2010-07-08

A Computational Fluid Dynamics Feature Extraction Method Using Subjective Logic

Clifton H. Mortensen
Brigham Young University - Provo

A Computational Fluid Dynamics Feature Extraction Method Using Subjective Logic

Clifton H. Mortensen

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
Scott L. Thomson
Julie Vanderhoff

Department of Mechanical Engineering

Brigham Young University

August 2010

ABSTRACT

A Computational Fluid Dynamics Feature Extraction Method

Using Subjective Logic

Clifton H. Mortensen

Department of Mechanical Engineering

Master of Science

Computational fluid dynamics simulations are advancing to correctly simulate highly complex fluid flow problems that can require weeks of computation on expensive high performance clusters. These simulations can generate terabytes of data and pose a severe challenge to a researcher analyzing the data. Presented in this document is a general method to extract computational fluid dynamics flow features concurrent with a simulation and as a post-processing step to drastically reduce researcher post-processing time. This general method uses software agents governed by subjective logic to make decisions about extracted features in converging and converged data sets. The software agents are designed to work inside the Concurrent Agent-enabled Feature Extraction concept [1] and operate efficiently on massively parallel high performance computing clusters. Also presented is a specific application of the general feature extraction method to vortex core lines. Each agent's belief tuple is quantified using a pre-defined 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. A simulation of a blunt fin is run showing convergence of the horseshoe vortex core to its final spatial location at 60% of the converged solution. Agents correctly select between two vortex core extraction algorithms and correctly identify the expected probabilities of vortex cores as the solution converges. A simulation of a delta wing is run showing coherently extracted primary vortex cores as early as 16% of the converged solution. Agents select primary vortex cores extracted by the Sujudi-Haimes algorithm as the most probable primary cores. These simulations show concurrent feature extraction is possible and that intelligent agents following the general feature extraction method are able to make appropriate decisions about converging and converged features based on pre-defined information.

Keywords: Clifton Mortensen, feature extraction, subjective logic, computational fluid dynamics, agent-based data mining, vortex core, massive data set post-processing

ACKNOWLEDGMENTS

The author would like to thank his thesis advisor, Dr. Steven Gorrell, for his constant guidance, support and editing. The author would also like to thank committee members, Dr. Scott Thomson and Dr. Julie Vanderhoff, for their input and aid. The author would like to thank the Air Force Office of Scientific Research for sponsoring part of this project and further acknowledge Dr. Rhonda Vickery and John van der Zwaag for use of the vortex extraction code. The author would also like to thank Dr. Robert Woodley and 21st Century Systems, Inc. for their help and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

a	Atomicity
$\mathbf{a}$	Acceleration
AA	Algorithm Agent
$\mathrm{AA_E}$	Extracting Algorithm Agent
$\mathrm{AA_{NE}}$	Non-extracting Algorithm Agent
b	Belief
$\mathbf{b}$	Jerk
CAFÉ	Concurrent Agent-enabled Feature Extraction
d	Disbelief
D	Discriminant
E	Probability expectation
e_0	Eigenvector corresponding to real eigenvalue
FD	Feature Displacement
h	Helicity
ℓ	Length
M	Mach number
MA	Master Agent
$\mathbf{n}$	Normalized eigenvector corresponding to real eigenvalue
$\mathbf{P}$	Cartesian position vector
r	Radius
R	Region in a feature
RP	Roth-Peikert
SH	Sujudi-Haimes
$\mathbf{t}$	Tangent vector
u	Uncertainty
$\mathbf{V}$	Velocity vector
$\mathbf{V_r}$	Reduced velocity vector
α	Angle of attack
ΔFD	Change in feature displacement
ζ	Vorticity
θ	Quality
ω	Opinion; Belief tuple
$\otimes$	Discounting operator
$\oplus$	Consensus operator
∇	Gradient operator

CHAPTER 1. INTRODUCTION

1.1 Motivation

Computational fluid dynamics (CFD) simulations numerically solve the governing equations of fluid motion. A common formulation is the Navier-Stokes equations formed from the application of Newton's second law to fluid motion combined with the conservation of mass and energy equations. The result is nonlinear partial differential equations with analytical solutions available in only the simplest cases. Through the use of parallel codes and supercomputers CFD simulations have increased in grid resolution and numerical accuracy to a point of correctly simulating highly complex fluid flow problems. Many of these advanced simulations are run on multi-node computing clusters requiring weeks to reach full convergence and generating terabytes of data. Yao [2] and List [3] have run unsteady Reynolds-averaged Navier-Stokes (URANS) simulations of gas turbine engine transonic fan stages with 166 million grid points and entire fans with over 300 million grid points respectively. These types of simulations typically run on 900 to 1200 processors, generate terabytes of raw data, and take hundreds of thousands of hours in computation time on expensive computing clusters to obtain converged solutions.

A common challenge when conducting high-fidelity simulations is the analysis of large amounts of data. Currently the time to analyze many massive data sets is equivalent to the wall time of computing the solution which can be on the order of hundreds of hours. To post-process large data sets there are a variety of software programs and techniques which can be quite discipline dependent. One approach common to post-processing massive time-accurate CFD data sets in turbomachinery applications requires a researcher to slowly sift through data to find useful information based on intuition and previous experience. Other approaches spanning many disciplines utilize software concepts and packages such as Evita [4], FieldView and ParaView. These types of programs are meant to post-process and visualize massive data sets and commonly include techniques such as feature extraction, construction of iso-surfaces using important scalar values and automated visualization based on researcher input criteria.

As simulations continue to increase in size new post-processing techniques and ideas are needed to build on previous techniques to help a researcher quickly parse through data to find useful information aiding in design improvement. The Concurrent Agent-enabled Feature Extraction (CAFÉ) [1] concept is currently being developed by Brigham Young University and 21st Century Systems, Inc. to meet this challenge. CAFÉ is an agent-based data mining software system designed to be a plug-in for CFD packages and to do concurrent analysis of CFD data. This research is a part of the CAFÉ concept.

1.2 Feature Extraction

Fluid flows are comprised of basic features that serve as building blocks for the overall flow. Post [5] defines features as "phenomena, structures or objects in a data set, that are of interest for a certain research or engineering problem." Some features of particular interest in high-fidelity time-accurate CFD simulations are vortices, shock waves, and separation and attachment lines. Usually flow features can be located in CFD data sets through visual inspection of streamlines, or flow properties such as pressure, which can be a cumbersome process of visualization and searching. Also, a simple visual inspection may not reveal all pertinent features in a data set. Feature extraction works on the problem of locating relevant features in a data set without visualization and does so in an automated fashion that requires little to no researcher input. Feature extraction is an automated process by which a feature is precisely located in a data set. It is especially useful because it can prioritize data for further analysis and provide insight to relevant flow physics. Also, if a full 3D transient data set is too large to be saved to a hard disk the data size of extracted features are orders of magnitude smaller allowing them all to be stored with ease.

Once features are extracted they can then be visualized making them understandable and useful by displaying information such as feature location, strength, interaction, creation, and dissipation. Extracted features have no real significance until they can be visualized to show where spatially and when temporally in a data set they occur, and how they affect the surrounding flow. Fortunately, vortices and separation and attachment lines may be visualized simply by lines, while a shock wave may be visualized by an opaque surface. Figure 1.1 gives an example visualization of an extracted shock surface surrounding a hypersonic vehicle. When visualized, features can give a researcher quick insight into where design improvement may be made.

A current limitation to feature extraction from massive data sets is the time it takes to extract features is often long enough to hinder post-processing rather than help. For example, if a software package is not able to run in parallel requiring one processor to be used on a data set that possibly took hundreds of processors to compute, the time to extract features could be too large to be useful. Also feature extraction has been done after a CFD simulation is converged requiring extra computation time after a simulation is complete. Feature extraction can be advanced by extracting features in parallel and concurrent with a running simulation allowing features to be available when a simulation is complete if not before a simulation is converged. This will decrease post-processing time and decrease turn around time for product development.

Feature extraction is often done algorithmically where each feature requires its own unique feature extraction algorithm. Unfortunately, for each feature there is not one markedly superior algorithm that extracts correctly all features within the spatiotemporal flow domain, but rather multiple algorithms per feature that have been optimized for specific flow conditions. Ma [6] states, "it is clear that there is no single best shock detection...algorithm." Likewise, Roth [7] states, "none of the [vortex extraction] methods is clearly superior in all the tested data sets." This leaves the problem of having to run a data set through multiple extraction algorithms and parse through the data output to find relevant features.

