameRP = outputFilePrefix + "_" + timeArrayString[i+1] + "_RP.vtk";

    ////****Sujudi-Haimes Section****////
    // Reading in time step of interest
    vtkSmartPointer<vtkPolyDataReader> polyReader1 =
        vtkSmartPointer<vtkPolyDataReader>::New();
    polyReader1->SetFileName(currentFileNameSH.c_str());
    polyReader1->Update();

    // Reading in next time step
    vtkSmartPointer<vtkPolyDataReader> polyReader2 =
        vtkSmartPointer<vtkPolyDataReader>::New();
    polyReader2->SetFileName(nextFileNameSH.c_str());
    polyReader2->Update();

    // Reading in prev time step
    vtkSmartPointer<vtkPolyDataReader> polyReader3 =
        vtkSmartPointer<vtkPolyDataReader>::New();
    polyReader3->SetFileName(prevFileNameSH.c_str());
    polyReader3->Update();

    // Tracking lines
    vtkSmartPointer<vtkAttributeTracking> tracker1 =
        vtkSmartPointer<vtkAttributeTracking>::New();
    tracker1->AddInputConnection(polyReader1->GetOutputPort());
    tracker1->AddInputConnection(polyReader2->GetOutputPort());
    tracker1->AddInputConnection(polyReader3->GetOutputPort());
    if(i == 1)
      tracker1->BoundaryDataSetOn();
    else
      tracker1->BoundaryDataSetOff();
    tracker1->ForwardPassOn();
    tracker1->BackwardPassOff();
    tracker1->SetMaximumTrackingID(maxTrackingIdSH);
    tracker1->SetLengthTolerance(lengthTolerance);
    tracker1->SetStrengthTolerance(strengthTolerance);
    tracker1->SetCurvatureTolerance(curvatureTolerance);
    tracker1->SetQualityTolerance(qualityTolerance);
    tracker1->SetDistanceTolerance(distanceTolerance);
    tracker1->SetLengthWeight(lengthWeight);
    tracker1->SetStrengthWeight(strengthWeight);
    tracker1->SetCurvatureWeight(curvatureWeight);
    tracker1->SetQualityWeight(qualityWeight);
    tracker1->SetDistanceWeight(distanceWeight);
    tracker1->Update();

    // Incrementing maximum tracking ID for current pass
    maxTrackingIdSH = tracker1->GetMaximumTrackingID();

    // Writing tracking results for current time step
    vtkSmartPointer<vtkPolyDataWriter> writer1 =
        vtkSmartPointer<vtkPolyDataWriter>::New();
    writer1->SetInput(tracker1->GetOutput(0));
    writer1->SetFileName(currentOutputFileNameSH.c_str());
    writer1->Write();
```

```
        // Writing tracking results for next time step
        vtkSmartPointer<vtkPolyDataWriter> writer2 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer2->SetInput(tracker1->GetOutput(1));
        writer2->SetFileName(nextOutputFileNameSH.c_str());
        writer2->Write();

        ////****Roth-Peikert Section****////
        // Reading in time step of interest
        vtkSmartPointer<vtkPolyDataReader> polyReader4 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader4->SetFileName(currentFileNameRP.c_str());
        polyReader4->Update();

        // Reading in next time step
        vtkSmartPointer<vtkPolyDataReader> polyReader5 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader5->SetFileName(nextFileNameRP.c_str());
        polyReader5->Update();

        // Reading in prev time step
        vtkSmartPointer<vtkPolyDataReader> polyReader6 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader6->SetFileName(prevFileNameRP.c_str());
        polyReader6->Update();

        // Tracking lines
        vtkSmartPointer<vtkAttributeTracking> tracker2 =
            vtkSmartPointer<vtkAttributeTracking>::New();
        tracker2->AddInputConnection(polyReader4->GetOutputPort());
        tracker2->AddInputConnection(polyReader5->GetOutputPort());
        tracker2->AddInputConnection(polyReader6->GetOutputPort());
        if(i == 1)
          tracker2->BoundaryDataSetOn();
        else
          tracker2->BoundaryDataSetOff();
        tracker2->ForwardPassOn();
        tracker2->BackwardPassOff();
        tracker2->SetMaximumTrackingID(maxTrackingIdRP);
        tracker2->SetLengthTolerance(lengthTolerance);
        tracker2->SetStrengthTolerance(strengthTolerance);
        tracker2->SetCurvatureTolerance(curvatureTolerance);
        tracker2->SetQualityTolerance(qualityTolerance);
        tracker2->SetDistanceTolerance(distanceTolerance);
        tracker2->SetLengthWeight(lengthWeight);
        tracker2->SetStrengthWeight(strengthWeight);
        tracker2->SetCurvatureWeight(curvatureWeight);
        tracker2->SetQualityWeight(qualityWeight);
        tracker2->SetDistanceWeight(distanceWeight);
        tracker2->Update();

        // Updating maximum tracking ID from most recent pass
        maxTrackingIdRP = tracker2->GetMaximumTrackingID();

        // Writing tracking results of current time step
        vtkSmartPointer<vtkPolyDataWriter> writer3 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer3->SetInput(tracker2->GetOutput(0));
        writer3->SetFileName(currentOutputFileNameRP.c_str());
        writer3->Write();

        // Writing tracking results of next time step
        vtkSmartPointer<vtkPolyDataWriter> writer4 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer4->SetInput(tracker2->GetOutput(1));
        writer4->SetFileName(nextOutputFileNameRP.c_str());
        writer4->Write();
      }
```

```
    // Backward pass through data sets
    for(int i = numberOfDataSets-2 ; i > 1 ; i--)
    {
      cout << "\tTime = " << timeArrayString[i] << endl;
      currentFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      if(i == numberOfDataSets-2)
        prevFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      else
        prevFileNameSH = outputFilePrefix + "_" + timeArrayString[i+1] + "_SH.vtk";
      nextFileNameSH = outputFilePrefix + "_" + timeArrayString[i-1] + "_SH.vtk";
      currentOutputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      nextOutputFileNameSH = outputFilePrefix + "_" + timeArrayString[i-1] + "_SH.vtk";

      currentFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
      if(i == numberOfDataSets-2)
        prevFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
      else
        prevFileNameRP = outputFilePrefix + "_" + timeArrayString[i+1] + "_RP.vtk";
      nextFileNameRP = outputFilePrefix + "_" + timeArrayString[i-1] + "_RP.vtk";
      currentOutputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
      nextOutputFileNameRP = outputFilePrefix + "_" + timeArrayString[i-1] + "_RP.vtk";

      ////****Sujudi-Haimes Section****////
      // Reading in time step of interest
      vtkSmartPointer<vtkPolyDataReader> polyReader1 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      polyReader1->SetFileName(currentFileNameSH.c_str());
      polyReader1->Update();

      // Reading in next time step
      vtkSmartPointer<vtkPolyDataReader> polyReader2 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      polyReader2->SetFileName(nextFileNameSH.c_str());
      polyReader2->Update();

      // Reading in prev time step
      vtkSmartPointer<vtkPolyDataReader> polyReader3 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      polyReader3->SetFileName(prevFileNameSH.c_str());
      polyReader3->Update();

      // Tracking lines
      vtkSmartPointer<vtkAttributeTracking> tracker1 =
          vtkSmartPointer<vtkAttributeTracking>::New();
      tracker1->AddInputConnection(polyReader1->GetOutputPort());
      tracker1->AddInputConnection(polyReader2->GetOutputPort());
      tracker1->AddInputConnection(polyReader3->GetOutputPort());
      if(i == numberOfDataSets-2)
        tracker1->BoundaryDataSetOn();
      else
        tracker1->BoundaryDataSetOff();
      tracker1->ForwardPassOff();
      tracker1->BackwardPassOn();
      tracker1->SetMaximumTrackingID(maxTrackingIdSH);
      tracker1->SetLengthTolerance(lengthTolerance);
      tracker1->SetStrengthTolerance(strengthTolerance);
      tracker1->SetCurvatureTolerance(curvatureTolerance);
      tracker1->SetQualityTolerance(qualityTolerance);
      tracker1->SetDistanceTolerance(distanceTolerance);
      tracker1->SetLengthWeight(lengthWeight);
      tracker1->SetStrengthWeight(strengthWeight);
      tracker1->SetCurvatureWeight(curvatureWeight);
      tracker1->SetQualityWeight(qualityWeight);
      tracker1->SetDistanceWeight(distanceWeight);
      tracker1->Update();

      // Updating maximum tracking ID from most recent pass
```

```
        maxTrackingIdSH = tracker1->GetMaximumTrackingID();

        // Writing tracking results of current time step
        vtkSmartPointer<vtkPolyDataWriter> writer1 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer1->SetInput(tracker1->GetOutput(0));
        writer1->SetFileName(currentOutputFileNameSH.c_str());
        writer1->Write();

        // Writing tracking results of next time step
        vtkSmartPointer<vtkPolyDataWriter> writer2 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer2->SetInput(tracker1->GetOutput(1));
        writer2->SetFileName(nextOutputFileNameSH.c_str());
        writer2->Write();

        ////****Roth-Peikert Section****////
        // Reading in time step of interest
        vtkSmartPointer<vtkPolyDataReader> polyReader4 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader4->SetFileName(currentFileNameRP.c_str());
        polyReader4->Update();

        // Reading in next time step
        vtkSmartPointer<vtkPolyDataReader> polyReader5 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader5->SetFileName(nextFileNameRP.c_str());
        polyReader5->Update();

        // Reading in previous time step
        vtkSmartPointer<vtkPolyDataReader> polyReader6 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader6->SetFileName(prevFileNameRP.c_str());
        polyReader6->Update();

        // Tracking lines
        vtkSmartPointer<vtkAttributeTracking> tracker2 =
            vtkSmartPointer<vtkAttributeTracking>::New();
        tracker2->AddInputConnection(polyReader4->GetOutputPort());
        tracker2->AddInputConnection(polyReader5->GetOutputPort());
        tracker2->AddInputConnection(polyReader6->GetOutputPort());
        if(i == numberOfDataSets-2)
          tracker2->BoundaryDataSetOn();
        else
          tracker2->BoundaryDataSetOff();
        tracker2->ForwardPassOff();
        tracker2->BackwardPassOn();
        tracker2->SetMaximumTrackingID(maxTrackingIdRP);
        tracker2->SetLengthTolerance(lengthTolerance);
        tracker2->SetStrengthTolerance(strengthTolerance);
        tracker2->SetCurvatureTolerance(curvatureTolerance);
        tracker2->SetQualityTolerance(qualityTolerance);
        tracker2->SetDistanceTolerance(distanceTolerance);
        tracker2->SetLengthWeight(lengthWeight);
        tracker2->SetStrengthWeight(strengthWeight);
        tracker2->SetCurvatureWeight(curvatureWeight);
        tracker2->SetQualityWeight(qualityWeight);
        tracker2->SetDistanceWeight(distanceWeight);
        tracker2->Update();

        // Updating maximum tracking ID from most recent pass
        maxTrackingIdRP = tracker2->GetMaximumTrackingID();

        // Writing tracking results of current time step
        vtkSmartPointer<vtkPolyDataWriter> writer3 =
            vtkSmartPointer<vtkPolyDataWriter>::New();
        writer3->SetInput(tracker2->GetOutput(0));
        writer3->SetFileName(currentOutputFileNameRP.c_str());
```

```
      writer3->Write();

      // Writing tracking results of next time step
      vtkSmartPointer<vtkPolyDataWriter> writer4 =
          vtkSmartPointer<vtkPolyDataWriter>::New();
      writer4->SetInput(tracker2->GetOutput(1));
      writer4->SetFileName(nextOutputFileNameRP.c_str());
      writer4->Write();
    }

    // Incrementing tracking tolerances for next pass
    lengthTolerance = lengthTolerance + lengthIncr;
    strengthTolerance = strengthTolerance + strengthIncr;
    curvatureTolerance = curvatureTolerance + curvatureIncr;
    qualityTolerance = qualityTolerance + qualityIncr;
    distanceTolerance = distanceTolerance + distanceIncr;
  }

  cout << "SH Maximum Tracking ID = " << maxTrackingIdSH << endl
       << "RP Maximum Tracking ID = " << maxTrackingIdRP << endl
       << endl;

  //--------------------------------------------------------------------------------
  // Measuring lifetime of feature paths

  // getting correct names for the files
  string inputFileNameSH, inputFileNameRP;

  // Instantiating lifetime arrays
  vtkIntArray *lifetimeArraySH = vtkIntArray::New();
  lifetimeArraySH->SetNumberOfValues(maxTrackingIdSH+1);
  lifetimeArraySH->SetNumberOfComponents(1);
  lifetimeArraySH->SetNumberOfTuples(maxTrackingIdSH+1);
  lifetimeArraySH->SetName("FeatureLifetimeSH");

  vtkIntArray *lifetimeArrayRP = vtkIntArray::New();
  lifetimeArrayRP->SetNumberOfValues(maxTrackingIdRP+1);
  lifetimeArrayRP->SetNumberOfComponents(1);
  lifetimeArrayRP->SetNumberOfTuples(maxTrackingIdRP+1);
  lifetimeArrayRP->SetName("FeatureLifetimeRP");

  // Initializing lifetime arrays
  // each index initially has a lifetime of 0
  for(int i = 0; i < maxTrackingIdSH; ++i)
    lifetimeArraySH->SetValue(i,0);
  for(int i = 0; i < maxTrackingIdRP; ++i)
    lifetimeArrayRP->SetValue(i,0);

  cout << "***********Measuring feature lifetimes.*************" << endl << endl;
  // Iterating through time steps to find feature lifetimes
  for(int i = 1; i < numberOfDataSets-1; ++i)
  {
    cout << "Time = " << timeArrayString[i] << endl;

    // Getting file names for input files
    inputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
    inputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";

    ////****Sujudi-Haimes Section****////
    // Reading input file
    vtkSmartPointer<vtkPolyDataReader> reader1 =
        vtkSmartPointer<vtkPolyDataReader>::New();
    reader1->SetFileName(inputFileNameSH.c_str());
    reader1->Update();

    // Passing data set to calculate feature lifetime
    vtkSmartPointer<vtkFeatureLifetime> lifeCalc1 =
        vtkSmartPointer<vtkFeatureLifetime>::New();
```

```
      lifeCalc1->SetInput(reader1->GetOutput());
      lifeCalc1->SetFeatureLifeArray(lifetimeArraySH);
      lifeCalc1->CalculateFeatureLifetimeOn();
      lifeCalc1->SetFeatureLifetimeOff();
      lifeCalc1->Update();

      ////****Roth-Peikert Section****////
      // Reading input file
      vtkSmartPointer<vtkPolyDataReader> reader2 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      reader2->SetFileName(inputFileNameRP.c_str());
      reader2->Update();

      // Passing data set to calculate feature lifetime
      vtkSmartPointer<vtkFeatureLifetime> lifeCalc2 =
          vtkSmartPointer<vtkFeatureLifetime>::New();
      lifeCalc2->SetInput(reader2->GetOutput());
      lifeCalc2->SetFeatureLifeArray(lifetimeArrayRP);
      lifeCalc2->CalculateFeatureLifetimeOn();
      lifeCalc2->SetFeatureLifetimeOff();
      lifeCalc2->Update();
    }

    // Measuring average feature lifetime
    double lifeSumSH(0), lifeSumRP(0), avgLifeSH, avgLifeRP;
    for(int i = 0 ; i < maxTrackingIdSH+1 ; ++i)
      lifeSumSH += lifetimeArraySH->GetComponent(i,0);
    for(int i = 0 ; i < maxTrackingIdRP+1 ; ++i)
      lifeSumRP += lifetimeArrayRP->GetComponent(i,0);

    avgLifeSH = lifeSumSH / (maxTrackingIdSH+1);
    avgLifeRP = lifeSumRP / (maxTrackingIdRP+1);

    cout << "SH total number of features = " << numLinesSH << endl
         << "SH number of untracked features = " << lifetimeArraySH->GetComponent(0,0) << endl
         << "SH percent untracked = " << lifetimeArraySH->GetComponent(0,0)/numLinesSH << endl
         << "SH average life = " << avgLifeSH << endl
         << "RP total number of features = " << numLinesRP << endl
         << "RP number of untracked features = " << lifetimeArrayRP->GetComponent(0,0) << endl
         << "RP percent untracked = " << lifetimeArrayRP->GetComponent(0,0)/numLinesRP << endl
         << "RP average life = " << avgLifeRP << endl
         << endl;

  //------------------------------------------------------------------------------
  // Setting feature lifetimes

    cout << "*******Setting feature lifetime arrays.************" << endl;
    // Iterating through time steps to set feature lifetimes
    for(int i = 1 ; i < numberOfDataSets-1 ; ++i)
    {

      // Getting file names for input/output files
      inputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      outputFileNameSH = outputFilePrefix + "_" + timeArrayString[i] + "_SH.vtk";
      inputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";
      outputFileNameRP = outputFilePrefix + "_" + timeArrayString[i] + "_RP.vtk";

      ////****Sujudi-Haimes Section****////
      // Reading input file
      vtkSmartPointer<vtkPolyDataReader> reader3 =
          vtkSmartPointer<vtkPolyDataReader>::New();
      reader3->SetFileName(inputFileNameSH.c_str());
      reader3->Update();