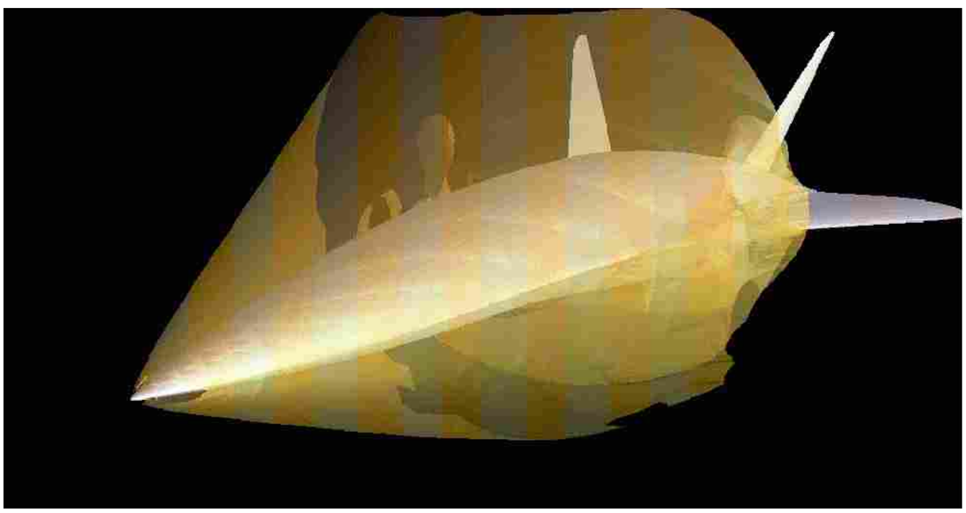


Figure 1.1: Visualization of an extracted shock surface. [5]

When extracting vortices Roth suggested that "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 [7]."

While Roth's statement was specifically about extracting vortices, the idea can be extended to any flow feature of interest with corresponding extraction algorithms. In this research multiple extraction algorithms are used to locate features instead of using only one extraction algorithm per feature. This leaves the job of trying to combine the output from each of these algorithms based on their strengths and weaknesses into one coherent highly probable set of features. To do this, intelligent software agents governed by subjective logic are used.

1.3 Software Agents

An intelligent software agent is a piece of software that can act autonomously without any user intervention. It is able to make decisions and decide the outcome of situations without being told by an end user what actions to take. An intelligent agent may use a pre-defined set of information to decide what action to take in any given situation or it may use a form of machine learning to identify what course of action is best. In this research agents are given a pre-defined set of information to govern their behavior which is then quantified and input into agent opinions defined by subjective logic [8–10].

Subjective logic is a mathematics based logic system that forms 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 agent opinion to give an a priori weight to a systems uncertainty. In this research the common assumption of $a = 0.5$ is used allowing atomicity to be dropped from the agent opinion leaving only belief, disbelief, and uncertainty. These three elements are shown in Eq. 1.1 where ω represents the entire opinion, or belief tuple.

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

Through the use of opinions agents are able to make intelligent decisions. Three opinion values in subjective logic allow agents to form opinions that are not strictly one way or the other. In other words, an agent has some subjectivity about the outcome of a situation. An agent can find, based on given information, how probable an outcome is rather than simply reducing the outcome to a binary situation of will, or will not, occur. Subjective logic is also useful when making decisions about uncertain situations and/or when data is missing or incomplete. Missing or incomplete data can be taken into account in an agent's uncertainty value. During concurrent feature extraction data will be highly uncertain requiring agents to make suitable decisions.

1.4 Objective

The objective of this research is to develop a method to extract flow features from CFD data sets while simulations are converging and as a post-processing step when simulations are converged. The developed method will be designed as a part of the CAFÉ concept. The method will be able to use more than one feature extraction algorithm per feature utilizing the strengths of each included algorithm. The general method will use software agents governed by subjective logic to determine the expected probability of extracted features from converging data sets and to aid in decisions made about features from converged data sets. It will be shown how to set each value in an agent opinion so that a final opinion may be formed for extracted features. This general method will be validated by applying the method to vortex core lines. Two CFD simulations will be given that replicate concurrent feature extraction of vortex core lines showing it is possible to extract features before CFD simulations are fully converged. Also, these two simulations will be used to validate the vortex core extraction method. It will be shown that the method can make appropriate decisions about the probability of vortex cores before and after a simulation has converged.

The developed method will contribute to the ability to use multiple feature extraction algorithms optimized for specific flow conditions and combine their feature outputs into one coherent and highly probable set of features that precisely locates all features within the spatiotemporal flow domain. Also, the method contributes a means to properly recognize the probability of features in converging data sets allowing an interpretation of features and their interactions with the flow before a CFD simulation has converged. The two CFD simulations contribute an understanding of concurrent feature extraction and insight to when flow features may be extracted and when flow features are spatially correct.

1.5 Overview

This document is organized as follows: Chapter 2 gives background on vortex extraction, subjective logic, trust networks, and some large data set post-processing programs. Chapter 3 gives the general method to extract flow features from CFD data sets using software agents governed by subjective logic. Chapter 4 gives a specific application of the general method to vortex core lines. Chapter 5 gives results of two CFD simulations that have vortex cores extracted from converging and converged data sets using the method described in Chapter 4. Chapter 6 gives recommendations for future research and Chapter 7 gives conclusions about the research.

CHAPTER 2. BACKGROUND & LITERATURE REVIEW

In this chapter vortices are defined along with some of their characteristics and a background is given on vortex extraction. A background is also given on subjective logic, trust networks and novel large data set post-processing concepts.

2.1 Fluid Vortices

Vortices are common occurrences in many types of engineering flows. They arise where there are large amounts of vorticity, or flow rotation. They can be effective and useful devices to mix flow or can account for high losses in applications such as turbomachinery. Accordingly, in some applications vortices may be sought after to increase their size and strength or in other applications vortices may be found to eliminate them and their corresponding losses. Vortex extraction is useful in either of these situations as it can give an effective visualization of the size, strength, and location of a vortex. Once a vortex is located, geometry or boundary conditions may be varied to find how to properly influence the properties of a vortex.

Fluid vortices can be defined in various ways and their definitions are ambiguous which lead to several extraction methods. A commonly accepted vortex definition in the feature extraction community comes from Robinson [11] which states, "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 behavior can be seen in Figure 2.1 where a fluid vortex extends from the wingtip of a moving aircraft and smoke allows an effective visualization. In this picture the vortex core is close to normal with the picture making a normal reference plane. It can be seen that on this reference plane the streamlines exhibit a circular pattern as given in the vortex definition.

Figure 2.1: Wingtip vortex [12].

This vortex definition leads to a physical vortex structure with two interdependent parts: the vortex core line and the swirling fluid motion around the core. At the core there is no velocity measured relative to the vortex in any direction except along the core line. All swirling motion contained within a vortex rotates about the core. This dual structure gives rise to two separate ideas for extracting vortices: extraction of a vortex region and extraction of a vortex core line.

2.2 Extracting Vortex Regions

One of the most basic ideas to extract a vortex region is to find areas of high vorticity where vorticity is calculated using Eq. 2.1. The idea is that areas in the flow with high vorticity are vortices. This may not always be true because other flow conditions have high vorticity but are not vortices such as boundary layers. Villasenor and Vincent [13] use vorticity requirements to extract vortex tubes.

$$\zeta = \nabla \times \mathbf{V} \tag{2.1}$$

Another idea to extract a vortex region is to locate areas with high helicity where helicity is calculated using Eq. 2.2. The dot product in helicity removes the vorticity component normal to the velocity vector giving a more accurate extraction of vortex regions than using vorticity only. All areas of high helicity, similar to areas of high vorticity, may not be vortices such as boundary layers. Levy [14] and Yates [15] utilize helicity when extracting vortex regions. Two other common vortex region extraction methods are utilized by Robinson [16] and Jeong and Hussain [17].

$$h = (\nabla \times \mathbf{V}) \cdot \mathbf{V} \tag{2.2}$$

Extraction of vortex regions using simple filtering criterion such as high vorticity, or high helicity, gives a quick and rough estimate of the shape, size, and position of a vortex. The numerical complexity of these methods is commonly low with a low computation time that is beneficial in applications requiring large data sets. Also, it is beneficial to work with quantities that are common in fluid dynamics such as vorticity.

The most glaring shortcoming of extracting a vortex region is that a vortex is not located precisely. In other words, at what exact spatial location is the center of the vortex, or the point at which the fluid rotates about? Extraction of vortex core lines solves the problem of precisely locating the center of a fluid vortex. Figure 2.2 displays an extracted vortex core line (light blue) with rotating streamlines and an overlain Line Integral Convolution (LIC) image to help visualize the circular motion of the flow around the core. LIC is a texture based technique for visualizing vector fields. Here the vortex has been located precisely and the streamlines will continue to rotate around the core until the vortex dissipates.