      // Passing data set to calculate feature lifetime
      vtkSmartPointer<vtkFeatureLifetime> lifeCalc3 =
          vtkSmartPointer<vtkFeatureLifetime>::New();
      lifeCalc3->SetInput(reader3->GetOutput());
```

```
    lifeCalc3->SetFeatureLifeArray(lifetimeArraySH);
    lifeCalc3->CalculateFeatureLifetimeOff();
    lifeCalc3->SetFeatureLifetimeOn();
    lifeCalc3->Update();

    // Writing output file
    vtkSmartPointer<vtkPolyDataWriter> writer1 =
        vtkSmartPointer<vtkPolyDataWriter>::New();
    writer1->SetInput(lifeCalc3->GetOutput());
    writer1->SetFileName(outputFileNameSH.c_str());
    writer1->Write();

    ////****Roth-Peikert Section****////
    // Reading input file
    vtkSmartPointer<vtkPolyDataReader> reader4 =
        vtkSmartPointer<vtkPolyDataReader>::New();
    reader4->SetFileName(inputFileNameRP.c_str());
    reader4->Update();

    // Passing data set to calculate feature lifetime
    vtkSmartPointer<vtkFeatureLifetime> lifeCalc4 =
        vtkSmartPointer<vtkFeatureLifetime>::New();
    lifeCalc4->SetInput(reader4->GetOutput());
    lifeCalc4->SetFeatureLifeArray(lifetimeArrayRP);
    lifeCalc4->CalculateFeatureLifetimeOff();
    lifeCalc4->SetFeatureLifetimeOn();
    lifeCalc4->Update();

    // Writing output file
    vtkSmartPointer<vtkPolyDataWriter> writer2 =
        vtkSmartPointer<vtkPolyDataWriter>::New();
    writer2->SetInput(lifeCalc4->GetOutput());
    writer2->SetFileName(outputFileNameRP.c_str());
    writer2->Write();
  }

  // Deleting lifetime arrays
  lifetimeArraySH->Delete();
  lifetimeArrayRP->Delete();
}


/**********************PERFORMING SUBJECTIVE LOGIC**********************/
if(logic)
{
  // Subjective logic cannot be computed until 3rd time step because of
  // feature tracking and time derivatives
  for(int i = 3; i < numberOfDataSets-1; ++i)
  {
    //Stop watch -- vortex
    CStopWatch stopWatchVortex = CStopWatch::CStopWatch();
    stopWatchVortex.startTimer();
    double timeToCompletionVortex, oldTimeVortex;

    cout << "Vortex: Computing Opinion for " << i << endl;
    stopWatchVortex.startTimer();

    // getting correct names for the files
    string activeFileNameSH, passiveFileNameSH, outputFileNameSH,
           activeFileNameRP, passiveFileNameRP, outputFileNameRP,
           outputFileNameVortex;

    activeFileNameSH  = outputFilePrefix + "_" + timeArrayString[i]   + "_SH.vtk";
    passiveFileNameSH = outputFilePrefix + "_" + timeArrayString[i-1] + "_Complete_SH.vtk";
    outputFileNameSH  = outputFilePrefix + "_" + timeArrayString[i]   + "_Complete_SH.vtk";

    activeFileNameRP  = outputFilePrefix + "_" + timeArrayString[i]   + "_RP.vtk";
    passiveFileNameRP = outputFilePrefix + "_" + timeArrayString[i-1] + "_Complete_RP.vtk";
```

```
        outputFileNameRP   = outputFilePrefix + "_" + timeArrayString[i]   + "_Complete_RP.vtk";

        outputFileNameVortex   = outputFilePrefix + "_" + timeArrayString[i] + "_Complete_Vortex.vtk
            ";

        ////****Sujudi-Haimes Section****////
        cout << "\tSujudi-Haimes\n";
        // Reading in Sujudi-Haimes vortex core lines
        vtkSmartPointer<vtkPolyDataReader> polyReader1 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader1->SetFileName(activeFileNameSH.c_str());
        polyReader1->Update();

        ////****Roth-Peikert Section****////
        cout << "\tRoth-Peikert\n";
        // Reading in Roth-Peikert vortex core lines
        vtkSmartPointer<vtkPolyDataReader> polyReader2 =
            vtkSmartPointer<vtkPolyDataReader>::New();
        polyReader2->SetFileName(activeFileNameRP.c_str());
        polyReader2->Update();

        // Computing minimum distance between points in Sujudi-Haimes data set
        // and points in Roth-Peikert data set.
        vtkSmartPointer<vtkMinimumDistance> minimumDistance1 =
            vtkSmartPointer<vtkMinimumDistance>::New();
        minimumDistance1->AddInputConnection(polyReader1->GetOutputPort());
        minimumDistance1->AddInputConnection(polyReader2->GetOutputPort());
        minimumDistance1->Update();

        // Computing minimum distance between points in Roth-Peikert data set
        // and points in Sujudi-Haimes data set.
        vtkSmartPointer<vtkMinimumDistance> minimumDistance2 =
            vtkSmartPointer<vtkMinimumDistance>::New();
        minimumDistance2->AddInputConnection(polyReader2->GetOutputPort());
        minimumDistance2->AddInputConnection(polyReader1->GetOutputPort());
        minimumDistance2->Update();

        // Creating the final opinion of the data set
        vtkSmartPointer<vtkCreateOpinion_Vortex> createOpinion1 =
            vtkSmartPointer<vtkCreateOpinion_Vortex>::New();
        createOpinion1->SetInput(minimumDistance1->GetOutput());
        createOpinion1->SujudiHaimesOn();
        createOpinion1->RothPeikertOff();
        createOpinion1->SetTransient(true);
        createOpinion1->SetFeatureLifeNorm(normFeatureLife);
        createOpinion1->Update();

        // Removing unneeded arrays
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("a");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("b");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("c");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("d");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("e");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("f");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("l");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("t");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("Discriminant");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("Gradients");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("TensorXVelocity");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("Time1stDerivatives");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("Time2ndDerivatives");
        createOpinion1->GetOutput()->GetPointData()->RemoveArray("VelocityMagnitude");

        // Creating the final opinion of the data set
        vtkSmartPointer<vtkCreateOpinion_Vortex> createOpinion2 =
            vtkSmartPointer<vtkCreateOpinion_Vortex>::New();
        createOpinion2->SetInput(minimumDistance2->GetOutput());
        createOpinion2->SujudiHaimesOff();
```

```
      createOpinion2->RothPeikertOn();
      createOpinion2->SetTransient(true);
      createOpinion2->SetFeatureLifeNorm(normFeatureLife);
      createOpinion2->Update();

      // Removing unneeded arrays
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("a");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("b");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("c");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("d");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("e");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("f");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("l");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("t");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("Gradients");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("Gradients1");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("TensorXVelocity");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("TensorXVelocity1");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("Time1stDerivatives");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("Time2ndDerivatives");
      createOpinion2->GetOutput()->GetPointData()->RemoveArray("VelocityMagnitude");

      // Combining believable vortex outputs
      vtkSmartPointer<vtkCombineFeatureSets> combineVortex =
          vtkSmartPointer<vtkCombineFeatureSets>::New();
      combineVortex->AddInputConnection(createOpinion1->GetOutputPort());
      combineVortex->AddInputConnection(createOpinion2->GetOutputPort());
      combineVortex->PointFeaturesOff();
      combineVortex->LineFeaturesOn();
      combineVortex->SetProbabilityExpectationThreshold(probExpThreshold);
      combineVortex->SetLengthTolerance(combLengthTol);
      combineVortex->SetDistanceTolerance(combDistTol);
      combineVortex->Update();

      // Writing file to check it
      vtkSmartPointer<vtkPolyDataWriter> pdWriter1 =
          vtkSmartPointer<vtkPolyDataWriter>::New();
      pdWriter1->SetInput(createOpinion1->GetOutput());
      pdWriter1->SetFileName(outputFileNameSH.c_str());
      pdWriter1->Write();

      // Writing file to check it
      vtkSmartPointer<vtkPolyDataWriter> pdWriter2 =
          vtkSmartPointer<vtkPolyDataWriter>::New();
      pdWriter2->SetInput(createOpinion2->GetOutput());
      pdWriter2->SetFileName(outputFileNameRP.c_str());
      pdWriter2->Write();

      // Writing file to check it
      vtkSmartPointer<vtkPolyDataWriter> pdWriter3 =
          vtkSmartPointer<vtkPolyDataWriter>::New();
      pdWriter3->SetInput(combineVortex->GetOutput());
      pdWriter3->SetFileName(outputFileNameVortex.c_str());
      pdWriter3->Write();

      stopWatchVortex.stopTimer();
      oldTimeVortex = stopWatchVortex.getElapsedTime();
      cout << "Vortex: Completed " << i << " in Time = " << oldTimeVortex << " s" << endl << endl
          ;
    }
  }


  return 1;
}
```

B.3 Header Files

In this section header files are listed for code I have written in C++. Header files have not been listed for code I did not write like vtkRothPeikert and vtkSujudiHaimes. All of the code uses VTK 5.8 code as superclasses. Two books from Kitware, Inc. explain the VTK object structure [76, 77]. The header files are listed in alphabetical order.

B.3.1 vtkAttributeTracking.h

```
// .NAME vtkAttributeTracking

// .SECTION Description
// vtkAttributeTracking is a filter that tracks features
// through time based on the feature's attributes.

#ifndef __vtkAttributeTracking_h
#define __vtkAttributeTracking_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkAttributeTracking : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkAttributeTracking,vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkAttributeTracking *New();

    // Turn on/off boundary data set calculations
    vtkSetMacro(BoundaryDataSet,     int);
    vtkGetMacro(BoundaryDataSet,     int);
    vtkBooleanMacro(BoundaryDataSet, int);  //false is 0

    // Turn on/off forward pass
    vtkSetMacro(ForwardPass,     int);
    vtkGetMacro(ForwardPass,     int);
    vtkBooleanMacro(ForwardPass, int);  //false is 0

    // Turn on/off backward pass
    vtkSetMacro(BackwardPass,     int);
    vtkGetMacro(BackwardPass,     int);
    vtkBooleanMacro(BackwardPass, int);  //false is 0

    // Setting length tolerance
    vtkSetMacro(MaximumTrackingID, int);
    vtkGetMacro(MaximumTrackingID, int);

    // Setting length tolerance
    vtkSetMacro(LengthTolerance, double);
    vtkGetMacro(LengthTolerance, double);

    // Setting vortex strength tolerance
    vtkSetMacro(StrengthTolerance, double);
    vtkGetMacro(StrengthTolerance, double);

```

```
    // Setting curvature tolerance
    vtkSetMacro(CurvatureTolerance, double);
    vtkGetMacro(CurvatureTolerance, double);

    // Setting quality tolerance
    vtkSetMacro(QualityTolerance, double);
    vtkGetMacro(QualityTolerance, double);

    // Setting distance tolerance
    vtkSetMacro(DistanceTolerance, double);
    vtkGetMacro(DistanceTolerance, double);

    // Setting length weight
    vtkSetMacro(LengthWeight, double);
    vtkGetMacro(LengthWeight, double);

    // Setting vortex strength weight
    vtkSetMacro(StrengthWeight, double);
    vtkGetMacro(StrengthWeight, double);

    // Setting curvature weight
    vtkSetMacro(CurvatureWeight, double);
    vtkGetMacro(CurvatureWeight, double);

    // Setting quality weight
    vtkSetMacro(QualityWeight, double);
    vtkGetMacro(QualityWeight, double);

    // Setting distance weight
    vtkSetMacro(DistanceWeight, double);
    vtkGetMacro(DistanceWeight, double);

  protected:
    vtkAttributeTracking();
    ~vtkAttributeTracking() {};

    // Usual data generation method
    int FillInputPortInformation( int port, vtkInformation* info );
    int RequestData( vtkInformation *, vtkInformationVector **, vtkInformationVector * );

    int BoundaryDataSet;
    int ForwardPass;
    int BackwardPass;
    int MaximumTrackingID;
    double LengthTolerance;
    double StrengthTolerance;
    double CurvatureTolerance;
    double QualityTolerance;
    double DistanceTolerance;
    double LengthWeight;
    double StrengthWeight;
    double CurvatureWeight;
    double QualityWeight;
    double DistanceWeight;

  private:
    vtkAttributeTracking(const vtkAttributeTracking&);     // Not implemented.
    void operator=(const vtkAttributeTracking&);    // Not implemented.
};

#endif
```

B.3.2 vtkCombineFeatureSets.h

```
// .NAME vtkCombineFeatureSets - computes feature displacement
```

```
// .SECTION Description
// vtkCombineFeatureSets is a filter that takes two feature data sets
// as input and outputs one feature set. The two data sets are combined
// and thresholded by probability expectation.

#ifndef __vtkCombineFeatureSets_h
#define __vtkCombineFeatureSets_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkCombineFeatureSets : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkCombineFeatureSets,vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkCombineFeatureSets *New();

    // turning on/off line feature methods
    vtkSetMacro(LineFeatures,     int);
    vtkGetMacro(LineFeatures,     int);
    vtkBooleanMacro(LineFeatures, int);

    // turning on/off point feature methods
    vtkSetMacro(PointFeatures,     int);
    vtkGetMacro(PointFeatures,     int);
    vtkBooleanMacro(PointFeatures, int);

    // setting probability expectation threshold
    vtkSetMacro(ProbabilityExpectationThreshold, double);
    vtkGetMacro(ProbabilityExpectationThreshold, double);

    // setting length tolerance
    vtkSetMacro(LengthTolerance, double);
    vtkGetMacro(LengthTolerance, double);

    // setting distance tolerance
    vtkSetMacro(DistanceTolerance, double);
    vtkGetMacro(DistanceTolerance, double);

  protected:
    vtkCombineFeatureSets();
    ~vtkCombineFeatureSets() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
    int FillInputPortInformation( int port, vtkInformation* info);

    vtkIntArray *SameLineArray;
    int LineFeatures;
    int PointFeatures;
    double ProbabilityExpectationThreshold;
    double LengthTolerance;
    double DistanceTolerance;

  private:
    vtkCombineFeatureSets(const vtkCombineFeatureSets&);     // Not implemented.
    void operator=(const vtkCombineFeatureSets&);   // Not implemented.
};

#endif
```

B.3.3 vtkCreateOpinion_Vortex.h

```
// .NAME vtkCreateOpinion_Vortex

// .SECTION Description
// vtkCreateOpinion_Vortex is a filter that computes the opinion
// of each extracted point.

#ifndef __vtkCreateOpinion_Vortex_h
#define __vtkCreateOpinion_Vortex_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkCreateOpinion_Vortex : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkCreateOpinion_Vortex,vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkCreateOpinion_Vortex *New();

    // Description: Set/Get constant used to find belief,
    // disbelief, and uncertainty values for Master Agent.
    vtkSetMacro(FeatureDisplacementConstant, double);
    vtkGetMacro(FeatureDisplacementConstant, double);

    // Description: Set/Get constant used to find belief,
    // disbelief, and uncertainty values for Master Agent.
    vtkSetMacro(ChangeInFeatureDisplacementConstant, double);
    vtkGetMacro(ChangeInFeatureDisplacementConstant, double);

    // Description: Set/Get feature life normalization value
    // which divides all the feature life values.
    vtkSetMacro(FeatureLifeNorm, int);
    vtkGetMacro(FeatureLifeNorm, int);

    // Description: Turn on/off Sujudi-Haimes as the
    // active extraction algorithm.
    vtkSetMacro(SujudiHaimes,     int);
    vtkGetMacro(SujudiHaimes,     int);
    vtkBooleanMacro(SujudiHaimes, int);

    // Description: Turn on/off Roth-Peikert as the
    // active extraction algorithm.
    vtkSetMacro(RothPeikert,     int);
    vtkGetMacro(RothPeikert,     int);
    vtkBooleanMacro(RothPeikert, int);

    // setting transient/steady-state
    vtkSetMacro(Transient,     int);
    vtkGetMacro(Transient,     int);
    vtkBooleanMacro(Transient, int);

    // Description: Set/Get largest vortex strength value which
    // divides all the vortex strength values.
    vtkSetMacro(VortexStrengthNorm, double);
    vtkGetMacro(VortexStrengthNorm, double);

    // Description: Set/Get largest curvature value which
    // divides all the curvature values.
    vtkSetMacro(CurvatureNorm, double);
    vtkGetMacro(CurvatureNorm, double);

```

```
    // Description: Set/Get largest quality value which
    // divides all the quality values.
    vtkSetMacro(QualityNorm, double);
    vtkGetMacro(QualityNorm, double);

    // Description: Set/Get largest quality value which
    // divides all the minimumDistance values.
    vtkSetMacro(MinimumDistanceNorm, double);
    vtkGetMacro(MinimumDistanceNorm, double);

    // Description: Set/Get largest quality value which
    // divides all the RP windingAngle values.
    vtkSetMacro(Lambda2Norm, double);
    vtkGetMacro(Lambda2Norm, double);

  protected:
    vtkCreateOpinion_Vortex();
    ~vtkCreateOpinion_Vortex() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

    double FeatureDisplacementConstant;
    double ChangeInFeatureDisplacementConstant;
    int FeatureLifeNorm;
    double VortexStrengthNorm;
    double CurvatureNorm;
    double QualityNorm;
    double MinimumDistanceNorm;
    double Lambda2Norm;
    int    SujudiHaimes;
    int    RothPeikert;
    int    Transient;

  private:
    vtkCreateOpinion_Vortex(const vtkCreateOpinion_Vortex&);    // Not implemented.
    void operator=(const vtkCreateOpinion_Vortex&);  // Not implemented.
};

#endif
```

B.3.4 vtkCurvature.h

```
// .NAME vtkCurvature - computes curvature of lines

// .SECTION Description
// vtkCurvature is a filter that computes the curvature of a polyline and
// sets a curvature value for each point in the line.

#ifndef __vtkCurvature_h
#define __vtkCurvature_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;
class vtkPointLocator;

class VTK_GRAPHICS_EXPORT vtkCurvature : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkCurvature,vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

```

```
    static vtkCurvature *New();

    // Description:

    // Description:
    // Turn on/off the calculation of curvature for multiple
    // line segments using a circle approximation.
    vtkSetMacro(MultiSegmentCurvature,      int);
    vtkGetMacro(MultiSegmentCurvature,      int);
    vtkBooleanMacro(MultiSegmentCurvature, int);  //false is 0

    // Description:
    // Turn on/off the calculation of curvature using only
    // the velocity field and not the geometry.
    vtkSetMacro(VelocityFieldCurvature,      int);
    vtkGetMacro(VelocityFieldCurvature,      int);
    vtkBooleanMacro(VelocityFieldCurvature, int);  //false is 0

    // Description:
    // Turn on/off the calculation of curvature using only
    // points and an octree point locator
    vtkSetMacro(PointwiseCurvature,      int);
    vtkGetMacro(PointwiseCurvature,      int);
    vtkBooleanMacro(PointwiseCurvature, int);  //false is 0

  protected:
    vtkCurvature();
    ~vtkCurvature() {};

    int MultiSegmentCurvature;
    int VelocityFieldCurvature;
    int PointwiseCurvature;

    // Usual data generation method
    virtual int FillInputPortInformation(int port, vtkInformation *info);
    virtual int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

  private:
    vtkCurvature(const vtkCurvature&);      // Not implemented.
    void operator=(const vtkCurvature&);    // Not implemented.
};

#endif
```

B.3.5 vtkFeatureAttributes.h

```
// .NAME vtkFeatureAttributes

// .SECTION Description
// vtkFeatureAttributes is a filter that calculates
// line attributes for use in feature tracking.

#ifndef __vtkFeatureAttributes_h
#define __vtkFeatureAttributes_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkFeatureAttributes : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkFeatureAttributes,vtkPolyDataAlgorithm);
```

```
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkFeatureAttributes *New();

  protected:
    vtkFeatureAttributes();
    ~vtkFeatureAttributes() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

  private:
    vtkFeatureAttributes(const vtkFeatureAttributes&);    // Not implemented.
    void operator=(const vtkFeatureAttributes&);   // Not implemented.
};

#endif
```

B.3.6 vtkFeatureLifetime.h

```
// .NAME vtkFeatureLifetime

// .SECTION Description
// vtkFeatureLifetime is a filter that calculates
// line attributes for use in feature tracking.

#ifndef __vtkFeatureLifetime_h
#define __vtkFeatureLifetime_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkFeatureLifetime : public vtkPolyDataAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkFeatureLifetime,vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkFeatureLifetime *New();

    // Turn on/off calculation of feature lifetimes
    vtkSetMacro(CalculateFeatureLifetime,     int);
    vtkGetMacro(CalculateFeatureLifetime,     int);
    vtkBooleanMacro(CalculateFeatureLifetime, int);  //false is 0

    // Turn on/off setting of feature lifetimes
    vtkSetMacro(SetFeatureLifetime,     int);
    vtkGetMacro(SetFeatureLifetime,     int);
    vtkBooleanMacro(SetFeatureLifetime, int);  //false is 0

    // Set/Get feature lifetime array
    vtkGetMacro(FeatureLifeArray, vtkIntArray*);
    vtkSetMacro(FeatureLifeArray, vtkIntArray*);

  protected:
    vtkFeatureLifetime();
    ~vtkFeatureLifetime() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

    int CalculateFeatureLifetime;
```

```
    int SetFeatureLifetime;
    vtkIntArray *FeatureLifeArray;

  private:
    vtkFeatureLifetime(const vtkFeatureLifetime&);      // Not implemented.
    void operator=(const vtkFeatureLifetime&);    // Not implemented.
};

#endif
```

B.3.7 vtkLambdaTwo.h

```
// .NAME vtkLambdaTwo

// .SECTION Description
// vtkLambdaTwo is a filter that computes the partial derivatives
// with respect to time in a data set.

#ifndef __vtkLambdaTwo_h
#define __vtkLambdaTwo_h

#include "vtkDataSetAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkLambdaTwo : public vtkDataSetAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkLambdaTwo, vtkDataSetAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkLambdaTwo *New();

    // creating the velocity array name
    vtkSetMacro(VelocityArrayName, const char *);
    vtkGetMacro(VelocityArrayName, const char *);

  protected:
    vtkLambdaTwo();
    ~vtkLambdaTwo() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
    int FillInputPortInformation( int port, vtkInformation* info);

    const char * VelocityArrayName;

  private:
    vtkLambdaTwo(const vtkLambdaTwo&);      // Not implemented.
    void operator=(const vtkLambdaTwo&);    // Not implemented.
};

#endif
```

B.3.8 vtkTimeDerivatives.h

```
// .NAME vtkTimeDerivatives

// .SECTION Description
// vtkTimeDerivatives is a filter that computes the partial derivatives
// with respect to time in a data set.
```

```
#ifndef __vtkTimeDerivatives_h
#define __vtkTimeDerivatives_h

#include "vtkDataSetAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkTimeDerivatives : public vtkDataSetAlgorithm
{
  public:
    vtkTypeRevisionMacro(vtkTimeDerivatives,vtkDataSetAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkTimeDerivatives *New();

    // Description: Set/Get time step for use in
    // calculating time derivatives.
    vtkSetMacro(TimeStep, double);
    vtkGetMacro(TimeStep, double);

    // creating the velocity array name
    vtkSetMacro(Velocity1ArrayName, const char *);
    vtkGetMacro(Velocity1ArrayName, const char *);

    // creating the velocity array name
    vtkSetMacro(Velocity2ArrayName, const char *);
    vtkGetMacro(Velocity2ArrayName, const char *);

    // creating the velocity array name
    vtkSetMacro(Velocity3ArrayName, const char *);
    vtkGetMacro(Velocity3ArrayName, const char *);

    // Description:
    // Turn on/off the calculation of forward-differenced derivatives
    vtkSetMacro(ForwardDifference,     int);
    vtkGetMacro(ForwardDifference,     int);
    vtkBooleanMacro(ForwardDifference, int);  //false is 0

    // Description:
    // Turn on/off the calculation of backward-differenced derivatives
    vtkSetMacro(BackwardDifference,     int);
    vtkGetMacro(BackwardDifference,     int);
    vtkBooleanMacro(BackwardDifference, int);  //false is 0

    // Description:
    // Turn on/off the calculation of central-differenced derivatives
    vtkSetMacro(CentralDifference,     int);
    vtkGetMacro(CentralDifference,     int);
    vtkBooleanMacro(CentralDifference, int);  //false is 0

  protected:
    vtkTimeDerivatives();
    ~vtkTimeDerivatives() {};

    // Usual data generation method
    int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
    int FillInputPortInformation( int port, vtkInformation* info);

    double TimeStep;
    const char * Velocity1ArrayName;
    const char * Velocity2ArrayName;
    const char * Velocity3ArrayName;
    int ForwardDifference;
    int BackwardDifference;
    int CentralDifference;
```

```

private:
  vtkTimeDerivatives(const vtkTimeDerivatives&);       // Not implemented.
  void operator=(const vtkTimeDerivatives&);     // Not implemented.
};

#endif
```

B.4 Source Files

In this section source files are listed for each of the header files in Section B.3. Source files have not been listed for code I did not write like vtkRothPeikert and vtkSujudiHaimes. All of the code uses VTK 5.8 code as superclasses. The source files are listed in alphabetical order.

B.4.1 vtkAttributeTracking.cxx

```
#include "vtkAttributeTracking.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkAttributeTracking, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkAttributeTracking);

//----------------------------------------------------------------------------
vtkAttributeTracking::vtkAttributeTracking()
{
  this->SetNumberOfInputPorts(1);
  this->SetNumberOfOutputPorts(2);
  this->BoundaryDataSet = false;
  this->ForwardPass = true;
  this->BackwardPass = false;
  this->MaximumTrackingID = 0;
  this->LengthTolerance = 0.1;
  this->StrengthTolerance = 0.1;
  this->CurvatureTolerance = 0.1;
  this->QualityTolerance = 0.1;
  this->DistanceTolerance = 0.1;
  this->LengthWeight = 0.25;
  this->StrengthWeight = 0.20;
  this->CurvatureWeight = 0.15;
  this->QualityWeight = 0.15;
  this->DistanceWeight = 0.25;
}

//----------------------------------------------------------------------------
int vtkAttributeTracking::FillInputPortInformation( int port, vtkInformation* info )
{
  if ( port == 0 )
  {
    info->Set(vtkDataObject::DATA_TYPE_NAME(), "vtkPolyData" );
    info->Set(vtkAlgorithm::INPUT_IS_REPEATABLE(), 1);

    return 1;
  }

  vtkErrorMacro("This filter does not have more than 1 input port!");
  return 0;
}

```

```
//----------------------------------------------------------------------------
int vtkAttributeTracking::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo1  = inputVector[0]->GetInformationObject(0);
  vtkInformation *inInfo2  = inputVector[0]->GetInformationObject(1);
  vtkInformation *inInfo3  = inputVector[0]->GetInformationObject(2);
  vtkInformation *outInfo1 = outputVector->GetInformationObject(0);
  vtkInformation *outInfo2 = outputVector->GetInformationObject(1);

  // get input and output
  vtkPolyData *input1  = vtkPolyData::SafeDownCast(inInfo1->Get(vtkDataObject::DATA_OBJECT())); // current time step
  vtkPolyData *input2  = vtkPolyData::SafeDownCast(inInfo2->Get(vtkDataObject::DATA_OBJECT())); // next time step
  vtkPolyData *input3  = vtkPolyData::SafeDownCast(inInfo3->Get(vtkDataObject::DATA_OBJECT())); // previous time step
  vtkPolyData *output1 = vtkPolyData::SafeDownCast(outInfo1->Get(vtkDataObject::DATA_OBJECT())); // current
  vtkPolyData *output2 = vtkPolyData::SafeDownCast(outInfo2->Get(vtkDataObject::DATA_OBJECT())); // next

  // Creating correspondence array
  vtkSmartPointer<vtkDoubleArray> correspondenceArray = vtkSmartPointer<vtkDoubleArray>::New();
  correspondenceArray->SetNumberOfValues(2*input1->GetNumberOfLines());
  correspondenceArray->SetNumberOfComponents(2);
  correspondenceArray->SetNumberOfTuples(input1->GetNumberOfLines());
  correspondenceArray->SetName("LineCorrespondence");

  // Creating new tracking ID array - current time step
  vtkSmartPointer<vtkIntArray> IDArray = vtkSmartPointer<vtkIntArray>::New();
  IDArray->SetNumberOfValues(input1->GetNumberOfLines());
  IDArray->SetNumberOfComponents(1);
  IDArray->SetNumberOfTuples(input1->GetNumberOfLines());
  IDArray->SetName("TrackingID");

  // Creating correspondence array - next time step
  vtkSmartPointer<vtkDoubleArray> correspondenceArrayNext = vtkSmartPointer<vtkDoubleArray>::New();
  correspondenceArrayNext->SetNumberOfValues(2*input2->GetNumberOfLines());
  correspondenceArrayNext->SetNumberOfComponents(2);
  correspondenceArrayNext->SetNumberOfTuples(input2->GetNumberOfLines());
  correspondenceArrayNext->SetName("LineCorrespondence");

  // Creating new tracking ID array - next time step
  vtkSmartPointer<vtkIntArray> IDArrayNext = vtkSmartPointer<vtkIntArray>::New();
  IDArrayNext->SetNumberOfValues(input2->GetNumberOfLines());
  IDArrayNext->SetNumberOfComponents(1);
  IDArrayNext->SetNumberOfTuples(input2->GetNumberOfLines());
  IDArrayNext->SetName("TrackingID");

  // Copying old correspondence & tracking ID arrays into new ones
  for(int i = 0 ; i < input1->GetNumberOfLines() ; i++)
  {
    correspondenceArray->SetComponent(i,0,input1->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,0));
    correspondenceArray->SetComponent(i,1,input1->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,1));
    IDArray->SetComponent(i,0,input1->GetCellData()->GetArray("TrackingID")->GetComponent(i,0));
  }
  for(int i = 0 ; i < input2->GetNumberOfLines() ; i++)
  {
    correspondenceArrayNext->SetComponent(i,0,input2->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,0));
```

```
  correspondenceArrayNext->SetComponent(i,1,input2->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,1));
  IDArrayNext->SetComponent(i,0,input2->GetCellData()->GetArray("TrackingID")->GetComponent(i,0));
}

// Removing old ID arrays from the input data sets
input1->GetCellData()->RemoveArray("LineCorrespondence");
input1->GetCellData()->RemoveArray("TrackingID");
input2->GetCellData()->RemoveArray("LineCorrespondence");
input2->GetCellData()->RemoveArray("TrackingID");

// Iterating through lines in current time step
for(int i = 0 ; i < input1->GetNumberOfLines() ; i++)
{
  // Instantiating variables
  int trackingID, trackingIDNext;
  double length, lengthNext;
  double strength, strengthNext;
  double curvature, curvatureNext;
  double quality, qualityNext;
  double bounds[6], boundsNext[6];
  double xC, yC, zC, xCNext, yCNext, zCNext;
  double fL, fS, fC, fQ, fD, corr;

  // Getting tracking ID of current line
  trackingID = int(IDArray->GetComponent(i,0));

  // Setting current attributes
  // Previously untracked line, i.e. trackingID = 0
  if(trackingID == 0)
  {
    length    = input1->GetCellData()->GetArray("LineLength")->GetComponent(i,0);
    strength  = input1->GetCellData()->GetArray("LineVortexStrength")->GetComponent(i,0);
    curvature = input1->GetCellData()->GetArray("LineCurvature")->GetComponent(i,0);
    quality   = input1->GetCellData()->GetArray("LineQuality")->GetComponent(i,0);

    // Getting bounds of line and finding center of bounding box
    input1->GetCellBounds(i,bounds);
    xC = (bounds[0]+bounds[1]) / 2;
    yC = (bounds[2]+bounds[3]) / 2;
    zC = (bounds[4]+bounds[5]) / 2;
  }

  // Previously tracked line, i.e. trackingID != 0
  else
  {
    bool extrapolate = false;
    // Finding corresponding line in prior time step
    int cellPrevID;
    if(!BoundaryDataSet)
    {
      for(int j = 0 ; j < input3->GetNumberOfLines() ; j++)
      {
        // Getting tracking ID of previous line
        int trackingIDPrev = int(input3->GetCellData()->GetArray("TrackingID")->GetComponent(j,0));
        if(trackingIDPrev == trackingID)
        {
          extrapolate = true;
          cellPrevID = j;
          break;
        }
      }
    }


    if(extrapolate)
```

```
      {
        // Use linear extrapolation to predict future attributes
        length    = 2*input1->GetCellData()->GetArray("LineLength")->GetComponent(i,0) -
                    input3->GetCellData()->GetArray("LineLength")->GetComponent(cellPrevID,0);
        strength  = 2*input1->GetCellData()->GetArray("LineVortexStrength")->GetComponent(i,0) -
                    input3->GetCellData()->GetArray("LineVortexStrength")->GetComponent(
                        cellPrevID,0);
        curvature = 2*input1->GetCellData()->GetArray("LineCurvature")->GetComponent(i,0) -
                    input3->GetCellData()->GetArray("LineCurvature")->GetComponent(cellPrevID,0);
        quality   = 2*input1->GetCellData()->GetArray("LineQuality")->GetComponent(i,0) -
                    input3->GetCellData()->GetArray("LineQuality")->GetComponent(cellPrevID,0);

        // Getting bounds of line and finding center of bounding box
        double boundsPrev[6];
        input1->GetCellBounds(i,bounds);
        input3->GetCellBounds(cellPrevID,boundsPrev);
        xC = bounds[0] + bounds[1] - (boundsPrev[0]+boundsPrev[1]) / 2;
        yC = bounds[2] + bounds[3] - (boundsPrev[2]+boundsPrev[3]) / 2;
        zC = bounds[4] + bounds[5] - (boundsPrev[4]+boundsPrev[5]) / 2;
      }
      else
      {
        length    = input1->GetCellData()->GetArray("LineLength")->GetComponent(i,0);
        strength  = input1->GetCellData()->GetArray("LineVortexStrength")->GetComponent(i,0);
        curvature = input1->GetCellData()->GetArray("LineCurvature")->GetComponent(i,0);
        quality   = input1->GetCellData()->GetArray("LineQuality")->GetComponent(i,0);