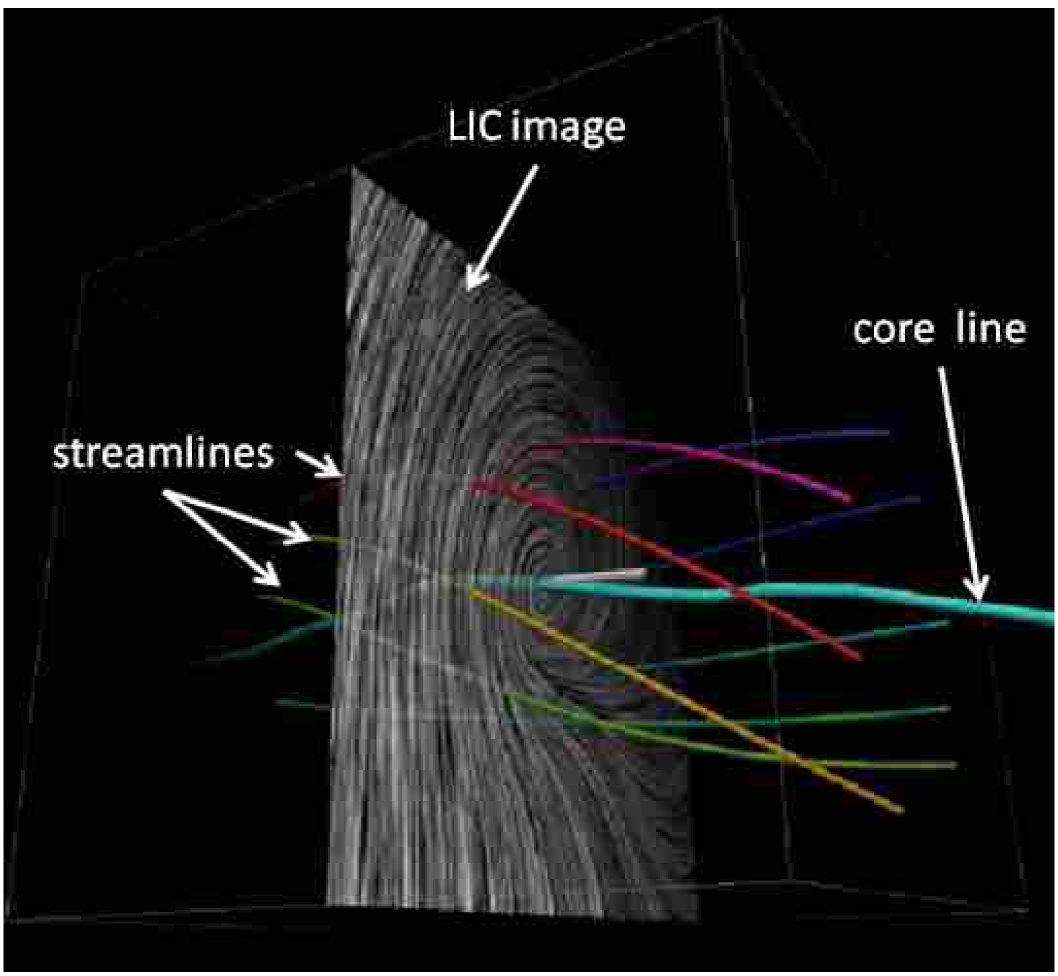


Figure 2.2: Display of rotating streamlines around a vortex core line (light blue) with an overlain LIC image normal to the core direction [7].

2.3 Extracting Vortex Core Lines

Many algorithms have been developed to locate vortex core lines and in this research it was determined that two algorithms were markedly superior. Two criteria helped to determine which algorithms fit our application: How accurately did the algorithm identify all fluid vortices within the flow domain? and would the algorithm adequately identify vortices in applications where concurrent data mining would be required such as turbomachine simulations?

2.3.1 Sujudi-Haimes Algorithm

The first vortex extraction algorithm chosen for this research was the Sujudi-Haimes (SH) algorithm [18]. The SH algorithm was designed as a robust vortex core line detection algorithm for use in large 3D transient problems. It is used in the CFD post-processing software packages EnSight 9 [19] and pV3 [20]. The SH algorithm has multiple software implementations. The first implementation was put forward by the creators of the algorithm which computes the eigenvalues of the velocity gradient matrix, shown in Eq. 2.3, at every cell location.

$$\nabla \mathbf{V} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \tag{2.3}$$

Once the three eigenvalues are found, only the cells where the set of eigenvalues contain one real valued and two complex conjugate eigenvalues corresponding to a saddle-spiral critical point are selected and the rest are discarded. Next a quantity called the reduced velocity is computed from Eq. 2.4 where **n** is the normalized eigenvector corresponding to the one real eigenvalue. The reduced velocity is then linearly interpolated across the entire cell. Two locations are then found along the cell boundaries where $\mathbf{V_r} = 0$ and these two locations form a line segment within the cell that is added as part of a vortex core line.

$$\mathbf{V_r} = \mathbf{V} - (\mathbf{V} \cdot \mathbf{n})\mathbf{n} \tag{2.4}$$

While the original implementation of the SH algorithm is correct, it is also computationally expensive. Solving for eigenvalues is expensive and eigenvalues everywhere in the computational domain must be computed before individual node locations may be filtered out. Roth [7] puts forward another definition of the SH algorithm that is less computationally expensive utilizing his parallel vectors operator. The underlying assumption of $\mathbf{V_r} = 0$ is that the velocity vector is parallel to the eigenvector obtained from the real eigenvalue (Eq. 2.5). If the velocity vector is not parallel to the eigenvector from the real eigenvalue then the condition $\mathbf{V_r} = 0$ cannot hold.

$$\mathbf{V} \parallel \mathbf{e_0} \tag{2.5}$$

When finding locations where the eigenvector from the real eigenvalue is parallel to the flow velocity Roth notes that this is equivalent to finding locations where

$$\mathbf{V} \parallel \nabla\mathbf{V} \cdot \mathbf{V} \tag{2.6}$$

as the velocity vector itself can be an eigenvector. This equation can then be reformulated to

$$\mathbf{V} \parallel \mathbf{a} \tag{2.7}$$

since

$$\mathbf{a} = \frac{D\mathbf{V}}{Dt} = \frac{\partial \mathbf{V}}{\partial t} + \nabla\mathbf{V} \cdot \mathbf{V} \tag{2.8}$$

where

$$\frac{\partial \mathbf{V}}{\partial t} = 0 \tag{2.9}$$

because we are only considering an instantaneous snapshot of the flow field.

This leads to the software implementation of the SH algorithm that finds all locations in the flow domain where Eq. 2.7 holds and then thresholds those points with a discriminant greater than zero ($D > 0$) to ensure that there is only one real eigenvalue and two complex conjugates. The value for the discriminant comes from the matrix in Eq. 2.3. Selected points that pass all criteria are then aggregated into lines. The latter implementation of the SH first order vortex extraction algorithm given in Eq. 2.7 is used in this research.

The SH algorithm was designed to locate vortices in linear flow fields that occur where there are spiral saddle and spiral node critical points. It works well when vortex core lines are straight and when the vortex strength is high (high rotational velocity about the core). The SH algorithm may extract erroneous core lines when the core line is curved or when the core line has a low strength (low rotational velocity about the core). It also may extract erroneous lines when the velocity along the core line is accelerating.

2.3.2 Roth-Peikert Algorithm

The second vortex extraction algorithm chosen for this research was the Roth-Peikert (RP) algorithm [7, 21]. The RP algorithm is specifically designed to extract fluid vortices in turbomachine simulations. Some of the testing done on the RP algorithm has come from CFD simulations of Kaplan turbines used in hydroelectric facilities. What makes the RP algorithm unique and well suited for complex flow fields is the fact that the RP algorithm is designed to locate curved rather than straight vortex core lines. Curved vortices commonly appear in turbomachinery data sets as the fluid travels through the flow domain in a curved fashion influenced by rotating blade rows.