        // Getting bounds of line and finding center of bounding box
        input1->GetCellBounds(i,bounds);
        xC = (bounds[0]+bounds[1]) / 2;
        yC = (bounds[2]+bounds[3]) / 2;
        zC = (bounds[4]+bounds[5]) / 2;
      }
    }

    // check to make sure that line has not been tracked already into the future
    bool alreadyTracked = false;
    for(int j = 0 ; j < input2->GetNumberOfLines() ; j++)
    {
      trackingIDNext = int(IDArrayNext->GetComponent(j,0));
      if(trackingID != 0 && trackingIDNext == trackingID)
      {
        alreadyTracked = true;
        break;
      }

    }

    // Tracking only if line has not already been tracked in the future
    if(!alreadyTracked)
    {
      // Creating cell ID array for later use
      vtkSmartPointer<vtkIdFilter> ids = vtkSmartPointer<vtkIdFilter>::New();
      ids->SetInput(input2);
      ids->PointIdsOff();
      ids->CellIdsOn();
      ids->FieldDataOn();
      ids->SetIdsArrayName("CellID");

      // Creating a sphere source for finding nearby core lines
      vtkSmartPointer<vtkSphere> sphere = vtkSmartPointer<vtkSphere>::New();
      sphere->SetRadius(length);
      sphere->SetCenter(xC,yC,zC);

      // Extracting lines in data set within bounding sphere
      vtkSmartPointer<vtkExtractPolyDataGeometry> extract = vtkSmartPointer<
          vtkExtractPolyDataGeometry>::New();
      extract->SetInput(ids->GetOutput());
```

```
extract->SetImplicitFunction(sphere);
extract->ExtractInsideOn();
extract->ExtractBoundaryCellsOn();
extract->Update();

// Getting cell ID's of extracted lines
vtkSmartPointer<vtkIdList> cellIds = vtkSmartPointer<vtkIdList>::New();
for(int j(0) ; j < extract->GetOutput()->GetNumberOfLines() ; ++j)
  cellIds->InsertId(j, extract->GetOutput()->GetCellData()->GetArray("CellID")->GetComponent(j,0));

// Compare line to all others in next time step
// Ignore lines already tracked (trackingID != 0)
double corrMax = -100;
int corrMaxLine(0);
for(int j = 0 ; j < extract->GetOutput()->GetNumberOfLines() ; j++)
{
  // Ignoring lines in next time step which have been tracked
  trackingIDNext = int(IDArrayNext->GetComponent(cellIds->GetId(j),0));
  if(trackingIDNext == 0)
  {
    // Getting attributes of line in next time step
    lengthNext    = input2->GetCellData()->GetArray("LineLength")->GetComponent(cellIds->GetId(j),0);
    strengthNext  = input2->GetCellData()->GetArray("LineVortexStrength")->GetComponent(cellIds->GetId(j),0);
    curvatureNext = input2->GetCellData()->GetArray("LineCurvature")->GetComponent(cellIds->GetId(j),0);
    qualityNext   = input2->GetCellData()->GetArray("LineQuality")->GetComponent(cellIds->GetId(j),0);

    // Getting bounds of line and finding center of bounding box
    input2->GetCellBounds(cellIds->GetId(j),boundsNext);
    xCNext = (boundsNext[0]+boundsNext[1]) / 2;
    yCNext = (boundsNext[2]+boundsNext[3]) / 2;
    zCNext = (boundsNext[4]+boundsNext[5]) / 2;

    // Computing correspondence functions
    fL = 1-(fabs(length-lengthNext)/((length > lengthNext) ?
          length : lengthNext))/LengthTolerance;
    fS = 1-(fabs(fabs(strength)-fabs(strengthNext))/((fabs(strength) > fabs(strengthNext)) ?
          fabs(strength) : fabs(strengthNext)))/StrengthTolerance;
    fC = 1-(fabs(curvature-curvatureNext)/((curvature > curvatureNext) ?
          curvature : curvatureNext))/CurvatureTolerance;
    fQ = 1-(fabs(quality-qualityNext)/((quality > qualityNext) ?
          quality : qualityNext))/QualityTolerance;
    fD = 1-sqrt(pow(xC-xCNext,2)+pow(yC-yCNext,2)+pow(zC-zCNext,2))/DistanceTolerance;

    // Compute overall correspondence
    corr = (fL*LengthWeight+fS*StrengthWeight+fC*CurvatureWeight+fQ*QualityWeight+fD*DistanceWeight) /
           (LengthWeight+StrengthWeight+CurvatureWeight+QualityWeight+DistanceWeight);

    // detecting tracking continuation
    if(corr > corrMax)
    {
      corrMaxLine = cellIds->GetId(j);
      corrMax = corr;
    }
  }
}

// Setting tracking ID arrays if line was tracked
if(corrMax > 0)
{
  // Setting array for next time step
  // Setting proper correspondence component depending on pass
```

```
        if(ForwardPass)
          correspondenceArrayNext->SetComponent(corrMaxLine,0,corrMax);
        if(BackwardPass)
          correspondenceArrayNext->SetComponent(corrMaxLine,1,corrMax);

        // Newly tracked path receives a new ID
        if(trackingID == 0)
          IDArrayNext->SetValue(corrMaxLine,MaximumTrackingID+1);

        // Continuing path receives prior tracking ID
        else
          IDArrayNext->SetValue(corrMaxLine,trackingID);

        // Setting array for current time step
        // Setting proper correspondence component depending on pass
        if(ForwardPass)
          correspondenceArray->SetComponent(i,1,corrMax);
        if(BackwardPass)
          correspondenceArray->SetComponent(i,0,corrMax);

        // Newly tracked path receives a new ID
        if(trackingID == 0)
          IDArray->SetValue(i,MaximumTrackingID+1);

        // Incrementing maximum tracking ID if a new path was made
        if(trackingID == 0)
          MaximumTrackingID++;
      }

      // Setting correspondence array for untracked lines
      else
      {
        // Set correspondence array if current correspondence is larger than previous value
        if(ForwardPass)
          if(corrMax > correspondenceArray->GetComponent(i,1))
            correspondenceArray->SetComponent(i,1,corrMax);

        if(BackwardPass)
          if(corrMax > correspondenceArray->GetComponent(i,0))
            correspondenceArray->SetComponent(i,0,corrMax);
      }
    }
  }

  // adding arrays to the input data set
  input1->GetCellData()->AddArray(correspondenceArray);
  input1->GetCellData()->AddArray(IDArray);
  input2->GetCellData()->AddArray(correspondenceArrayNext);
  input2->GetCellData()->AddArray(IDArrayNext);

  // Copying the input data and structure to the outputs
  output1->CopyStructure(input1);
  output1->GetPointData()->PassData(input1->GetPointData());
  output1->GetCellData()->PassData(input1->GetCellData());
  output1->GetFieldData()->PassData(input1->GetFieldData());
  output2->CopyStructure(input2);
  output2->GetPointData()->PassData(input2->GetPointData());
  output2->GetCellData()->PassData(input2->GetCellData());
  output2->GetFieldData()->PassData(input2->GetFieldData());

  return 1;
}

//----------------------------------------------------------------------------
void vtkAttributeTracking::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.2 vtkCombineFeatureSets.cxx

```
#include "vtkCombineFeatureSets.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkCombineFeatureSets, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkCombineFeatureSets);

//----------------------------------------------------------------------------
vtkCombineFeatureSets::vtkCombineFeatureSets()
{
  this->SetNumberOfInputPorts(1);
  this->SetNumberOfOutputPorts(1);
  this->LineFeatures = true;
  this->PointFeatures = false;
  this->ProbabilityExpectationThreshold = 0.8;
  this->LengthTolerance = 0.25;
  this->DistanceTolerance = 0.2;
}

//----------------------------------------------------------------------------
int vtkCombineFeatureSets::FillInputPortInformation( int port, vtkInformation* info )
{
  if ( port == 0 )
  {
    info->Set(vtkDataObject::DATA_TYPE_NAME(), "vtkPolyData" );
    info->Set(vtkAlgorithm::INPUT_IS_REPEATABLE(), 1);

    return 1;
  }

  vtkErrorMacro("This filter does not have more than 1 input port!");
  return 0;
}

//----------------------------------------------------------------------------
int vtkCombineFeatureSets::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo1  = inputVector[0]->GetInformationObject(0);
  vtkInformation *inInfo2 = inputVector[0]->GetInformationObject(1);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);

  // get the 2 inputs and 1 ouptut
  // input1 is the data object that we will be calculating the feature displacement for
  vtkPolyData *input1 = vtkPolyData::SafeDownCast(inInfo1->Get(vtkDataObject::DATA_OBJECT()));
  vtkPolyData *input2 = vtkPolyData::SafeDownCast(inInfo2->Get(vtkDataObject::DATA_OBJECT()));
  vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

  // Handle line features
  if(LineFeatures)
  {
    vtkSmartPointer<vtkDoubleArray> lineProbExpArray1 = vtkSmartPointer<vtkDoubleArray>::New();
    lineProbExpArray1->SetNumberOfComponents(1);
    lineProbExpArray1->SetNumberOfTuples(input1->GetNumberOfLines());
    lineProbExpArray1->SetNumberOfValues(input1->GetNumberOfLines());
    lineProbExpArray1->SetName("LineProbabilityExpectation");

    vtkSmartPointer<vtkDoubleArray> lineProbExpArray2 = vtkSmartPointer<vtkDoubleArray>::New();
    lineProbExpArray2->SetNumberOfComponents(1);
    lineProbExpArray2->SetNumberOfTuples(input2->GetNumberOfLines());
    lineProbExpArray2->SetNumberOfValues(input2->GetNumberOfLines());
    lineProbExpArray2->SetName("LineProbabilityExpectation");
```

```
  // Finding line average probability expectation for 1st data set
  std::vector <int> cellPointList1;
  for(int i(0) ; i < input1->GetNumberOfLines() ; ++i)
  {
    // Putting cell point ids into an array
    vtkIdList *cellPtIds1 = input1->GetCell(i)->GetPointIds();
    cellPointList1.resize(cellPtIds1->GetNumberOfIds());
    for(int j = 0 ; j < cellPtIds1->GetNumberOfIds() ; j++)
    {
      cellPointList1[j] = cellPtIds1->GetId(j);
    }

    double probExpSum1(0), probExpMean1;

    // Summing prob. exp. values in line
    for(int j(0) ; j < input1->GetCell(i)->GetNumberOfPoints() ; j++)
      probExpSum1 += input1->GetPointData()->GetArray("ProbabilityExpectation")->GetComponent(
          cellPointList1[j],0);

    // Find average probability expectation value for line
    probExpMean1 = probExpSum1 / input1->GetCell(i)->GetNumberOfPoints();

    // Setting prob. exp. average for points in line
    lineProbExpArray1->SetValue(i, probExpMean1);
  }

  // Finding line average probability expectation for 2nd data set
  std::vector <int> cellPointList2;
  for(int i(0) ; i < input2->GetNumberOfLines() ; ++i)
  {
    // Putting cell point ids into an array
    vtkIdList *cellPtIds2 = input2->GetCell(i)->GetPointIds();
    cellPointList2.resize(cellPtIds2->GetNumberOfIds());
    for(int j = 0 ; j < cellPtIds2->GetNumberOfIds() ; j++)
    {
      cellPointList2[j] = cellPtIds2->GetId(j);
    }

    double probExpSum2(0), probExpMean2;

    // Summing prob. exp. values in line
    for(int j(0) ; j < input2->GetCell(i)->GetNumberOfPoints() ; j++)
      probExpSum2 += input2->GetPointData()->GetArray("ProbabilityExpectation")->GetComponent(
          cellPointList2[j],0);

    // Find average probability expectation value for line
    probExpMean2 = probExpSum2 / input2->GetCell(i)->GetNumberOfPoints();

    // Setting prob. exp. average for points in line
    lineProbExpArray2->SetValue(i, probExpMean2);
  }

  // Adding arrays to input
  input1->GetCellData()->AddArray(lineProbExpArray1);
  input2->GetCellData()->AddArray(lineProbExpArray2);

  // Thresh input1 by probability expectation
  vtkSmartPointer<vtkThreshold> thresh1 = vtkSmartPointer<vtkThreshold>::New();
  thresh1->SetInput(input1);
  thresh1->ThresholdByUpper(ProbabilityExpectationThreshold);
  thresh1->SetInputArrayToProcess(0,0,0,1,"LineProbabilityExpectation");
  thresh1->Update();

  // Convert threshold1 to polydata
  vtkSmartPointer<vtkGeometryFilter> polyData1 = vtkSmartPointer<vtkGeometryFilter>::New();
  polyData1->SetInput(thresh1->GetOutput());
  polyData1->Update();
```

```
  // Thresh input2 by probability expectation
  vtkSmartPointer<vtkThreshold> thresh2 = vtkSmartPointer<vtkThreshold>::New();
  thresh2->SetInput(input2);
  thresh2->ThresholdByUpper(ProbabilityExpectationThreshold);
  thresh2->SetInputArrayToProcess(0,0,0,1,"LineProbabilityExpectation");
  thresh2->Update();

  // Convert threshold2 to polydata
  vtkSmartPointer<vtkGeometryFilter> polyData2 = vtkSmartPointer<vtkGeometryFilter>::New();
  polyData2->SetInput(thresh2->GetOutput());
  polyData2->Update();

  // Find matching lines in data sets
  int num(0);
  std::vector<int> deletedCells;
  for (int i(0) ; i < polyData1->GetOutput()->GetNumberOfLines() ; ++i)
  {
    // Naming variables
    double length, bounds[6], position[3], fL, fD, corr;
    double corrMin(1);
    bool matched = false;
    int cellMatchID;

    // Getting line length and position
    length = polyData1->GetOutput()->GetCellData()->GetArray("LineLength")->GetComponent(i,0);
    polyData1->GetOutput()->GetCellBounds(i,bounds);
    position[0] = (bounds[0]+bounds[1]) / 2;
    position[1] = (bounds[2]+bounds[3]) / 2;
    position[2] = (bounds[4]+bounds[5]) / 2;

    // Creating cell ID array for later use
    vtkSmartPointer<vtkIdFilter> ids = vtkSmartPointer<vtkIdFilter>::New();
    ids->SetInput(polyData2->GetOutput());
    ids->PointIdsOff();
    ids->CellIdsOn();
    ids->FieldDataOn();
    ids->SetIdsArrayName("CellID");

    // Creating a sphere source for finding nearby core lines
    vtkSmartPointer<vtkSphere> sphere = vtkSmartPointer<vtkSphere>::New();
    sphere->SetRadius(0.5*length);
    sphere->SetCenter(position[0], position[1], position[2]);

    // Extracting lines in data set within bounding sphere
    vtkSmartPointer<vtkExtractPolyDataGeometry> extract = vtkSmartPointer<
        vtkExtractPolyDataGeometry>::New();
    extract->SetInput(ids->GetOutput());
    extract->SetImplicitFunction(sphere);
    extract->ExtractInsideOn();
    extract->ExtractBoundaryCellsOn();
    extract->Update();

    // Getting cell ID's of extracted lines
    vtkSmartPointer<vtkIdList> cellIds = vtkSmartPointer<vtkIdList>::New();
    for(int j(0) ; j < extract->GetOutput()->GetNumberOfLines() ; ++j)
      cellIds->InsertId(j, extract->GetOutput()->GetCellData()->GetArray("CellID")->
          GetComponent(j,0));

    // Comparing current line in input1 to nearby lines in input2
    bool deleted = false;
    for(int j(0) ; j < extract->GetOutput()->GetNumberOfLines() ; ++j)
    {
      // Making sure current line has not been deleted
      for(int k(0) ; k < deletedCells.size() ; ++k)
        if(j == deletedCells[k])
        {
          deleted = true;
```

```
          break;
        }


      if(!deleted)
      {
        // Getting line length and position
        double length2, bounds2[6], position2[3];
        length2 = polyData2->GetOutput()->GetCellData()->GetArray("LineLength")->GetComponent(cellIds->GetId(j),0);
        polyData2->GetOutput()->GetCellBounds(cellIds->GetId(j),bounds2);
        position2[0] = (bounds2[0]+bounds2[1]) / 2;
        position2[1] = (bounds2[2]+bounds2[3]) / 2;
        position2[2] = (bounds2[4]+bounds2[5]) / 2;

        // Computing correspondence functions
        fL = fabs(length-length2)/((length > length2) ? length : length2);
        fD = sqrt(pow(position[0]-position2[0],2)+pow(position[1]-position2[1],2)+
                pow(position[2]-position2[2],2))/length;

        // Setting matched lines
        if(fL < LengthTolerance && fD < DistanceTolerance)
        {
          matched = true;
          corr = (fL+fD)/2;

          // Ensuring the best match is made
          if(corr < corrMin)
          {
            cellMatchID = cellIds->GetId(j);
            corrMin = corr;
          }
        }
      }
    }

    // Comparing average probability expectations and choose best line
    if(matched)
    {
      double probExp, probExp2;
      probExp =  polyData1->GetOutput()->GetCellData()->GetArray("LineProbabilityExpectation")->
                      GetComponent(i,0);
      probExp2 = polyData2->GetOutput()->GetCellData()->GetArray("LineProbabilityExpectation")->
                      GetComponent(cellMatchID,0);

      // Mark less probable line for removal
      if(probExp > probExp2)
      {
        polyData2->GetOutput()->DeleteCell(cellMatchID);
        deletedCells.push_back(cellMatchID);
      }
      else
      {
        polyData1->GetOutput()->DeleteCell(i);
        num++;
      }
    }
  }

  cout << "Number of lines deleted: input 1 = " << num << endl
       << "                         input 2 = " << deletedCells.size() << endl;

  // Deleting lines marked for removal
  polyData1->GetOutput()->RemoveDeletedCells();
  polyData2->GetOutput()->RemoveDeletedCells();