Figure 2.3 displays a perfectly semi-circular vortex core line with two circling streamlines. The three vectors **V**, **a**, and **b** are velocity, acceleration and jerk respectively. It can be seen that for the case of a perfectly semi-circular vortex core line the condition $\mathbf{V} \parallel \mathbf{a}$ from the first order method of SH does not hold. That method will be unable to extract this type of core. This figure does show that a separate condition may be used to extract this type of core line:

$$\mathbf{V} \parallel \mathbf{b}. \tag{2.10}$$

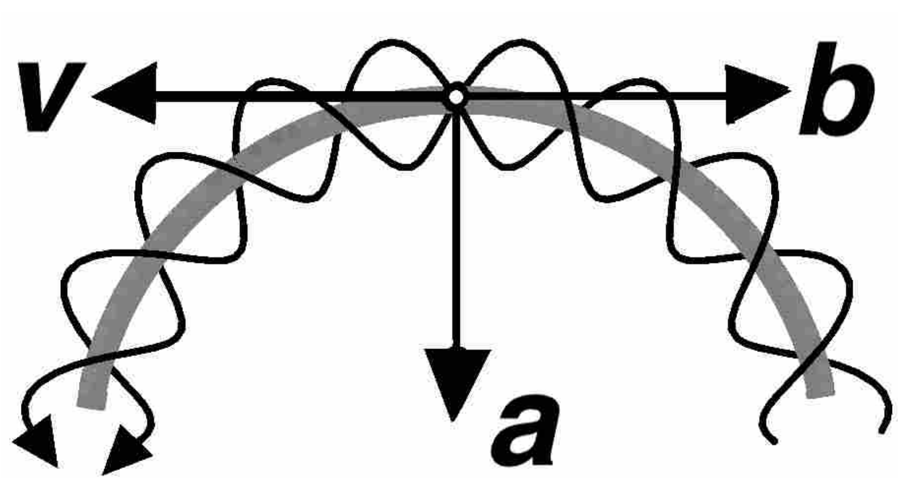


Figure 2.3: Display of rotating streamlines around a curved vortex core line. [7]

The fluid jerk is defined as the second substantial derivative of the fluid velocity

$$\mathbf{b} = \frac{D^2\mathbf{V}}{Dt^2} = \frac{D\mathbf{a}}{Dt} = \frac{\partial \mathbf{a}}{\partial t} + \nabla \mathbf{a} \cdot \mathbf{V} \tag{2.11}$$

where

$$\frac{\partial \mathbf{a}}{\partial t} = 0. \tag{2.12}$$

because we are only considering an instantaneous snapshot of the flow field. Substituting Eq. 2.8 into Eq. 2.11 we get

$$\mathbf{b} = \nabla(\nabla \mathbf{V} \cdot \mathbf{V}) \cdot \mathbf{V} \tag{2.13}$$

yielding

$$\mathbf{V} \parallel \nabla(\nabla \mathbf{V} \cdot \mathbf{V}) \cdot \mathbf{V}. \tag{2.14}$$

The RP algorithm takes advantage of Eq. 2.14 and proceeds in a point by point fashion to find locations in the flow domain where the the fluid jerk is parallel to the fluid velocity. The points that don't meet the condition are dropped and the points that do meet the condition are aggregated into vortex core lines.

Neither of these two algorithms (SH & RP) adequately extracts all vortex core lines in all flow situations. Both of the algorithms have strengths and weaknesses that are complementary to the other. In this research, we use both algorithms to maximize our chances of adequately detecting all vortex core lines within the spatiotemporal flow domain. Where one algorithm might fail, the other algorithm may not and then an agent-based decision can be made to chose which vortex core lines from the algorithm outputs are the most probable.

2.3.3 Other Vortex Core Extraction Methods

Another useful vortex core extraction algorithm developed by Jiang [22] is a 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. This algorithm is not used in this research. Other vortex core extraction algorithms have been given by Banks and Singer [23], Globus et al. [24], Pagendarm et al. [25], and Miura and Kida [26].

2.4 Vortex Characteristics

Vortex characteristics are useful inputs to the agents that can aid in their decisions about the expected probability of features. While there are many ways to characterize a vortex, the specific vortex characteristics used in this research are quality, strength and curvature.

2.4.1 Quality

Quality is a vortex characteristic originally defined by Roth [7]. In this research quality is the angle between a vortex core line and its associated velocity vector. This value is given in Eq. 2.15 where $\mathbf{t}$ is the tangent vector to the vortex core line and $\mathbf{V}$ is the local velocity vector. Generally a vortex core line is not a streamline which would be represented by a quality criterion of zero, but with the assumption that there is close to zero rotational motion about a vortex core the core line will be similar to a streamline. This behavior yields a small angle between the core line and the velocity vector, or a low quality value. Figure 2.4 gives a graphical representation of quality. The red vectors are velocity vectors and the black line is a vortex core. Near the left of the core the core has a low quality value and near the right the core has a high quality value. At the location where the quality value is low the vortex core is more likely to be extracted in its proper spatial position and where the quality value is high there is a higher chance that the vortex has been extracted spuriously. Commonly, a vortex core will either have many low quality values or many high quality values rather than an equal distribution of both.

$$\theta = \cos^{-1}\left(\frac{\mathbf{V}}{|\mathbf{V}|}\cdot\frac{\mathbf{t}}{|\mathbf{t}|}\right) \tag{2.15}$$

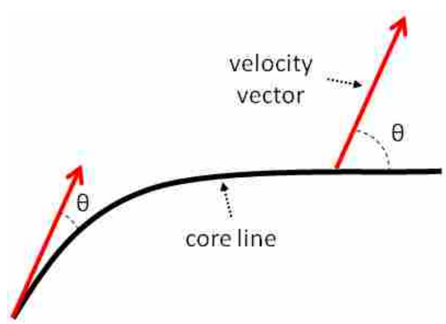


Figure 2.4: Vortex quality measure at both ends of a hypothetical core line.

2.4.2 Strength

Vortex strength measures how fast flow rotates locally around a vortex's core. In a two dimensional flow field the speed of rotation can be measured by the absolute value of the imaginary part of an eigenvalue of $\nabla\mathbf{V}$. This definition is tied to two dimensional vector field topology which gives a value for the imaginary part of the eigenvalue only in rotating flows at critical points such as a repelling focus, attracting focus and center. An overview of two dimensional vector field topology in fluid flows is given by Helman and Hesselink [27, 28].

Since vortex cores are usually contained in three dimensions instead of two, a standard two dimensional plane needs to be defined to use the two dimensional criteria. Roth [7] suggests to use a plane perpendicular to the velocity at the vortex core since the core position and the velocity at the core are already known. The local flow can be projected onto this two dimensional plane and the local vortex strength found from the imaginary part of either eigenvalue.

2.4.3 Curvature

Curvature is defined as the reciprocal of the radius of a circle as shown in Eq. 2.16 where r is the radius of the circle. Figure 2.3 depicts a perfectly curved vortex core line with rotating streamlines. Since vortex cores are straight line segments connected end to end a curvature definition is needed that can be applied to these straight segments. The curvature of a vortex core is calculated from the circle's radius that contains the two core line endpoints and the midpoint.

$$Curvature = \frac{1}{r} \tag{2.16}$$

2.5 Subjective Logic

Subjective logic is a propositional logic that gives a human estimate for the probable outcome of a situation. Commonly, standard propositional logic is used to find if an outcome is true or false but as humans we don't always think in absolutes. For example, what if you were asked are you going to have a good day today at work? The answer to this question is probably not a strict yes or no but rather likely to contain some variability. If the boss gives me that raise then

I will have a good day. If the noisy person in the cube next to me is loud, then I probably will not have a good day. All the events that affect having a good day commonly do not have a strict either/or occurrence as well. Subjective logic works with all this variability to give an answer that is more human. Since the boss is not likely to give me the raise and based on past work days of my coworker being loud than there is a low belief that my day will go well, a high disbelief that my day will go well and some uncertainty as to the outcome because some unexpected good thing might happen. It is the three values of belief, disbelief, and uncertainty given in Eq. 1.1 that make up the subjective logic opinion on how the day will go.

To form an opinion each component of the belief tuple is given a numerical value which allows the opinion to be given an exact measure. To maintain uniformity in an opinion the summation of an opinion is always equal to unity which is displayed in Eq. 2.17. Quantification and Eq. 2.17 allow operators to work with opinions in a mathematically rigorous fashion.

$$b + d + u = 1 \tag{2.17}$$

2.5.1 Opinion Triangle

An opinion can easily be visualized using the opinion triangle shown in Figure 2.5 where $\omega_x = (0.40,\ 0.10,\ 0.50,\ 0.60)$ is given as an example opinion. This opinion contains four values because atomicity has been retained. The opinion can be located by traversing any two of the lines connecting the mid-point of a triangle leg and an intersection of two triangle legs. Each of these three lines is solid with an arrow at the tip and labeled either belief, disbelief or uncertainty. Also, they go from 0 to 1 and have ten steps with a width of one tenth per step. To locate ω_x travel 0.10, or one step, on the disbelief line starting at 0. At this step there is a dotted line orthogonal to the disbelief line. The opinion must lie somewhere on this dotted line. Now travel 0.50, or 5 steps, on the uncertainty line from 0. At this step there is also a dotted line orthogonal to the uncertainty line. The place where the two dotted lines cross is the location of ω_x. Any opinion may be located using two out of the three lines for belief, disbelief and uncertainty.