```

```
    // Combine both data sets
    vtkSmartPointer<vtkAppendPolyData> appendDataSets = vtkSmartPointer<vtkAppendPolyData>::New();
    appendDataSets->AddInput(polyData1->GetOutput());
    appendDataSets->AddInput(polyData2->GetOutput());
    appendDataSets->Update();

    // Clean duplicate points/lines
    vtkSmartPointer<vtkCleanPolyData> cleanDataSet = vtkSmartPointer<vtkCleanPolyData>::New();
    cleanDataSet->SetInput(appendDataSets->GetOutput());
    cleanDataSet->Update();

    // Copying the input data and structure to the output
    output->CopyStructure(cleanDataSet->GetOutput());
    output->GetPointData()->PassData(cleanDataSet->GetOutput()->GetPointData());
    output->GetCellData()->PassData(cleanDataSet->GetOutput()->GetCellData());
    output->GetFieldData()->PassData(cleanDataSet->GetOutput()->GetFieldData());
  }

  // Handle point features
  if(PointFeatures)
  {
    // Thresh input1 by probability expectation
    vtkSmartPointer<vtkThresholdPoints> thresh1 = vtkSmartPointer<vtkThresholdPoints>::New();
    thresh1->SetInput(input1);
    thresh1->ThresholdByUpper(ProbabilityExpectationThreshold);
    thresh1->SetInputArrayToProcess(0,0,0,0,"ProbabilityExpectation");
    thresh1->Update();

    // Thresh input2 by probability expectation
    vtkSmartPointer<vtkThresholdPoints> thresh2 = vtkSmartPointer<vtkThresholdPoints>::New();
    thresh2->SetInput(input2);
    thresh2->ThresholdByUpper(ProbabilityExpectationThreshold);
    thresh2->SetInputArrayToProcess(0,0,0,0,"ProbabilityExpectation");
    thresh2->Update();

    // Combine both data sets
    vtkSmartPointer<vtkAppendPolyData> appendDataSets = vtkSmartPointer<vtkAppendPolyData>::New();
    appendDataSets->AddInput(thresh1->GetOutput());
    appendDataSets->AddInput(thresh2->GetOutput());
    appendDataSets->Update();

    // Clean duplicate points/lines
    vtkSmartPointer<vtkCleanPolyData> cleanDataSet = vtkSmartPointer<vtkCleanPolyData>::New();
    cleanDataSet->SetInput(appendDataSets->GetOutput());
    cleanDataSet->Update();

    // Copying the input data and structure to the output
    output->CopyStructure(cleanDataSet->GetOutput());
    output->GetPointData()->PassData(cleanDataSet->GetOutput()->GetPointData());
    output->GetCellData()->PassData(cleanDataSet->GetOutput()->GetCellData());
    output->GetFieldData()->PassData(cleanDataSet->GetOutput()->GetFieldData());
  }

  return 1;
}

//----------------------------------------------------------------------------
void vtkCombineFeatureSets::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.3 vtkCreateOpinion_Vortex.cxx

```
#include "vtkCreateOpinion_Vortex.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkCreateOpinion_Vortex, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkCreateOpinion_Vortex);

//----------------------------------------------------------------------------
vtkCreateOpinion_Vortex::vtkCreateOpinion_Vortex()
{
  this->FeatureLifeNorm                     = 15;
  this->SujudiHaimes                        = true;
  this->RothPeikert                         = false;
  this->Transient                           = false;
  this->FeatureLifeNorm                     = 1;
  this->FeatureDisplacementConstant         = 0.02;
  this->ChangeInFeatureDisplacementConstant = 2.25;
  this->VortexStrengthNorm                  = 0;
  this->CurvatureNorm                       = 0;
  this->QualityNorm                         = 80;
  this->MinimumDistanceNorm                 = 0.2;
  this->Lambda2Norm                         = 1;
}

//----------------------------------------------------------------------------
int vtkCreateOpinion_Vortex::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo  = inputVector[0]->GetInformationObject(0);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);

  // get input and output
  vtkPolyData *input = vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));
  vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

  ////////////////////////////////////////
  // Constants for b, d, u equations
  double m1_b_MA,   m2_b_MA,   m1_d_MA,   m2_d_MA,   m1_u_MA,   m2_u_MA,
         m1_b_RPNE, m2_b_RPNE, m1_d_RPNE, m2_d_RPNE, m1_u_RPNE, m2_u_RPNE,
         m1_b_SHE,  m2_b_SHE,  m1_d_SHE,  m2_d_SHE,  m1_u_SHE,  m2_u_SHE,
         m1_b_RPE,  m2_b_RPE,  m1_d_RPE,  m2_d_RPE,  m1_u_RPE,  m2_u_RPE,
         m1_b_SHNE, m2_b_SHNE, m1_d_SHNE, m2_d_SHNE, m1_u_SHNE, m2_u_SHNE;
  // Master Agent
  m1_b_MA = 0.5;
  m2_b_MA = 0.5;
  m1_d_MA = -0.5;
  m2_d_MA = 0.5;
  m1_u_MA = 1.0;
  m2_u_MA = 5.0;

  // RP_Non-Extracting
  m1_b_RPNE = 0.8;
  m2_b_RPNE = 0.2;
  m1_d_RPNE = -0.8;
  m2_d_RPNE = 0.8;
  m1_u_RPNE = 1.0;
  m2_u_RPNE = 0.0;

  // SH_Extracting
  m1_b_SHE = 0.6;
  m2_b_SHE = 0.4;
  m1_d_SHE = -0.4;
```

```
m2_d_SHE = 0.4;
m1_u_SHE = 0.5;
m2_u_SHE = -10.0;

// RP_Extracting
m1_b_RPE = 0.6;
m2_b_RPE = 0.4;
m1_d_RPE = -0.4;
m2_d_RPE = 0.4;
m1_u_RPE = 0.5;
m2_u_RPE = -10.0;

// SH_Non-Extracting
m1_b_SHNE = 0.8;
m2_b_SHNE = 0.2;
m1_d_SHNE = -0.8;
m2_d_SHNE = 0.8;
m1_u_SHNE = 1.0;
m2_u_SHNE = 0.0;
////////////////////////////////////////

// creating Master Agent opinion array
vtkSmartPointer<vtkDoubleArray> MAArray = vtkSmartPointer<vtkDoubleArray>::New();
MAArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
MAArray->SetNumberOfComponents(3);
MAArray->SetNumberOfTuples(input->GetNumberOfPoints());
MAArray->SetName("MA");

// Creating array to store algorithm agent opinion when
// the Roth-Peikert algorithm extracts the cores
vtkSmartPointer<vtkDoubleArray> AARPArray = vtkSmartPointer<vtkDoubleArray>::New();
AARPArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
AARPArray->SetNumberOfComponents(3);
AARPArray->SetNumberOfTuples(input->GetNumberOfPoints());
AARPArray->SetName("AARP");

// Creating array to store algorithm agent opinion when
// the Sujudi-Haimes algorithm extracts the cores
vtkSmartPointer<vtkDoubleArray> AASHArray = vtkSmartPointer<vtkDoubleArray>::New();
AASHArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
AASHArray->SetNumberOfComponents(3);
AASHArray->SetNumberOfTuples(input->GetNumberOfPoints());
AASHArray->SetName("AASH");

// Creating array to store final opinion
vtkSmartPointer<vtkDoubleArray> finalOpinionArray = vtkSmartPointer<vtkDoubleArray>::New();
finalOpinionArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
finalOpinionArray->SetNumberOfComponents(3);
finalOpinionArray->SetNumberOfTuples(input->GetNumberOfPoints());
finalOpinionArray->SetName("FinalOpinion");

// Creating array to store probability expectation value
vtkSmartPointer<vtkDoubleArray> probExpArray = vtkSmartPointer<vtkDoubleArray>::New();
probExpArray->SetNumberOfValues(input->GetNumberOfPoints());
probExpArray->SetNumberOfComponents(1);
probExpArray->SetNumberOfTuples(input->GetNumberOfPoints());
probExpArray->SetName("ProbabilityExpectation");

// Calculating Master Agent (MA) opinion on vortex core lines
int life;
double b, d, u, normalLife, corrPrev, corrNext, corr, FD, CFD, tupleCheck, equalizer, alpha,
    beta;
std::vector <int> cellPointList;
if(Transient)
{
  for(int i = 0 ; i < input->GetNumberOfLines() ; ++i)
  {
    // Storing cell point IDs in current time step
```

```
      vtkIdList *cellPtIds;
      cellPtIds = input->GetCell(i)->GetPointIds();
      cellPointList.resize(cellPtIds->GetNumberOfIds());
      for(int j = 0 ; j < cellPtIds->GetNumberOfIds() ; ++j)
        cellPointList[j] = cellPtIds->GetId(j);

      life = int(input->GetCellData()->GetArray("FeatureLife")->GetComponent(i,0));
      normalLife = (double) life / FeatureLifeNorm;
      if(normalLife > 1) {normalLife = 1;}

      corrPrev = input->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,0);
      corrNext = input->GetCellData()->GetArray("LineCorrespondence")->GetComponent(i,1);
      corr = (corrPrev > corrNext) ? corrPrev : corrNext;
      b  = m1_b_MA * normalLife + m2_b_MA;
      if(b < 0) {b = 0;}
      d  = m1_d_MA * normalLife + m2_d_MA;
      if(d > 1) {d = 1;}
      u  = m1_u_MA/(1 + exp(m2_u_MA*corr));
      if(u > 1) {u = 1;}

      tupleCheck = b + d + u;
      if(tupleCheck > 1)
      {
        if(u == 1)
        {
          b = 0;
          d = 0;
        }
        else
        {
          equalizer = ((u + d + b)-1)/2;
          b = b - equalizer;
          d = d - equalizer;
          if(b < 0) {b = 0;}
          if(d < 0) {d = 0;}
          tupleCheck = b + d + u;
          if(tupleCheck > 1)
          {
            if(b == 0) {d = 1 - u;}
            if(d == 0) {b = 1 - u;}
          }
        }
      }

      for(int j(0) ; j < input->GetCell(i)->GetNumberOfPoints() ; ++j)
      {
        MAArray->SetComponent(cellPointList[j], 0, b);
        MAArray->SetComponent(cellPointList[j], 1, d);
        MAArray->SetComponent(cellPointList[j], 2, u);
      }
    }
  }

  else
  {
    for(int i = 0 ; i < input->GetNumberOfPoints() ; ++i)
    {
      FD = input->GetPointData()->GetArray("FeatureDisplacement")->GetComponent(i,0);
      CFD = input->GetPointData()->GetArray("ChangeInFeatureDisplacement")->GetComponent(i,0);
      b  = (-ChangeInFeatureDisplacementConstant * CFD - FeatureDisplacementConstant * FD)/2 + 1;
      if(b < 0) {b = 0;}
      d  = FeatureDisplacementConstant * FD;
      if(d > 1) {d = 1;}
      u  = ChangeInFeatureDisplacementConstant * CFD;
      if(u > 1) {u = 1;}

      tupleCheck = b + d + u;
      if(tupleCheck > 1)
```

```
      {
        if(u == 1)
        {
          b = 0;
          d = 0;
        }
        else
        {
          equalizer = ((u + d + b)-1)/2;
          b = b - equalizer;
          d = d - equalizer;
          if(b < 0) {b = 0;}
          if(d < 0) {d = 0;}
          tupleCheck = b + d + u;
          if(tupleCheck > 1)
          {
            if(b == 0) {d = 1 - u;}
            if(d == 0) {b = 1 - u;}
          }
        }
      }

      MAArray->SetComponent(i, 0, b);
      MAArray->SetComponent(i, 1, d);
      MAArray->SetComponent(i, 2, u);
    }
  }

  // initializing variables
  double vortexStrength, curvature, quality, minimumDistance, lambda2,
         normalVortexStrength, normalCurvature, normalQuality, normalAverage,
              normalMinimumDistance;

  // calculating belief tuple values as if Sujudi-Haimes was the
  // extraction algorithm for the set of vortex cores.
  if(SujudiHaimes)
  {
    for(int i = 0 ; i < input->GetNumberOfPoints() ; ++i)
    {
      // creating the AARP opinion for the Roth-Peikert algorithm when RP DOES NOT extract the
          points
      // putting vortex strength value in proper form
      vortexStrength = input->GetPointData()->GetArray("VortexStrength")->GetComponent(i,0);
      this->VortexStrengthNorm = input->GetFieldData()->GetArray("VortexStrengthGeometricMean")->
          GetComponent(0,0);
      normalVortexStrength = fabs(vortexStrength/VortexStrengthNorm);
      if(normalVortexStrength > 1) {normalVortexStrength = 1;}

      // putting curvature value in proper form
      curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
      this->CurvatureNorm = input->GetFieldData()->GetArray("CurvatureGeometricMean")->
          GetComponent(0,0);
      if(curvature > CurvatureNorm) {curvature = CurvatureNorm;}
      normalCurvature = fabs(curvature/CurvatureNorm - 1);

      // putting quality value in proper form
      quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
      if(quality > QualityNorm) {quality = QualityNorm;}
      normalQuality = fabs(quality/QualityNorm - 1);

      // finding the average of the three values
      normalAverage = (normalVortexStrength + normalCurvature + normalQuality) / 3;

      // putting minimum distance value in proper form
      minimumDistance = input->GetPointData()->GetArray("MinimumDistance")->GetComponent(i,0);
      this->MinimumDistanceNorm = input->GetFieldData()->GetArray("MinimumDistanceGeometricMean")
          ->GetComponent(0,0);
      normalMinimumDistance = fabs(minimumDistance/MinimumDistanceNorm);
```

```
    if(normalMinimumDistance > 1) {normalMinimumDistance = 1;}

    // the function that sets the belief value
    b = m1_b_RPNE * normalAverage + m2_b_RPNE;            //<------------------------------
    if(b > 1) {b = 1;}
    // the function that sets the disbelief value
    d = m1_d_RPNE * normalAverage + m2_d_RPNE;            //<------------------------------
    if(d < 0) {d = 0;}
    // the function that sets the uncertainty value
    u = m1_u_RPNE * normalMinimumDistance + m2_u_RPNE;            //<------------------u=norm *0.5----------

    tupleCheck = b + d + u;

    // checking the belief tuple to make sure it sums to 1. i.e. b+d+u=1
    if(tupleCheck > 1)
    {
      // If b + d + u doesn't equal 1 then update u and d
      equalizer = ((b + d + u) - 1) / 2;
      u = u - equalizer;
      b = b - equalizer;
      if(u < 0) {u = 0;}
      if(b < 0) {b = 0;}
      tupleCheck = u + b + d;
      if(tupleCheck > 1)
      {
        if(u == 0) {b = 1 - d;}
        if(b == 0) {u = 1 - d;}
      }
    }

  AARPArray->SetComponent(i, 0, b);
  AARPArray->SetComponent(i, 1, d);
  AARPArray->SetComponent(i, 2, u);
  }

/////////////////////////////////////////////////////////////////////////////////

  for(int i = 0 ; i < input->GetNumberOfPoints() ; ++i)
  {
    // creating the AASH opinion for the Sujudi-Haimes algorithm when SH DOES extract the points.
    // putting vortex strength value in proper form
    vortexStrength = input->GetPointData()->GetArray("VortexStrength")->GetComponent(i,0);
    this->VortexStrengthNorm = input->GetFieldData()->GetArray("VortexStrengthGeometricMean")->GetComponent(0,0);
    normalVortexStrength = fabs(vortexStrength/VortexStrengthNorm);
    if(normalVortexStrength > 1) {normalVortexStrength = 1;}

    // putting curvature value in proper form
    curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
    this->CurvatureNorm = input->GetFieldData()->GetArray("CurvatureGeometricMean")->GetComponent(0,0);
    if(curvature > CurvatureNorm) {curvature = CurvatureNorm;}
    normalCurvature = fabs(curvature/CurvatureNorm - 1);

    // putting quality value in proper form
    quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
    if(quality > QualityNorm) {quality = QualityNorm;}
    normalQuality = fabs(quality/QualityNorm - 1);

    // finding the average of the three values
    normalAverage = (normalVortexStrength + normalCurvature + normalQuality) / 3;

    // putting lambda2 value in proper form
    lambda2 = input->GetPointData()->GetArray("Lambda2")->GetComponent(i,0);

    // the function that sets the b-value
```

```
    b = m1_b_SHE * normalAverage + m2_b_SHE;      //<------Original = 0.4 * norm + 0.6-----------------
    if(b > 1) {b = 1;}
    // the function that sets the d-value
    d = m1_d_SHE * normalAverage + m2_d_SHE;      //<------Original = -0.4 * norm + 0.4-----------------
    if(d < 0) {d = 0;}
    // the function that sets the u-value
    u = m1_u_SHE/(1 + exp(m2_u_SHE*lambda2));      //<------Original = 0.5 * norm -----------------

    tupleCheck = b + d + u;

    // checking the belief tuple to make sure it sums to 1. i.e. b+d+u=1
    if(tupleCheck > 1)
    {
      // If b + d + u doesn't equal 1 then update u and d
      equalizer = ((b + d + u) - 1) / 2;
      u = u - equalizer;
      d = d - equalizer;
      if(u < 0) {u = 0;}
      if(d < 0) {d = 0;}
      tupleCheck = u + b + d;
      if(tupleCheck > 1)
      {
        if(u == 0) {d = 1 - b;}
        if(d == 0) {u = 1 - b;}
      }
    }

   AASHArray->SetComponent(i, 0, b);
   AASHArray->SetComponent(i, 1, d);
   AASHArray->SetComponent(i, 2, u);
   }
 }

/////////*********************************************************//////////

 // calculating belief tuple values as if RothPeikert was the
 // extraction algorithm for the set of vortex cores.
 if(RothPeikert)
 {
   for(int i = 0 ; i<input->GetNumberOfPoints() ; ++i)
   {
     // creating the AARP opinion for the Roth-Peikert algorithm when RP DOES extract the points
     // putting vortex strength value in proper form
     vortexStrength = input->GetPointData()->GetArray("VortexStrength")->GetComponent(i,0);
     this->VortexStrengthNorm = input->GetFieldData()->GetArray("VortexStrengthGeometricMean")->
         GetComponent(0,0);
     normalVortexStrength = fabs(vortexStrength/VortexStrengthNorm);
     if(normalVortexStrength > 1) {normalVortexStrength = 1;}

     // putting curvature value in proper form
     curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
     this->CurvatureNorm = input->GetFieldData()->GetArray("CurvatureGeometricMean")->
         GetComponent(0,0);
     normalCurvature = curvature/CurvatureNorm;
     if(normalCurvature > 1) {normalCurvature = 1;}

     // putting quality value in proper form
     quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
     if(quality > QualityNorm) {quality = QualityNorm;}
     normalQuality = fabs(quality/QualityNorm - 1);

     // finding the average of the three values
     normalAverage = (normalVortexStrength + normalCurvature + normalQuality) / 3;

     // putting lambda2 value in proper form
```

```
    lambda2 = input->GetPointData()->GetArray("Lambda2")->GetComponent(i,0);

    // the function that sets the b-value
    b = m1_b_RPE * normalAverage + m2_b_RPE;       //<-------Original = 0.4 * norm + 0.6-------------------
    if(b > 1) {b = 1;}
    // the function that sets the d-value
    d = m1_d_RPE * normalAverage + m2_d_RPE;       //<-------Original = -0.4 * norm + 0.4-------------------
    if(d < 0) {d = 0;}
    // the function that sets the u-value
    u = m1_u_RPE/(1 + exp(m2_u_RPE*lambda2));      //<-------Original = 0.5 * norm -------------------

    tupleCheck = b + d + u;

    // checking the belief tuple to make sure it sums to 1. i.e. b+d+u=1
    if(tupleCheck > 1)
    {
      // If b + d + u doesn't equal 1 then update u and d
      equalizer = ((b + d + u) - 1) / 2;
      u = u - equalizer;
      d = d - equalizer;
      if(u < 0) {u = 0;}
      if(d < 0) {d = 0;}
      tupleCheck = u + b + d;
      if(tupleCheck > 1)
      {
        if(u == 0) {d = 1 - b;}
        if(d == 0) {u = 1 - b;}
      }
    }

  AARPArray->SetComponent(i, 0, b);
  AARPArray->SetComponent(i, 1, d);
  AARPArray->SetComponent(i, 2, u);
  }

//////////////////////////////////////////////////////////////////////////////

  for(int i = 0 ; i<input->GetNumberOfPoints() ; ++i)
  {
    // creating the AASH opinion for the Sujudi-Haimes algorithm when SH DOES NOT extract the points
    // putting vortex strength value in proper form
    vortexStrength = input->GetPointData()->GetArray("VortexStrength")->GetComponent(i,0);
    this->VortexStrengthNorm = input->GetFieldData()->GetArray("VortexStrengthGeometricMean")->GetComponent(0,0);
    normalVortexStrength = fabs(vortexStrength/VortexStrengthNorm);
    if(normalVortexStrength > 1) {normalVortexStrength = 1;}

    // putting curvature value in proper form
    curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
    this->CurvatureNorm = input->GetFieldData()->GetArray("CurvatureGeometricMean")->GetComponent(0,0);
    normalCurvature = fabs(curvature/CurvatureNorm);
    if(normalCurvature > 1) {normalCurvature = 1;}

    // putting quality value in proper form
    quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
    if(quality > QualityNorm) {quality = QualityNorm;}
    normalQuality = fabs(quality/QualityNorm - 1);

    // finding the average of the three values
    normalAverage = (normalVortexStrength + normalCurvature + normalQuality) / 3;

    // putting minimum distance value in proper form
    minimumDistance = input->GetPointData()->GetArray("MinimumDistance")->GetComponent(i,0);
```

```
        this->MinimumDistanceNorm = input->GetFieldData()->GetArray("MinimumDistanceGeometricMean")
            ->GetComponent(0,0);
        normalMinimumDistance = fabs(minimumDistance/MinimumDistanceNorm);
        if(normalMinimumDistance > 1) {normalMinimumDistance = 1;}

        // the function that sets the belief value
        b = m1_b_SHNE * normalAverage + m2_b_SHNE;          //<------------------------------
        if(b > 1) {b = 1;}
        // the function that sets the disbelief value
        d = m1_d_SHNE * normalAverage + m2_d_SHNE;          //<------------------------------
        if(d < 0) {d = 0;}
        // the function that sets the uncertainty value
        u = m1_u_SHNE * normalMinimumDistance + m2_u_SHNE;      //<------------------------------

        tupleCheck = b + d + u;

        // checking the belief tuple to make sure it sums to 1. i.e. b+d+u=1
        if(tupleCheck > 1)
        {
          // If b + d + u doesn't equal 1 then update u and b
          equalizer = ((b + d + u) - 1) / 2;
          u = u - equalizer;
          b = b - equalizer;
          if(u < 0) {u = 0;}
          if(b < 0) {b = 0;}
          tupleCheck = u + b + d;
          if(tupleCheck > 1)
          {
            if(u == 0) {b = 1 - d;}
            if(b == 0) {u = 1 - d;}
          }
        }

        AASHArray->SetComponent(i, 0, b);
        AASHArray->SetComponent(i, 1, d);
        AASHArray->SetComponent(i, 2, u);
      }
    }

    // Combining all the opinions into the final opinion.
    double MA[3], AARP[3], AASH[3], MAxAASH[3], MAxAARP[3], k, finalOpinion[3], gamma;
    for(int i = 0 ; i<input->GetNumberOfPoints() ; ++i)
    {
      MAArray->GetTuple(i,MA);
      AARPArray->GetTuple(i,AARP);
      AASHArray->GetTuple(i,AASH);

      // Discounting operator
      MAxAARP[0] = MA[0] * AARP[0];
      MAxAARP[1] = MA[0] * AARP[1];
      MAxAARP[2] = MA[1] + MA[2] + MA[0] * AARP[2];

      // Discounting operator
      MAxAASH[0] = MA[0] * AASH[0];
      MAxAASH[1] = MA[0] * AASH[1];
      MAxAASH[2] = MA[1] + MA[2] + MA[0] * AASH[2];

      // Consensus operator for combining beliefs
      k = MAxAARP[2] + MAxAASH[2] - MAxAARP[2] * MAxAASH[2];
      if(k != 0)
      {
        finalOpinion[0] = (MAxAARP[0] * MAxAASH[2] + MAxAASH[0] * MAxAARP[2]) / k;
        finalOpinion[1] = (MAxAARP[1] * MAxAASH[2] + MAxAASH[1] * MAxAARP[2]) / k;
        finalOpinion[2] = (MAxAARP[2] * MAxAASH[2]) / k;
      }
      else
      {
        gamma = MAxAASH[2] / MAxAARP[2];
```

```
      finalOpinion[0] = (gamma * MAxAARP[0]+MAxAASH[0]) / (gamma + 1);
      finalOpinion[1] = (gamma * MAxAARP[1]+MAxAASH[1]) / (gamma + 1);
      finalOpinion[2] = 0;
    }
    finalOpinion[0] = (MAxAARP[0] * MAxAASH[2] + MAxAASH[0] * MAxAARP[2]) / k;
    finalOpinion[1] = (MAxAARP[1] * MAxAASH[2] + MAxAASH[1] * MAxAARP[2]) / k;
    finalOpinion[2] = (MAxAARP[2] * MAxAASH[2]) / k;

    finalOpinionArray->SetTuple(i, finalOpinion);

    // calculating the probability expectation value
    probExpArray->SetValue(i, finalOpinion[0]+0.5*finalOpinion[2]);
  }

  // adding arrays to the input data set
  input->GetPointData()->AddArray(MAArray);
  input->GetPointData()->AddArray(AASHArray);
  input->GetPointData()->AddArray(AARPArray);
  input->GetPointData()->AddArray(finalOpinionArray);
  input->GetPointData()->AddArray(probExpArray);

  // Copying the input data and structure to the output
  output->CopyStructure(input);
  output->GetPointData()->PassData(input->GetPointData());
  output->GetCellData()->PassData(input->GetCellData());
  output->GetFieldData()->PassData(input->GetFieldData());

  return 1;
}

//----------------------------------------------------------------------------
void vtkCreateOpinion_Vortex::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.4 vtkCurvature.cxx

```
#include "vtkCurvature.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkCurvature, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkCurvature);

vtkCurvature::vtkCurvature()
{
  this->MultiSegmentCurvature = false;
  this->VelocityFieldCurvature = false;
  this->PointwiseCurvature = false;
}

int vtkCurvature::FillInputPortInformation(int port, vtkInformation *info)
{
}

//----------------------------------------------------------------------------
int vtkCurvature::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo = inputVector[0]->GetInformationObject(0);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);
```

```
  // get the input and ouptut
  vtkPolyData *input = vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));
  vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

////////////////////////////////////////////////////////////////////////////////

  if(MultiSegmentCurvature)
  {
    // initializing values
    double p0[3], p1[3], p2[3];
    double a, b, c, sum1, sum2, sum3, radius, curvature;
    double logSum(0),logCurvature(0), logMean(0), gMean(0);

    // Initializing the curvature array to add to polydata
    vtkSmartPointer<vtkDoubleArray> curvatureArray = vtkSmartPointer<vtkDoubleArray>::New();
    curvatureArray->SetNumberOfComponents(1);
    curvatureArray->SetNumberOfTuples(input->GetNumberOfPoints());
    curvatureArray->SetName("Curvature");

    // compute geometric mean of curvature values
    vtkSmartPointer<vtkDoubleArray> curvatureGMean = vtkSmartPointer<vtkDoubleArray>::New();
    curvatureGMean->SetNumberOfComponents(1);
    curvatureGMean->SetNumberOfTuples(1);
    curvatureGMean->SetName("CurvatureGeometricMean");

    for(int i = 0 ; i < input->GetNumberOfLines() ; i++)
    {
      for (int j = 0 ; j < input->GetCell(i)->GetNumberOfPoints(); j++)
      {
        // getting point Ids to use later
        vtkSmartPointer<vtkIdList> ptIds = vtkSmartPointer<vtkIdList>::New();
        input->GetCellPoints(i,ptIds);

        // First core point:
        // use 1s,t 3rd, and 5th points in line
        if(j == 0)
        {
          input->GetCell(i)->GetPoints()->GetPoint(j,p0);
          input->GetCell(i)->GetPoints()->GetPoint(j+2,p1);
          input->GetCell(i)->GetPoints()->GetPoint(j+4,p2);
        }

        // Second core point:
        // use 1st, 3rd, and 5th points in line
        else if(j == 1)
        {
          input->GetCell(i)->GetPoints()->GetPoint(j-1,p0);
          input->GetCell(i)->GetPoints()->GetPoint(j+1,p1);
          input->GetCell(i)->GetPoints()->GetPoint(j+3,p2);
        }

        // Second to last core point:
        // use 1st, 3rd, and 5th points at end of line
        else if(j == input->GetCell(i)->GetNumberOfPoints()-2)
        {
          input->GetCell(i)->GetPoints()->GetPoint(j-3,p0);
          input->GetCell(i)->GetPoints()->GetPoint(j-1,p1);
          input->GetCell(i)->GetPoints()->GetPoint(j+1,p2);
        }

        // Last core point:
        // use 1st, 3rd, and 5th points at end of line
        else if(j == input->GetCell(i)->GetNumberOfPoints()-1)
        {
          input->GetCell(i)->GetPoints()->GetPoint(j-4,p0);
          input->GetCell(i)->GetPoints()->GetPoint(j-2,p1);
          input->GetCell(i)->GetPoints()->GetPoint(j,p2);
```

```
        }

        // All other core points:
        // use points 2 away
        else
        {
          input->GetCell(i)->GetPoints()->GetPoint(j-2,p0);
          input->GetCell(i)->GetPoints()->GetPoint(j,p1);
          input->GetCell(i)->GetPoints()->GetPoint(j+2,p2);
        }

        // Calculating distances between points
        a = sqrt(pow(p1[0]-p0[0],2) + pow(p1[1]-p0[1],2) + pow(p1[2]-p0[2],2));
        b = sqrt(pow(p2[0]-p1[0],2) + pow(p2[1]-p1[1],2) + pow(p2[2]-p1[2],2));
        c = sqrt(pow(p2[0]-p0[0],2) + pow(p2[1]-p0[1],2) + pow(p2[2]-p0[2],2));
        sum1 = -a+b+c;      sum2 = a-b+c;      sum3 = a+b-c;

        // Case of points on a straight line
        if (sum1 < 1e-100 || sum2 < 1e-100 || sum3 < 1e-100)
          curvature = 0;
        else
        {
          // Calculating radius of circumcircle
          radius = a*b*c / sqrt((a+b+c)*(-a+b+c)*(a-b+c)*(a+b-c));
          curvature = 1/radius;
        }

        if(curvature < 0.00001)
          curvature = 0.00001;

        // Compute logarithm sum
        logCurvature = log10(curvature);
        logSum += logCurvature;

        curvatureArray->SetComponent(ptIds->GetId(j),0,curvature);
      }
    }

    // Compute geometric mean
    logMean = logSum / input->GetNumberOfPoints();
    gMean = pow(10.0,logMean);

    curvatureGMean->SetValue(0,gMean);

    input->GetPointData()->AddArray(curvatureArray);
    input->GetFieldData()->AddArray(curvatureGMean);

    // Copying the input data and structure to the output
    output->CopyStructure(input);
    output->GetPointData()->PassData(input->GetPointData());
    output->GetCellData()->PassData(input->GetCellData());
    output->GetFieldData()->PassData(input->GetFieldData());
  }

////////////////////////////////////////////////////////////////////////////////

  else if(VelocityFieldCurvature)
  {
    // calculate curvature vector
    vtkSmartPointer<vtkArrayCalculator> calc = vtkSmartPointer<vtkArrayCalculator>::New();
    calc->AddScalarVariable("a_x", "TensorXVelocity", 0);
    calc->AddScalarVariable("a_y", "TensorXVelocity", 1);
    calc->AddScalarVariable("a_z", "TensorXVelocity", 2);
    calc->AddScalarVariable("v_x", "NormVelocity", 0);
    calc->AddScalarVariable("v_y", "NormVelocity", 1);
    calc->AddScalarVariable("v_z", "NormVelocity", 2);
    calc->SetResultArrayName("CurvatureVector");
    calc->SetFunction("iHat*((v_y*a_z - v_z*a_y)/(v_x*v_x + v_y*v_y + v_z*v_z)^1.5) +"
```

```
                        "jHat*((v_z*a_x - v_x*a_z)/(v_x*v_x + v_y*v_y + v_z*v_z)^1.5) +"
                        "kHat*((v_x*a_y - v_y*a_x)/(v_x*v_x + v_y*v_y + v_z*v_z)^1.5)");
    calc->SetInput(input);
    calc->ReleaseDataFlagOn();
    calc->Update();

    // calculate curvature from curvature vector
    vtkSmartPointer<vtkArrayCalculator> calc2 = vtkSmartPointer<vtkArrayCalculator>::New();
    calc2->AddScalarVariable("c_x", "CurvatureVector", 0);
    calc2->AddScalarVariable("c_y", "CurvatureVector", 1);
    calc2->AddScalarVariable("c_z", "CurvatureVector", 2);
    calc2->SetResultArrayName("Curvature");
    calc2->SetFunction("(c_x*c_x + c_y*c_y + c_z*c_z)^0.5");
    calc2->SetInput(calc->GetOutput());
    calc2->ReleaseDataFlagOn();
    calc2->Update();

    // compute geometric mean of curvature values
    vtkSmartPointer<vtkDoubleArray> curvatureGMean = vtkSmartPointer<vtkDoubleArray>::New();
    curvatureGMean->SetNumberOfComponents(1);
    curvatureGMean->SetNumberOfTuples(1);
    curvatureGMean->SetName("CurvatureGeometricMean");

    double logSum(0);
    for (int i = 0 ; i < input->GetNumberOfPoints() ; i++)
    {
        double logCurvature = log10(calc2->GetOutput()->GetPointData()->GetArray("Curvature")->GetComponent(i,0));
        logSum += logCurvature;
    }

    double logMean = logSum / input->GetNumberOfPoints();
    double gMean = pow(10.0, logMean);

    curvatureGMean->SetTuple1(0, gMean);

    output->GetFieldData()->AddArray(curvatureGMean);

    // Copying the input data and structure to the output
    output->CopyStructure(calc2->GetOutput());
    output->GetPointData()->PassData(calc2->GetOutput()->GetPointData());
    output->GetCellData()->PassData(calc2->GetOutput()->GetCellData());
    output->GetFieldData()->PassData(input->GetFieldData());
  }

///////////////////////////////////////////////////////////////////////////////

  else if(PointwiseCurvature)
  {
    // Obtaining change in feature displacement at each point
    // Initializing the array and naming variables
    vtkSmartPointer<vtkDoubleArray> curvatureArray = vtkSmartPointer<vtkDoubleArray>::New();
    curvatureArray->SetNumberOfValues(input->GetNumberOfPoints());
    curvatureArray->SetNumberOfComponents(1);
    curvatureArray->SetNumberOfTuples(input->GetNumberOfPoints());
    curvatureArray->SetName("Curvature");

    // Create the tree
    vtkSmartPointer<vtkOctreePointLocator> octree = vtkSmartPointer<vtkOctreePointLocator>::New();
    octree->SetDataSet(input);
    octree->BuildLocator();

    //declare variables
    double distance[5];
    double point_holder[15];

    //Loop through each point
```

```
for (int j(0); j <  input->GetNumberOfPoints(); j++)
{
  // Find the k closest points to (0,0,0)
  unsigned int k = 5;
  double testPoint[3];

  testPoint[0] = input->GetPoints()->GetData()->GetComponent(j,0);
  testPoint[1] = input->GetPoints()->GetData()->GetComponent(j,1);
  testPoint[2] = input->GetPoints()->GetData()->GetComponent(j,2);

  vtkSmartPointer<vtkIdList> result = vtkSmartPointer<vtkIdList>::New();

  octree->FindClosestNPoints(k, testPoint, result);

  //loop for every k-th point
  for(vtkIdType i = 0; i < k ; i++)
  {
    //find the distance between each point of interest
    double p[3];
    input->GetPoint(result->GetId(i), p);

    if(i == 0)
    {
      distance[0]=sqrt(pow((testPoint[0]-p[0]),2)+pow((testPoint[1]-p[1]),2)+pow((testPoint
          [2]-p[2]),2));
      point_holder[0]=p[0];
      point_holder[1]=p[1];
      point_holder[2]=p[2];
    }
    if(i == 1)
    {
      distance[1]=sqrt(pow((testPoint[0]-p[0]),2)+pow((testPoint[1]-p[1]),2)+pow((testPoint
          [2]-p[2]),2));
      point_holder[3]=p[0];
      point_holder[4]=p[1];
      point_holder[5]=p[2];
    }
    if(i == 2)
    {
      distance[2]=sqrt(pow((testPoint[0]-p[0]),2)+pow((testPoint[1]-p[1]),2)+pow((testPoint
          [2]-p[2]),2));
      point_holder[6]=p[0];
      point_holder[7]=p[1];
      point_holder[8]=p[2];
    }
    if(i == 3)
    {
      distance[3]=sqrt(pow((testPoint[0]-p[0]),2)+pow((testPoint[1]-p[1]),2)+pow((testPoint
          [2]-p[2]),2));
      point_holder[9]=p[0];
      point_holder[10]=p[1];
      point_holder[11]=p[2];
    }
    if(i == 4)
    {
      distance[4]=sqrt(pow((testPoint[0]-p[0]),2)+pow((testPoint[1]-p[1]),2)+pow((testPoint
          [2]-p[2]),2));
      point_holder[12]=p[0];
      point_holder[13]=p[1];
      point_holder[14]=p[2];
    }
    distance[0]=sqrt(pow((point_holder[12]-point_holder[9]),2)+pow((point_holder[13]-
        point_holder[10]),2)+pow((point_holder[14]-point_holder[11]),2));
  }

  // Set up some variables to make curvature calculation easier
  double a, b, c, sum1, sum2, sum3, radius, curvature;
  a = distance[0];  b = distance[3];  c = distance[4];
```

```
      sum1 = -a+b+c;    sum2 = a-b+c;     sum3 = a+b-c;

      // case of points on a straight line
      if (sum1 < 1e-100 || sum2 < 1e-100 || sum3 < 1e-100)
        curvature = 0;
      // Calculate radius of circle circumscribed by 3 points
      else
      {
        radius = a*b*c / sqrt((a+b+c)*(-a+b+c)*(a-b+c)*(a+b-c));
        curvature = 1/radius;
      }

      // Make zero curvature low for geometric mean
      if(curvature < 0.00000000001)
        curvature = 0.000001;

      //calculate curvature based on radius
      curvatureArray->SetValue(j, curvature);
    }

    // adding computed arrays to input1
    input->GetPointData()->AddArray(curvatureArray);

    // Copying the input data and structure to the output
    output->CopyStructure(input);
    output->GetPointData()->PassData(input->GetPointData());
    output->GetCellData()->PassData(input->GetCellData());
    output->GetFieldData()->PassData(input->GetFieldData());
  }

  return 1;
}

//----------------------------------------------------------------------------
void vtkCurvature::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
  os << indent << "MultiSegmentCurvature: "   << (this->MultiSegmentCurvature ? "On\n" : "Off\n");
  os << indent << "VelocityFieldCurvature: "  << (this->VelocityFieldCurvature ? "On\n" : "Off\n");
  os << indent << "PointwiseCurvature: "      << (this->PointwiseCurvature ? "On\n" : "Off\n");
}
```