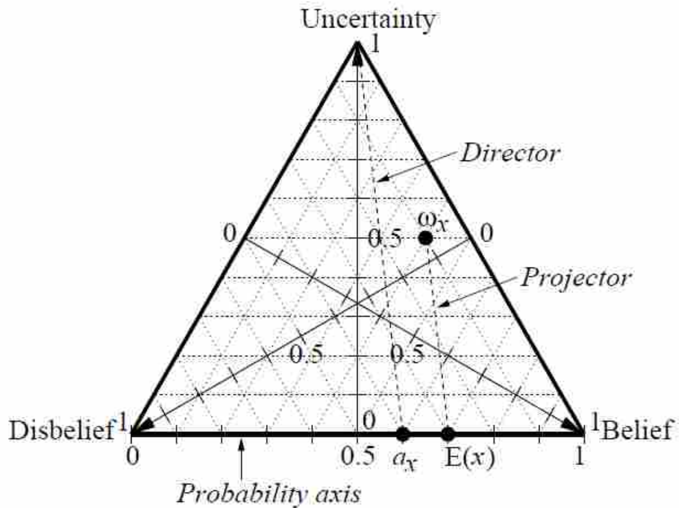


Figure 2.5: Opinion triangle with ω_x as an example [9].

2.5.2 Probability Expectation

When evaluating an opinion, probability expectation (E) is a useful value. This value gives the expected probability of an outcome based on the opinion and can be calculated using Eq. 2.18. It takes the entire belief value of an opinion into account and some of the uncertainty. Some uncertainty is taken into account because uncertainty is a measure of the unknowns in an outcome. Some of the unknowns may positively affect an outcome while some may negatively affect an outcome. Atomicity defines how much uncertainty should go to positively affecting an outcome.

$$E = b + au \tag{2.18}$$

In Figure 2.5 the horizontal base line is the probability axis that contains all possible probability expectation values from 0 to 1. Returning to our example opinion, ω_x, the probability expectation value for this opinion is 0.70. It can be found by following the director from the opinion location to where the director crosses the probability axis. The director is the line that extends from the top of the triangle to the location of atomicity on the probability axis. Atomicity may be found on the probability axis by traveling from left to right with 0 at the far left and 1 at the far right. When the common assumption $a = 0.50$ is made the director is always orthogonal to the probability axis and the probability expectation value is given in Eq. 2.19. In this research E is evaluated using this assumption.

$$E = b + \frac{1}{2}u \tag{2.19}$$

The probability expectation value gives the expected probability of a situation which is different than probability. Expected probability defines what an agent expects the probability to be and is not an exact measure of probability. For example, when a school class starts and most grades given in the class are B grades, the probability that student 1 will get a B grade is higher than probabilities that student 1 will get any other grade. What if student 1 has a history of getting good grades and is in a subject where he/she excels? This information does not change the probability that student 1 will get a B grade but it can change the expected probability. If it is input into subjective logic the expected probability that student 1 will get an A grade could be higher than the expected probability that student 1 will get a B grade. With information about a situation an agent can expect the probability of an outcome to be higher or lower than it otherwise would have.

2.6 Trust Networks

After selection and implementation of the feature extraction algorithms the intelligent agents needed to be designed to encapsulate the algorithms and combine the algorithms output into coherent feature sets. In this research the intelligent software agents are designed in the form of a trust network. Trust networks [29] are a way to quantify trust that is transferred from one individual to another. For example, Figure 2.6 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 inferential trust in individual C. In the agent architecture individuals are called 'agents' and the means by which trust is quantitatively transferred between agents is subjective logic.

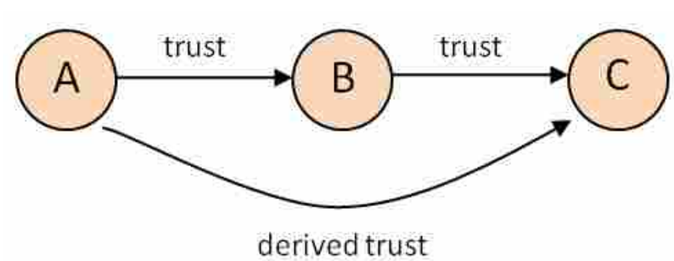


Figure 2.6: Simple trust network showing A's derived trust in C from B.

2.6.1 Discounting Operator

In a trust network there are two separate operators that transfer trust: the discounting operator and the consensus operator. The discounting operator is used when agents in a trust network lie along the same path as in Figure 2.6. The discounting operator is defined by Jøsang [8], and uses the symbol $\otimes$ giving

$$\omega_C^A = \omega_B^A \otimes \omega_C^B \tag{2.20}$$

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. The trust that A has in C from the discounting operator can be calculated using the following equations:

$$b_C^A = b_B^A b_C^B \tag{2.21}$$

$$d_C^A = b_B^A d_C^B \tag{2.22}$$

$$u_C^A = d_B^A + u_B^A + b_B^A u_C^B. \tag{2.23}$$

2.6.2 Consensus Operator

The consensus operator is used when one agent holds two opinions on the same agent, or piece of information, and they need to be combined into a single opinion. The consensus operator is defined by Jøsang [9], and uses the symbol $\oplus$ giving

$$\omega_Z^{XY} = \omega_Z^X \oplus \omega_Z^Y \tag{2.24}$$

where again the superscripts represent the agent having the trust and the subscripts represent the agent, or piece of information, on which the trust is based. To calculate the opinion ω_Z^{XY} using the consensus operator the following equations for belief, disbelief and uncertainty are used:

$$b_Z^{XY} = (b_Z^X u_Z^Y + b_Z^Y u_Z^X)/\kappa \tag{2.25}$$

$$for\ \kappa \neq 0 \qquad d_Z^{XY} = (d_Z^X u_Z^Y + d_Z^Y u_Z^X)/\kappa \tag{2.26}$$

$$u_Z^{XY} = (u_Z^X u_Z^Y)/\kappa \tag{2.27}$$

$$b_Z^{XY} = \frac{\gamma b_Z^X + b_Z^Y}{\gamma + 1} \tag{2.28}$$

$$for\ \kappa = 0 \qquad d_Z^{XY} = \frac{\gamma d_Z^X + d_Z^Y}{\gamma + 1} \tag{2.29}$$

$$u_Z^{XY} = 0 \tag{2.30}$$

where

$$\kappa = u_Z^X + u_Z^Y - u_Z^X u_Z^Y \tag{2.31}$$

and

$$\gamma = \frac{u_Z^Y}{u_Z^X}. \tag{2.32}$$

2.6.3 Example Trust Network

A trust network where the consensus and discounting operators would be needed is shown in Figure 2.7. Here A needs to form a final opinion on D (ω_D^A).

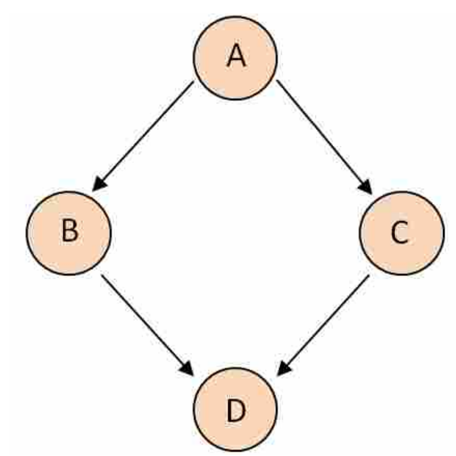


Figure 2.7: Graphical representation of a simple trust network requiring the consensus and discounting operators to calculate trust.

To form the final opinion the discounting operator is used once along each trust path giving

$$\omega_D^{A_B} = \omega_B^A \otimes \omega_D^B \tag{2.33}$$

$$\omega_D^{A_C} = \omega_C^A \otimes \omega_D^C \tag{2.34}$$

where the superscript notation A_B simply represents A's opinion based on B's opinion of D. The two new derived opinions may be combined using the consensus operator giving

$$\omega_D^A = \omega_D^{A_B} \oplus \omega_D^{A_C}, \tag{2.35}$$

or in its long form

$$\omega_D^A = (\omega_B^A \otimes \omega_D^B) \oplus (\omega_C^A \otimes \omega_D^C). \tag{2.36}$$

In a trust network each intelligent agent needs to form an opinion on other agents in the network and/or actual information being passed to the agent by the CFD simulation. When setting an agent opinion the entire belief tuple containing belief, disbelief and uncertainty needs to be given a value that follows Eq. 2.17. For example, in Eq. 2.36 the opinions ω_B^A, ω_D^B, ω_C^A and ω_D^C each need to have a separate belief, disbelief and uncertainty value before the final opinion, ω_D^A, can be found. After the opinions are set, the mathematics of the consensus and discounting operators can come into play to compute the final opinion.

Intelligent agents need their opinions defined beforehand in order to operate properly. The creation and evaluation of these opinions based on factors stemming from the actual simulation and factors specific to the extraction algorithms themselves allows the agents to make 'intelligent' decisions. For an example of information that might be used to set an agent's belief tuple think of a feature that is extracted early on in a steady simulation, perhaps at 100 iterations of a simulation that takes 10,000 iterations to run to full convergence. The disbelief and uncertainty of this feature will be higher than say a feature that has been extracted at 9,900 iterations into the same 10,000 iteration simulation. Setting agent opinions means taking information that is known to influence feature extraction in either a positive or negative manner and then computing a numerical belief, disbelief and uncertainty value for each agent belief tuple according to that information.

2.7 Massive Data Set Post-processing Concepts

The created intelligent agent structure is meant to be incorporated into the CAFÉ concept. In this section CAFÉ is defined as well as another similar massive data set post-processing concept called Evita.

2.7.1 CAFÉ

CAFÉ uses an agent-based structure that was designed for decision support in software applications and can be incorporated into some of the most popular CFD packages through the use of a convenient plug-in. CAFÉ uses multiple feature extraction algorithms to increase the accuracy of extracted features and machine learning to find characteristics that are of particular importance to a given researcher.

Figure 2.8 shows a conceptual view of the CAFÉ tool. Software running a physics-based simulation produces enormous amounts of data. The data is mined with various algorithms contained in agents. The feature extraction transforms the multi-variate data into reduced order information and is exchanged amongst the agents and with the operator. The transformed data is much easier to share, allowing the system to tune itself and guide the data-mining efforts. Since the agents communicate in information space, rather than the data space, the amount of bandwidth needed for the agents to interact is far less than needed for even a hierarchical data-mining scheme.

CAFÉ's capability to do concurrent analysis, i.e., during the simulation run time, can ameliorate excessive post-processing storage needs by targeting specific regions where features have been detected. Technologists recommend, and system developers redact specifications, that the computing system design has scalability as a requirement to anticipate the growth in data processing. System scalability might include growth margin in the number of processors, network bandwidth, type of distributed architecture (homogeneous, heterogeneous), programs, algorithms, and perhaps the programming languages with emphasis on memory management. Systems that are not easily extended are referred to as brittle, requiring a redesign or technology change. Software agents allow CAFÉ to be applied across massively parallel computing systems alongside state of the art CFD software programs taking full advantage of the parallel environment.

2.7.2 Evita

The closest program to CAFÉ is a concept designed specifically for large data set exploration called Evita [4]. Evita gives two paradigms for feature mining: point classification and aggregate classification. Point classification verifies points as features before they are aggregated while aggregate classification aggregates points before they are verified and then verifies the aggregate. It also gives an approach using wavelet transforms to eliminate unimportant features and locate areas of high interest to a researcher. Evita uses one feature extraction algorithm to create a binary classification of the flow domain that is then given to supplied data mining algorithms to

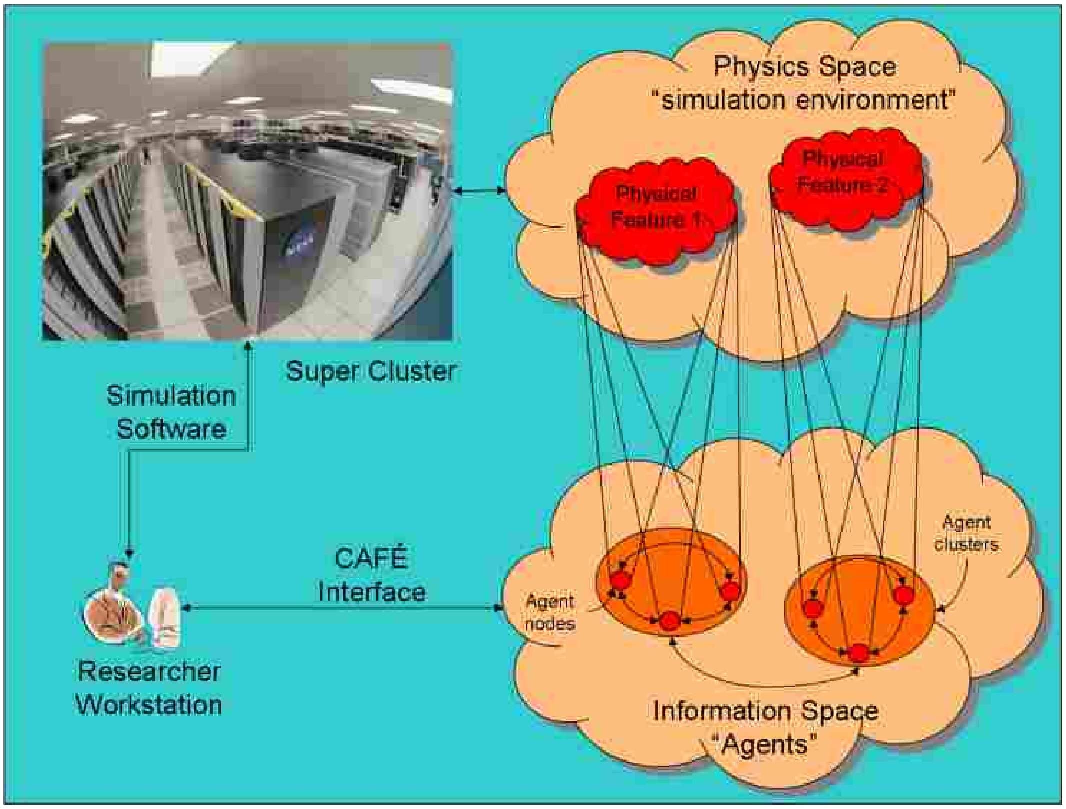


Figure 2.8: CAFÉ conceptual picture showing how the physics domain maps to nodes in the information space. These nodes communicate among each other, direct the data-mining activity, and interact with the operator.

classify, cluster, and categorize identified features before they are presented to a researcher. The computer components that comprise Evita are: an offline preprocessor, a server and a client.

CHAPTER 3. GENERAL FEATURE EXTRACTION METHOD

In this chapter a general method to extract flow features from CFD data sets using intelligent software agents governed by subjective logic is defined. In Chapter 4 the general extraction method is applied to fluid vortices. The general extraction method is defined as follows:

1. Extract features using feature extraction algorithms
2. Filter obviously extraneous features
3. Create agent opinions at regions contained in each extracted feature
4. Combine agent opinions to form final opinions of features
5. Aggregate one final feature set from all available feature sets

3.1 Extracting & Filtering Features

First, a CFD data set is run through feature extraction algorithms contained in intelligent software agents yielding data sets of reduced size containing only features called feature sets. If two extraction algorithms are in use then there will be two feature sets, one per algorithm. Each feature set produced is usually significantly different than other feature sets from the same data set. This can result in large variability between extracted features.

Usually, variability can be seen in features that have been extracted extraneously because each algorithm tends to extract different extraneous features. Of features that are extracted extraneously some are clearly extraneous from the start. Computation time can be saved if clearly extraneous features can be filtered out early using a simple threshold criterion. This criterion may be a common quantity such as pressure where any feature with an average pressure above the threshold value is kept and the remaining features are filtered out. The filtering threshold may be set low to filter out clearly extraneous features only and let most features through since agents are better suited to filter out features that are not clearly extraneous.

3.2 Forming Opinions on Extracted Features

Once features have been extracted and sent through a simple filter 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. 2.17. Agents form their opinions based on a pre-defined set of information known to influence the extraction of features.

When all agent opinions have been formed, a final governing opinion may be formed using the discounting and consensus operators defined in Sections 2.6.1 and 2.6.2 respectively. The final opinion consists of three values: belief, disbelief, and uncertainty. It is these values that give an estimate of the expected probability of a feature. If a feature has a high expected probability it will have a high belief, low disbelief and low uncertainty. If a feature has a low expected probability it will have a high disbelief and/or high uncertainty with a low belief. Recall that expected probability is computed from an opinion using Eq. 2.19.