B.4.5 vtkFeatureAttributes.cxx

```
#include "vtkFeatureAttributes.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkFeatureAttributes, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkFeatureAttributes);

//----------------------------------------------------------------------------
vtkFeatureAttributes::vtkFeatureAttributes()
{

}

//----------------------------------------------------------------------------
int vtkFeatureAttributes::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
```

```
// get the info objects
vtkInformation *inInfo  = inputVector[0]->GetInformationObject(0);
vtkInformation *outInfo = outputVector->GetInformationObject(0);

// get input and output
vtkPolyData *input = vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));
vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

// Creating line length array
vtkSmartPointer<vtkDoubleArray> lengthArray = vtkSmartPointer<vtkDoubleArray>::New();
lengthArray->SetNumberOfValues(input->GetNumberOfLines());
lengthArray->SetNumberOfComponents(1);
lengthArray->SetNumberOfTuples(input->GetNumberOfLines());
lengthArray->SetName("LineLength");

// Creating line vortex strength array
vtkSmartPointer<vtkDoubleArray> strengthArray = vtkSmartPointer<vtkDoubleArray>::New();
strengthArray->SetNumberOfValues(input->GetNumberOfLines());
strengthArray->SetNumberOfComponents(1);
strengthArray->SetNumberOfTuples(input->GetNumberOfLines());
strengthArray->SetName("LineVortexStrength");

// Creating line curvature array
vtkSmartPointer<vtkDoubleArray> curvatureArray = vtkSmartPointer<vtkDoubleArray>::New();
curvatureArray->SetNumberOfValues(input->GetNumberOfLines());
curvatureArray->SetNumberOfComponents(1);
curvatureArray->SetNumberOfTuples(input->GetNumberOfLines());
curvatureArray->SetName("LineCurvature");

// Creating line quality array
vtkSmartPointer<vtkDoubleArray> qualityArray = vtkSmartPointer<vtkDoubleArray>::New();
qualityArray->SetNumberOfValues(input->GetNumberOfLines());
qualityArray->SetNumberOfComponents(1);
qualityArray->SetNumberOfTuples(input->GetNumberOfLines());
qualityArray->SetName("LineQuality");

// Creating tracking ID array
vtkSmartPointer<vtkIntArray> trackingIDArray = vtkSmartPointer<vtkIntArray>::New();
trackingIDArray->SetNumberOfValues(input->GetNumberOfLines());
trackingIDArray->SetNumberOfComponents(1);
trackingIDArray->SetNumberOfTuples(input->GetNumberOfLines());
trackingIDArray->SetName("TrackingID");

// Creating line correspondence array
vtkSmartPointer<vtkDoubleArray> correspondenceArray = vtkSmartPointer<vtkDoubleArray>::New();
correspondenceArray->SetNumberOfValues(input->GetNumberOfLines());
correspondenceArray->SetNumberOfComponents(2);
correspondenceArray->SetNumberOfTuples(input->GetNumberOfLines());
correspondenceArray->SetName("LineCorrespondence");

// Creating event array
vtkSmartPointer<vtkIntArray> eventArray = vtkSmartPointer<vtkIntArray>::New();
eventArray->SetNumberOfValues(input->GetNumberOfLines());
eventArray->SetNumberOfComponents(1);
eventArray->SetNumberOfTuples(input->GetNumberOfLines());
eventArray->SetName("SplitMergeEvent");

// Creating feature lifetime array
vtkSmartPointer<vtkIntArray> featureLifeArray = vtkSmartPointer<vtkIntArray>::New();
featureLifeArray->SetNumberOfValues(input->GetNumberOfLines());
featureLifeArray->SetNumberOfComponents(1);
featureLifeArray->SetNumberOfTuples(input->GetNumberOfLines());
featureLifeArray->SetName("FeatureLife");

// Computing average attributes for each line
std::vector<int> cellPointList;
for(int i = 0 ; i < input->GetNumberOfLines() ; i++)
{
```

```
    // Putting cell point ids into an array
    vtkIdList *cellPtIds;
    cellPtIds = input->GetCell(i)->GetPointIds();
    cellPointList.resize(cellPtIds->GetNumberOfIds());
    for(int j = 0 ; j < cellPtIds->GetNumberOfIds() ; j++)
    {
      cellPointList[j] = cellPtIds->GetId(j);
    }

    // Instantiating variables
    double strengthSum(0), curvatureSum(0), qualitySum(0),
           avgStrength, avgCurvature, avgQuality;

    // Summing up point values in the line
    for(int j = 0 ; j < input->GetCell(i)->GetNumberOfPoints() ; j++)
    {
      strengthSum  += input->GetPointData()->GetArray("VortexStrength")->GetComponent(cellPointList[j],0);
      curvatureSum += input->GetPointData()->GetArray("Curvature")->GetComponent(cellPointList[j],0);
      qualitySum   += input->GetPointData()->GetArray("Quality")->GetComponent(cellPointList[j],0);
    }

    // Finding the average of each attribute
    avgStrength  = strengthSum  / input->GetCell(i)->GetNumberOfPoints();
    avgCurvature = curvatureSum / input->GetCell(i)->GetNumberOfPoints();
    avgQuality   = qualitySum   / input->GetCell(i)->GetNumberOfPoints();

    // Setting attribute arrays
    lengthArray->SetValue(i,input->GetPointData()->GetArray("l")->GetComponent(cellPointList[0],0));
    strengthArray->SetValue(i,avgStrength);
    curvatureArray->SetValue(i,avgCurvature);
    qualityArray->SetValue(i,avgQuality);
    trackingIDArray->SetValue(i,0);
    eventArray->SetValue(i,0);
    featureLifeArray->SetValue(i,1);
    correspondenceArray->SetTuple2(i,-100,-100);
  }

  // adding arrays to the input data set
  input->GetCellData()->AddArray(lengthArray);
  input->GetCellData()->AddArray(strengthArray);
  input->GetCellData()->AddArray(curvatureArray);
  input->GetCellData()->AddArray(qualityArray);
  input->GetCellData()->AddArray(trackingIDArray);
  input->GetCellData()->AddArray(eventArray);
  input->GetCellData()->AddArray(featureLifeArray);
  input->GetCellData()->AddArray(correspondenceArray);

  // Copying the input data and structure to the output
  output->CopyStructure(input);
  output->GetPointData()->PassData(input->GetPointData());
  output->GetCellData()->PassData(input->GetCellData());
  output->GetFieldData()->PassData(input->GetFieldData());

  return 1;
}

//----------------------------------------------------------------------------
void vtkFeatureAttributes::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.6 vtkFeatureLifetime.cxx

```
#include "vtkFeatureLifetime.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkFeatureLifetime, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkFeatureLifetime);

//----------------------------------------------------------------------------
vtkFeatureLifetime::vtkFeatureLifetime()
{
  this->CalculateFeatureLifetime = true;
  this->SetFeatureLifetime = false;
}

//----------------------------------------------------------------------------
int vtkFeatureLifetime::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo  = inputVector[0]->GetInformationObject(0);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);

  // get input and output
  vtkPolyData *input = vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));
  vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

  if(CalculateFeatureLifetime)
  {
    for(int i = 0 ; i < input->GetNumberOfLines() ; i++)
    {
      // Getting tracking ID of line
      int trackingID = int(input->GetCellData()->GetArray("TrackingID")->GetComponent(i, 0));

      // Incrementing feature life array by 1 at index of tracking ID
      FeatureLifeArray->SetComponent(trackingID, 0, FeatureLifeArray->GetComponent(trackingID,0)+1);
    }
  }

  if(SetFeatureLifetime)
  {
    // creating new tracking ID array - current time step
    vtkSmartPointer<vtkIntArray> lifetimeArray = vtkSmartPointer<vtkIntArray>::New();
    lifetimeArray->SetNumberOfValues(input->GetNumberOfLines());
    lifetimeArray->SetNumberOfComponents(1);
    lifetimeArray->SetNumberOfTuples(input->GetNumberOfLines());
    lifetimeArray->SetName("FeatureLife");

    // Copying old feature life arrays into new ones
    for(int i = 0 ; i < input->GetNumberOfLines() ; i++)
      lifetimeArray->SetComponent(i,0,input->GetCellData()->GetArray("FeatureLife")->GetComponent(i,0));

    // Removing old feature life array from the input data set
    input->GetCellData()->RemoveArray("FeatureLife");

    // Setting feature lifetimes for each tracking ID
    std::vector <int> cellPointList;
    for(int i = 0 ; i < input->GetNumberOfLines() ; i++)
    {
      // Getting tracking ID of line
      double trackingID = input->GetCellData()->GetArray("TrackingID")->GetComponent(i,0);

```

```
      // Setting array for current line
      // Untracked path receives a lifetime of 1
      if(trackingID == 0)
        lifetimeArray->SetValue(i,1);