There is no exact measure of when a feature is correct or when a feature is incorrect. For example, if a feature has a belief of 0.80, a disbelief of 0.10 and an uncertainty of 0.10 is the feature correct? The answer is, it depends. Subjective logic is a logic that deals in subjective beliefs where there is no clear cut definition of correct and incorrect. Instead of finding if a feature is correct it is found if a feature has a high expected probability. With the opinion $\omega = (0.80,\ 0.10,\ 0.10,)$ there is an expected probability of 85% which indicates that the feature is probable. It will be shown in Section 5.1.3 that while we are dealing in relative correctness it is fairly straightforward to see which features are the most probable.

3.2.1 Agent Structure

The graphical representation of the agent structure used to form opinions on the existence of features is shown in Figure 3.1. AA is the algorithm agent which contains actual feature extraction algorithms with subscripts 1 and 2 denoting encapsulation of separate algorithms. MA is the master agent which combines information from multiple AAs to form its opinion. R refers to a region in the computational domain that is under inspection by the intelligent agents to find whether or not the feature is probable. For line-type features such as vortex core lines, separation lines and attachment lines, R is a grid point contained in the extracted line. For features such as shock waves, R can be a 2D or 3D region contained in the extracted shock. The end goal is for the MA to form an opinion on the R meaning that the MA will have some belief, disbelief, and uncertainty about the feature that contains the R.

Each AA forms its own opinion on the R denoted by $\omega_{\mathrm{R}}^{\mathrm{AA}_1}$ and $\omega_{\mathrm{R}}^{\mathrm{AA}_2}$. This notation gives the agent forming the opinion as the superscript and the region the opinion is formed on as the subscript. The MA forms an opinion on each AA in use given by $\omega_{\mathrm{AA}_1}^{\mathrm{MA}}$ and $\omega_{\mathrm{AA}_2}^{\mathrm{MA}}$. Once the initial opinions are formed they can be combined into a final opinion, $\omega_{\mathrm{R}}^{\mathrm{MA}}$, on the existence of a feature in the R. Eq. 3.1 uses the consensus and discounting operators to give the final opinion and Eqs. 3.2–3.4 give the belief tuple values in the final opinion for the common condition $\kappa \neq 0$.

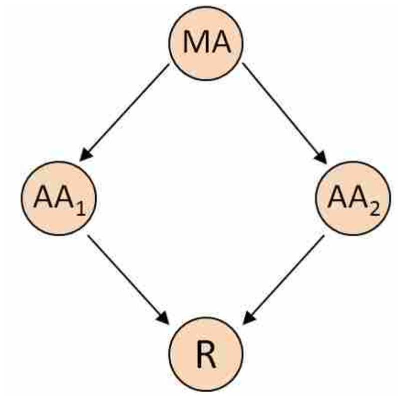


Figure 3.1: Graphical representation of two algorithm agent structure.

$$\omega_{\mathrm{R}}^{\mathrm{MA}} = \left(\omega_{\mathrm{AA}_1}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA}_1}\right) \oplus \left(\omega_{\mathrm{AA}_2}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA}_2}\right) \tag{3.1}$$

$$b_{\mathrm{R}}^{\mathrm{MA}} = \frac{(b_{\mathrm{AA}_1}^{\mathrm{MA}} b_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) + (b_{\mathrm{AA}_2}^{\mathrm{MA}} b_{\mathrm{R}}^{\mathrm{AA}_2})(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})}{\kappa} \tag{3.2}$$

$$d_{\mathrm{R}}^{\mathrm{MA}} = \frac{(b_{\mathrm{AA}_1}^{\mathrm{MA}} d_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) + (b_{\mathrm{AA}_2}^{\mathrm{MA}} d_{\mathrm{R}}^{\mathrm{AA}_2})(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})}{\kappa} \tag{3.3}$$

$$u_{\mathrm{R}}^{\mathrm{MA}} = \frac{(d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2})}{\kappa} \tag{3.4}$$

where

$$\begin{aligned} \kappa \;=\; & (d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1}) + (d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}) \\ & - (d_{\mathrm{AA}_1}^{\mathrm{MA}} + u_{\mathrm{AA}_1}^{\mathrm{MA}} + b_{\mathrm{AA}_1}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_1})(d_{\mathrm{AA}_2}^{\mathrm{MA}} + u_{\mathrm{AA}_2}^{\mathrm{MA}} + b_{\mathrm{AA}_2}^{\mathrm{MA}} u_{\mathrm{R}}^{\mathrm{AA}_2}). \end{aligned} \tag{3.5}$$

While Figure 3.1 displays two AAs any number of AAs may be incorporated into the agent structure allowing the use of any number of extraction algorithms. Figure 3.2 shows how each algorithm plays a role in only one of the transitive trust paths allowing a modular handling of multiple algorithms. A transitive trust path can be visualized as any one path from the MA to the R. Any algorithm's path may be added or removed from the trust network without affecting other branches of the network. This allows the agent structure to easily handle new and updated extraction algorithms. For example, if a new separation and attachment line extraction algorithm is defined it can be encapsulated in an agent and easily inserted into the agent structure without requiring a large change in the previous agent structure. For multiple AAs, N may be increased to account for all included algorithm agents, or N may be decreased to 1 when only a single AA is used. Eq. 3.6 uses the consensus and discounting operators to give the final opinion as a combination of all opinions for any number of AAs. With an increased number of algorithms there are more feature sets allowing the agents to search through an increased amount of features giving more information on what features are probable and what are not. Also, with added algorithms features that were not previously extracted could possibly be extracted. An agent cannot select a feature if it is not in one of the available feature sets.

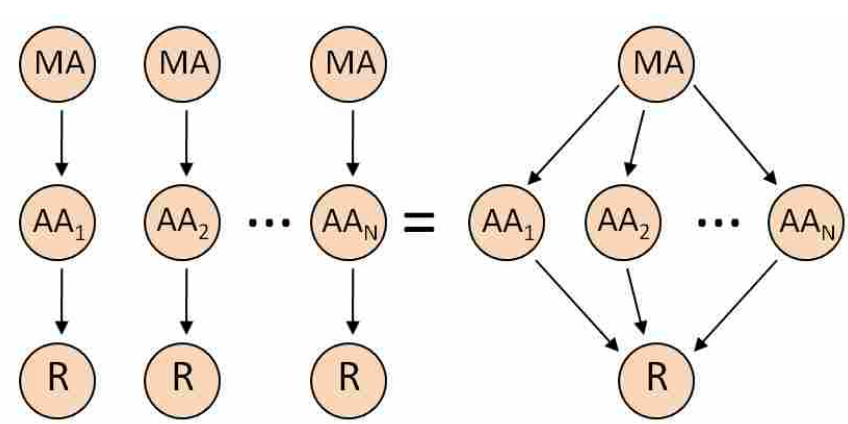


Figure 3.2: Graphical representation of modular agent structure.

$$\omega_{\mathrm{R}}^{\mathrm{MA}} = \left(\omega_{\mathrm{AA_1}}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA_1}}\right) \oplus \left(\omega_{\mathrm{AA_2}}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA_2}}\right) \oplus \cdots \oplus \left(\omega_{\mathrm{AA_N}}^{\mathrm{MA}} \otimes \omega_{\mathrm{R}}^{\mathrm{AA_N}}\right) \tag{3.6}$$

3.2.2 Algorithm Agent Opinions

The first agent opinions to set are the AA, or algorithm agent, opinions. Recall that in a two AA structure there are two feature extraction algorithms that output two separate feature sets. This case can be seen in Figure 3.1 and Eq. 3.1. It is important to think of each feature set as separate, remembering that each feature set has been extracted by a different extraction algorithm and thus by a different AA. Consider Figure 3.3 containing two hypothetical separate simple line-type feature sets produced by AA_1 (black) and AA_2 (blue). The black line comprises feature set 1 and the blue line comprises feature set 2. While these line-type feature sets are displayed together they are separate sets.

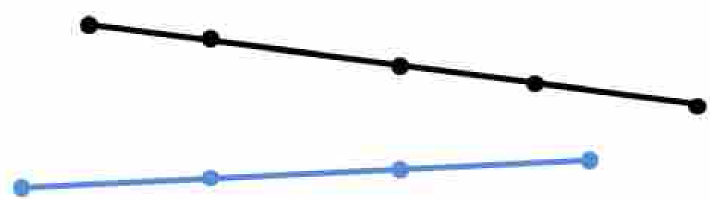


Figure 3.3: Two separate simple sets of line-type features hypothetically extracted by AA_1 (black) and AA_2 (blue).