      // Tracked path receives measured lifetime
      else
        lifetimeArray->SetComponent(i,0,FeatureLifeArray->GetComponent(trackingID,0));
    }

    input->GetCellData()->AddArray(lifetimeArray);
  }

  // Copying the input data and structure to the output
  output->CopyStructure(input);
  output->GetPointData()->PassData(input->GetPointData());
  output->GetCellData()->PassData(input->GetCellData());
  output->GetFieldData()->PassData(input->GetFieldData());

  return 1;
}

//----------------------------------------------------------------------------
void vtkFeatureLifetime::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.7 vtkLambdaTwo.cxx

```
#include "vtkLambdaTwo.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkLambdaTwo, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkLambdaTwo);

//----------------------------------------------------------------------------
vtkLambdaTwo::vtkLambdaTwo()
{
  this->SetNumberOfInputPorts(1);
  this->SetNumberOfOutputPorts(1);
  this->VelocityArrayName = "Velocity";
}

//----------------------------------------------------------------------------
int vtkLambdaTwo::FillInputPortInformation(int, vtkInformation *info)
{
  info->Set(vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE(), "vtkDataSet");
  return 1;
}

//----------------------------------------------------------------------------
int vtkLambdaTwo::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo = inputVector[0]->GetInformationObject(0);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);

  // get the input and ouptut
  vtkDataSet *input = vtkDataSet::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));
  vtkDataSet *output = vtkDataSet::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));
```

```
// Computing lambda_2 at each point
// Creating array to hold lambda_2
vtkSmartPointer<vtkDoubleArray> lambda2Array = vtkSmartPointer<vtkDoubleArray>::New();
lambda2Array->SetName("Lambda2");
lambda2Array->SetNumberOfComponents(1);
lambda2Array->SetNumberOfTuples(input->GetNumberOfPoints());

// Computing vorticity at each point
// Creating array to hold vorticity
vtkSmartPointer<vtkDoubleArray> vorticityArray = vtkSmartPointer<vtkDoubleArray>::New();
vorticityArray->SetName("Vorticity");
vorticityArray->SetNumberOfComponents(3);
vorticityArray->SetNumberOfTuples(input->GetNumberOfPoints());

// creating arrays to hold velocity components
vtkSmartPointer<vtkDoubleArray> xVelocity = vtkSmartPointer<vtkDoubleArray>::New();
vtkSmartPointer<vtkDoubleArray> yVelocity = vtkSmartPointer<vtkDoubleArray>::New();
vtkSmartPointer<vtkDoubleArray> zVelocity = vtkSmartPointer<vtkDoubleArray>::New();
xVelocity->SetName("xVelocity");
yVelocity->SetName("yVelocity");
zVelocity->SetName("zVelocity");
xVelocity->SetNumberOfValues(input->GetNumberOfPoints());
yVelocity->SetNumberOfValues(input->GetNumberOfPoints());
zVelocity->SetNumberOfValues(input->GetNumberOfPoints());

for(int i = 0 ; i < input->GetNumberOfPoints() ; i++)
{
  xVelocity->SetValue(i,input->GetPointData()->GetArray(VelocityArrayName)->GetComponent(i,0));
  yVelocity->SetValue(i,input->GetPointData()->GetArray(VelocityArrayName)->GetComponent(i,1));
  zVelocity->SetValue(i,input->GetPointData()->GetArray(VelocityArrayName)->GetComponent(i,2));
}
input->GetPointData()->AddArray(xVelocity);
input->GetPointData()->AddArray(yVelocity);
input->GetPointData()->AddArray(zVelocity);

// Calculating the gradient of x-velocity
vtkSmartPointer<vtkGradientFilter> vgf1 = vtkSmartPointer<vtkGradientFilter>::New();
vgf1->SetInput(input);
vgf1->SetInputScalars(vtkDataObject::FIELD_ASSOCIATION_POINTS,"xVelocity");
vgf1->SetResultArrayName("uGradient");
vgf1->Update();

// Calculating the gradient of y-velocity
vtkSmartPointer<vtkGradientFilter> vgf2 = vtkSmartPointer<vtkGradientFilter>::New();
vgf2->SetInput(vgf1->GetOutput());
vgf2->SetInputScalars(vtkDataObject::FIELD_ASSOCIATION_POINTS,"yVelocity");
vgf2->SetResultArrayName("vGradient");
vgf2->Update();

// Calculating the gradient of z-velocity
vtkSmartPointer<vtkGradientFilter> vgf3 = vtkSmartPointer<vtkGradientFilter>::New();
vgf3->SetInput(vgf2->GetOutput());
vgf3->SetInputScalars(vtkDataObject::FIELD_ASSOCIATION_POINTS,"zVelocity");
vgf3->SetResultArrayName("wGradient");
vgf3->Update();

// putting the velocity gradients into one 9 component array
vtkSmartPointer<vtkDoubleArray> vgArray = vtkSmartPointer<vtkDoubleArray>::New();
vgArray->SetName("VelocityGradients");
vgArray->SetNumberOfComponents(9);
vgArray->SetNumberOfTuples(input->GetNumberOfPoints());
double J[3][3];

for(int i = 0 ; i < input->GetNumberOfPoints() ; i++)
{
  J[0][0] = vgf1->GetOutput()->GetPointData()->GetArray("uGradient")->GetComponent(i,0);
  J[0][1] = vgf1->GetOutput()->GetPointData()->GetArray("uGradient")->GetComponent(i,1);
```

```
    J[0][2] = vgf1->GetOutput()->GetPointData()->GetArray("uGradient")->GetComponent(i,2);
    J[1][0] = vgf2->GetOutput()->GetPointData()->GetArray("vGradient")->GetComponent(i,0);
    J[1][1] = vgf2->GetOutput()->GetPointData()->GetArray("vGradient")->GetComponent(i,1);
    J[1][2] = vgf2->GetOutput()->GetPointData()->GetArray("vGradient")->GetComponent(i,2);
    J[2][0] = vgf3->GetOutput()->GetPointData()->GetArray("wGradient")->GetComponent(i,0);
    J[2][1] = vgf3->GetOutput()->GetPointData()->GetArray("wGradient")->GetComponent(i,1);
    J[2][2] = vgf3->GetOutput()->GetPointData()->GetArray("wGradient")->GetComponent(i,2);
    vgArray->SetComponent(i,0,J[0][0]);
    vgArray->SetComponent(i,1,J[0][1]);
    vgArray->SetComponent(i,2,J[0][2]);
    vgArray->SetComponent(i,3,J[1][0]);
    vgArray->SetComponent(i,4,J[1][1]);
    vgArray->SetComponent(i,5,J[1][2]);
    vgArray->SetComponent(i,6,J[2][0]);
    vgArray->SetComponent(i,7,J[2][1]);
    vgArray->SetComponent(i,8,J[2][2]);

    // Calculating the transpose of the velocity gradient tensor
    double Jt[3][3];
    vtkMath::Transpose3x3(J,Jt);

    // Calculating the strain rate tensor
    double S[3][3];
    S[0][0] = 0.5*(J[0][0]+Jt[0][0]);
    S[0][1] = 0.5*(J[0][1]+Jt[0][1]);
    S[0][2] = 0.5*(J[0][2]+Jt[0][2]);
    S[1][0] = 0.5*(J[1][0]+Jt[1][0]);
    S[1][1] = 0.5*(J[1][1]+Jt[1][1]);
    S[1][2] = 0.5*(J[1][2]+Jt[1][2]);
    S[2][0] = 0.5*(J[2][0]+Jt[2][0]);
    S[2][1] = 0.5*(J[2][1]+Jt[2][1]);
    S[2][2] = 0.5*(J[2][2]+Jt[2][2]);

    // Calculating the vorticity tensor
    double O[3][3];
    O[0][0] = 0.5*(J[0][0]-Jt[0][0]);
    O[0][1] = 0.5*(J[0][1]-Jt[0][1]);
    O[0][2] = 0.5*(J[0][2]-Jt[0][2]);
    O[1][0] = 0.5*(J[1][0]-Jt[1][0]);
    O[1][1] = 0.5*(J[1][1]-Jt[1][1]);
    O[1][2] = 0.5*(J[1][2]-Jt[1][2]);
    O[2][0] = 0.5*(J[2][0]-Jt[2][0]);
    O[2][1] = 0.5*(J[2][1]-Jt[2][1]);
    O[2][2] = 0.5*(J[2][2]-Jt[2][2]);

    // Calculating vorticity vector
    double vorticity[3];
    vorticity[0] = 2*O[1][2];
    vorticity[1] = 2*O[2][0];
    vorticity[2] = 2*O[0][1];

    vorticityArray->SetTuple3(i,vorticity[0],vorticity[1],vorticity[2]);

    // Combining the strain rate and vorticity tensors (S^2+O^2)
    double **t = new double*[3];
    for(int j = 0 ; j < 3 ; j++)
      t[j] = new double[3];
    t[0][0] = pow(S[0][0],2)+pow(O[0][0],2);
    t[0][1] = pow(S[0][1],2)+pow(O[0][1],2);
    t[0][2] = pow(S[0][2],2)+pow(O[0][2],2);
    t[1][0] = pow(S[1][0],2)+pow(O[1][0],2);
    t[1][1] = pow(S[1][1],2)+pow(O[1][1],2);
    t[1][2] = pow(S[1][2],2)+pow(O[1][2],2);
    t[2][0] = pow(S[2][0],2)+pow(O[2][0],2);
    t[2][1] = pow(S[2][1],2)+pow(O[2][1],2);
    t[2][2] = pow(S[2][2],2)+pow(O[2][2],2);

    // Calculating the eigenvalues of S^2 + O^2
```

```
    double *eigenvalues = new double[3];
    double **eigenvectors = new double*[3];
    for(int j = 0 ; j < 3 ; j++)
      eigenvectors[j] = new double[3];
    vtkMath::Jacobi(t,eigenvalues,eigenvectors);

    // Deleting pointers
    for(int j = 0 ; j < 3 ; j++)
    {
      delete [] t[j];
      delete [] eigenvectors[j];
    }
    delete [] t;
    delete [] eigenvectors;

    // Setting the value of lambda_2 at the point
    lambda2Array->SetComponent(i,0,eigenvalues[1]);

    delete [] eigenvalues;
  }

  input->GetPointData()->AddArray(vgArray);
  input->GetPointData()->AddArray(lambda2Array);
  input->GetPointData()->AddArray(vorticityArray);

  // Removing unrequired arrays
  input->GetPointData()->RemoveArray("xVelocity");
  input->GetPointData()->RemoveArray("yVelocity");
  input->GetPointData()->RemoveArray("zVelocity");

  // Copying the input data and structure to the output
  output->CopyStructure(input);
  output->GetPointData()->PassData(input->GetPointData());
  output->GetCellData()->PassData(input->GetCellData());
  output->GetFieldData()->PassData(input->GetFieldData());

  return 1;
}

//----------------------------------------------------------------------------
void vtkLambdaTwo::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

B.4.8 vtkTimeDerivatives.cxx

```
#include "vtkTimeDerivatives.h"

#include <headers.h>

vtkCxxRevisionMacro(vtkTimeDerivatives, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkTimeDerivatives);

//----------------------------------------------------------------------------
vtkTimeDerivatives::vtkTimeDerivatives()
{
  this->SetNumberOfInputPorts(1);
  this->SetNumberOfOutputPorts(1);
  this->TimeStep = 0;
  this->Velocity1ArrayName = "Velocity1";
  this->Velocity2ArrayName = "Velocity2";
  this->Velocity3ArrayName = "Velocity3";
  this->ForwardDifference = false;
  this->BackwardDifference = false;
```

```
  this->CentralDifference = true;
}

//----------------------------------------------------------------------------
int vtkTimeDerivatives::FillInputPortInformation(int, vtkInformation *info)
{
  info->Set(vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE(), "vtkDataSet");
  info->Set(vtkAlgorithm::INPUT_IS_REPEATABLE(), 1);

  return 1;
}

//----------------------------------------------------------------------------
int vtkTimeDerivatives::RequestData(
  vtkInformation *vtkNotUsed(request),
  vtkInformationVector **inputVector,
  vtkInformationVector *outputVector)
{
  // get the info objects
  vtkInformation *inInfo1 = inputVector[0]->GetInformationObject(0);
  vtkInformation *inInfo2 = inputVector[0]->GetInformationObject(1);
  vtkInformation *inInfo3 = inputVector[0]->GetInformationObject(2);
  vtkInformation *outInfo = outputVector->GetInformationObject(0);

  // get the 2 inputs and 1 ouptut
  // input1 is the data object that we will be calculating the derivatives for
  vtkDataSet *input1 = vtkDataSet::SafeDownCast(inInfo1->Get(vtkDataObject::DATA_OBJECT()));
  vtkDataSet *input2 = vtkDataSet::SafeDownCast(inInfo2->Get(vtkDataObject::DATA_OBJECT()));
  vtkDataSet *input3 = vtkDataSet::SafeDownCast(inInfo3->Get(vtkDataObject::DATA_OBJECT()));
  vtkDataSet *output = vtkDataSet::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

  // Obtaining the first derivative in time at each point
  // Initializing the array and naming variables
  vtkSmartPointer<vtkDoubleArray> FirstDerArray = vtkSmartPointer<vtkDoubleArray>::New();
  FirstDerArray->SetNumberOfValues(input1->GetNumberOfPoints()*3);
  FirstDerArray->SetNumberOfComponents(3);
  FirstDerArray->SetNumberOfTuples(input1->GetNumberOfPoints());
  FirstDerArray->SetName("Time1stDerivatives");

  // Obtaining the second derivative in time at each point
  // Initializing the array and naming variables
  vtkSmartPointer<vtkDoubleArray> SecondDerArray = vtkSmartPointer<vtkDoubleArray>::New();
  SecondDerArray->SetNumberOfValues(input1->GetNumberOfPoints()*3);
  SecondDerArray->SetNumberOfComponents(3);
  SecondDerArray->SetNumberOfTuples(input1->GetNumberOfPoints());
  SecondDerArray->SetName("Time2ndDerivatives");

  double u1Der, u2Der, v1Der, v2Der, w1Der, w2Der;

////////////////////////////////////////////////////////////////////////////////

  if(ForwardDifference)
  {
    // input1 ----->time i
    // input2 ----->time i+1
    // input3 ----->time i+2

    for (int i = 0 ; i < input1->GetNumberOfPoints() ; i++)
    {
      // Compute forward 1st derivatives (2nd-order)
      u1Der = (-3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,0) +
            4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) -
            input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(2*TimeStep);
      v1Der = (-3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,1) +
            4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) -
            input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(2*TimeStep);
      w1Der = (-3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,2) +
            4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) -
```

```
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(2*TimeStep);

      // Compute forward 2nd derivatives (1st-order)
      u2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,0) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(pow(TimeStep,2));
      v2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,1) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(pow(TimeStep,2));
      w2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,2) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(pow(TimeStep,2));

      FirstDerArray->SetComponent(i, 0, u1Der);
      FirstDerArray->SetComponent(i, 1, v1Der);
      FirstDerArray->SetComponent(i, 2, w1Der);

      SecondDerArray->SetComponent(i, 0, u2Der);
      SecondDerArray->SetComponent(i, 1, v2Der);
      SecondDerArray->SetComponent(i, 2, w2Der);
    }

    // adding computed arrays to input1
    input1->GetPointData()->AddArray(FirstDerArray);
    input1->GetPointData()->AddArray(SecondDerArray);

    // Copying the input data and structure to the output
    output->CopyStructure(input1);
    output->GetPointData()->PassData(input1->GetPointData());
    output->GetCellData()->PassData(input1->GetCellData());
  }

////////////////////////////////////////////////////////////////////////////////

  else if(BackwardDifference)
  {
    // input1 ----->time i
    // input2 ----->time i-1
    // input3 ----->time i-2

    for (int i = 0 ; i < input1->GetNumberOfPoints() ; i++)
    {
      // Compute backward 1st derivatives (2nd-order)
      u1Der = (3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,0) -
          4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(2*TimeStep);
      v1Der = (3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,1) -
          4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(2*TimeStep);
      w1Der = (3*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,2) -
          4*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(2*TimeStep);

      // Compute backward 2nd derivatives (1st-order)
      u2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,0) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(pow(TimeStep,2));
      v2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,1) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) +
          input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(pow(TimeStep,2));
      w2Der = (input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,2) -
          2*input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) +
```

```
      input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(pow(TimeStep,2));

    FirstDerArray->SetComponent(i, 0, u1Der);
    FirstDerArray->SetComponent(i, 1, v1Der);
    FirstDerArray->SetComponent(i, 2, w1Der);

    SecondDerArray->SetComponent(i, 0, u2Der);
    SecondDerArray->SetComponent(i, 1, v2Der);
    SecondDerArray->SetComponent(i, 2, w2Der);
  }

  // adding computed arrays to input1
  input1->GetPointData()->AddArray(FirstDerArray);
  input1->GetPointData()->AddArray(SecondDerArray);

  // Copying the input data and structure to the output
  output->CopyStructure(input1);
  output->GetPointData()->PassData(input1->GetPointData());
  output->GetCellData()->PassData(input1->GetCellData());
}

////////////////////////////////////////////////////////////////////////////

else if(CentralDifference)
{
  // input1 ----->time i
  // input2 ----->time i+1
  // input3 ----->time i-1

  for (int i = 0 ; i < input1->GetNumberOfPoints() ; i++)
  {
    // Compute central 1st derivatives (2nd-order)
    u1Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) -
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(2*TimeStep);
    v1Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) -
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(2*TimeStep);
    w1Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) -
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(2*TimeStep);

    // Compute central 2nd derivatives (2nd-order)
    u2Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,0) -
        2*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,0) +
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,0))/(pow(TimeStep,2));
    v2Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,1) -
        2*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,1) +
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,1))/(pow(TimeStep,2));
    w2Der = (input2->GetPointData()->GetArray(Velocity2ArrayName)->GetComponent(i,2) -
        2*input1->GetPointData()->GetArray(Velocity1ArrayName)->GetComponent(i,2) +
        input3->GetPointData()->GetArray(Velocity3ArrayName)->GetComponent(i,2))/(pow(TimeStep,2));

    FirstDerArray->SetComponent(i, 0, u1Der);
    FirstDerArray->SetComponent(i, 1, v1Der);
    FirstDerArray->SetComponent(i, 2, w1Der);

    SecondDerArray->SetComponent(i, 0, u2Der);
    SecondDerArray->SetComponent(i, 1, v2Der);
    SecondDerArray->SetComponent(i, 2, w2Der);
  }

  // adding computed arrays to input1
  input1->GetPointData()->AddArray(FirstDerArray);
  input1->GetPointData()->AddArray(SecondDerArray);

  // Copying the input data and structure to the output
```

```
      output->CopyStructure(input1);
      output->GetPointData()->PassData(input1->GetPointData());
      output->GetCellData()->PassData(input1->GetCellData());
      output->GetFieldData()->PassData(input1->GetFieldData());
    }

  return 1;
}

//----------------------------------------------------------------------------
void vtkTimeDerivatives::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```