With these two feature sets in mind the opinions of AA_1 and AA_2 for feature set 1 may be defined. AA_1 needs to form an opinion at each point contained in each line in feature set 1. Also, AA_2 needs to form an opinion at each point contained in each line in feature set 1. Why does AA_2 need to form an opinion on feature set 1 even if it does not extract the features, or the exact points, contained in the feature? This follows from Figure 3.1 and the resulting Eq. 3.1. If AA_2 does not form an opinion at each point, or each R, then the left-hand-side of Eq. 3.1 cannot be evaluated for there will be no values in $\omega_{\mathrm{R}}^{\mathrm{AA}_2}$. Both AA_1 and AA_2 need to form an opinion at each point in feature set 1 leading to a dichotomy for defining the algorithm agents. AA_1 extracts the features in feature set 1 so it is termed the extracting algorithm agent (AA_E). AA_2 does not extract the features in feature set 1 so it is termed the non-extracting algorithm agent (AA_{NE}).

After opinions are defined for feature set 1, AA_1 and AA_2 need to define their opinions for feature set 2. Recall that feature set 2 is extracted by the feature extraction algorithm contained in AA_2. This changes how opinions are set from feature set 1. In feature set 1, AA_1 was the extracting algorithm agent and AA_2 was the non-extracting algorithm agent. Now the roles are reversed for feature set 2. AA_2 extracts the features in feature set 2 so it is AA_E, and AA_1 is AA_{NE}. Once each feature set has had opinions defined for AA_1 and AA_2 there are no more AA opinions to define.

This dichotomy between extracting and non-extracting algorithm agent opinions works just as well with multiple algorithms contained in the trust network as shown in Figure 3.2 and Eq. 3.6. At each created feature set there will be one AA_E and the rest of the algorithm agents will be non-extracting algorithm agents. With this dichotomy in place it is now possible to define the AA_E and AA_{NE} opinions.

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 information is shown in Table 3.1. Recall that the three belief tuple values must conform to Eq. 2.17.

Table 3.1: How to set AA_E opinion.

AA_E	Set by
b	Strengths
d	Weaknesses
u	Feature characteristics

Belief set by algorithm strengths means that each strength is given to the agent and a belief is set based on whether or not the extracted region has the strength characteristics. For example, work done by Roth [7] showed that the SH algorithm adequately extracts vortex core lines when they are close to straight and when the vortex has high strength. These two strength characteristics are given to AA_E and a high belief of one is set if the region has both characteristics, a low value near zero is set if the region has neither characteristic and any value in between the high and low values may be given for all other cases.

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. Continuing the example used for belief, the SH algorithm does not work well for curved vortices or vortices with a low strength. So if a vortex has both of these weakness characteristics the disbelief will be set high, if neither characteristic is present than the disbelief value is zero and for other cases a disbelief value may be set between the high and low values.

Uncertainty is set from scientifically known characteristics of the flow feature. For example, it is not common for a shock wave to form in empty space but rather there is usually a physical boundary near the shock. For external flows there is commonly a physical body that the flow adjusts to forming the shock. For internal flows, such as flow in a nozzle, a shock wave will be near to the nozzle walls. A criterion that a shock should be within a certain distance from a physical boundary can be given to the agent. If a shock forms up against a physical boundary the uncertainty will be zero, if the shock forms away from a boundary the uncertainty will be high and other uncertainty values are given for situations in between. There are many types of flow feature characteristics that can be input to the agents. The only criterion that must be met is that this information is quantifiable. It is not possible to input information that is not quantifiable. This criterion also holds for information used to set the belief and disbelief.

The main strength of setting the AA_E opinion values in this manner is that this template can easily be adapted to any feature with corresponding feature extraction algorithms. For example, if a feature has three feature extraction algorithms then as long as the strengths of each algorithm, the weaknesses of each algorithm, and some information about the physical formation of the feature are known they can be added to agents allowing them to make decisions about the expected probability of extracted features.

Non-extracting Algorithm Agent Opinion

After defining the AA_E's opinion a definition can be given for the AA_{NE}'s opinion. The belief tuple set for the 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 the distance from the closest extracted region. For the AA_{NE} the belief and disbelief values are set from the AA_E strengths and weaknesses.

The uncertainty is set according to the minimum distance between any region extracted by the AA_{NE} and the region under consideration. For example, if there are two feature sets and the region under inspection is contained in feature set 1 then the minimum distance would be measured between that region and the closest region contained in feature set 2. The idea is when the AA_{NE} extracts a region close to the AA_E, the AA_{NE} is more certain about the region so its uncertainty is near zero. When the AA_{NE} does not extract a region close to the region under inspection, it is uncertain about the AA_E's extracted region meaning that its uncertainty will be high.

Table 3.2: How to set AA_{NE} opinion.

AA_{NE}	Set by
b	AA_E Strengths
d	AA_E Weaknesses
u	Minimum distance from AA_{NE} extracted point

3.2.3 Master Agent 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 AAs and provide a final decision on the extracted features. The MA's belief tuple is based on the idea that as a simulation converges to a final solution, so too will a feature converge from some beginning spatial location to a final location. This is implemented through a displacement measure called feature displacement (FD). Feature displacement is a measure of the displacement, or movement, of a region between any number of iterations. FD is divided by a reference length which nondimensionalizes the FD making it easier to work with across separate simulations with large variations in length scales. For line-type features the reference length is the line length. Eq. 3.7 gives the FD when the region is a point. Here the subscript i refers to the iteration under investigation and $i-1$ refers to the previous iteration where features were extracted which could be hundreds or thousands of iterations prior.

$$FD_i = \frac{|P_i - P_{i-1}|}{\ell_i} \tag{3.7}$$

Figure 3.4 gives an example of feature displacement between two line-type features. One of the lines is extracted at two hundred iterations with the other extracted at three hundred iterations. If a similar point is taken from each line the feature displacement at that point is defined as the magnitude of the distance between the two points divided by the length of the line at three hundred iterations. Each point contained in the three hundred iteration core line has a feature displacement based on the two hundred iteration core line.

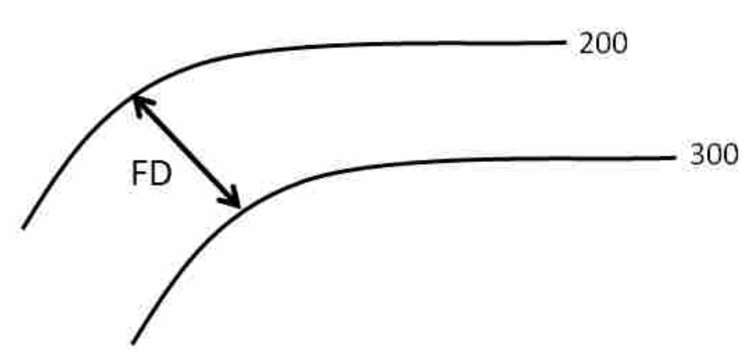


Figure 3.4: Two line-type features extracted at 200 and 300 iterations show that feature displacement is found between a similar point on each line.

Another quantity used to define the MA's opinion is the change in feature displacement (ΔFD). This value corresponds to the absolute value of the slope of a feature displacement vs. iterations plot. Change in feature displacement is defined as:

$$\Delta FD_i = \frac{|FD_i - FD_{i-1}|}{\#\, of\ iterations}. \tag{3.8}$$

With feature displacement and change in feature displacement defined, the belief tuple for the MA is specified as follows: belief is set by feature displacement and change in feature displacement, disbelief is set by feature displacement and uncertainty is set by change in feature displacement. This information is shown in Table 3.3.

The belief value will be high, or close to one, when the feature displacement is small as well as the change in feature displacement. The belief value will be low, or close to zero, when the feature displacement and the change in feature displacement is high. This says that the MA believes a feature when the feature has moved only a small amount between iterations under investigation and has a trend that suggests the feature will not move substantially with more iterations. The disbelief value will be low when feature displacement is low and high when feature displacement is high. This says that the MA disbelieves a feature if there is a large amount of feature displacement or does not disbelieve a feature if the feature displacement is small. The uncertainty value is high when change in feature displacement is high and low when change in feature displacement is low. This says that the MA is uncertain about the feature if the feature could move substantially with more iterations.

Table 3.3: The MA opinion values set for any feature extraction algorithm.

MA Belief Tuple	
b	FD & ΔFD
d	FD
u	ΔFD

3.3 Aggregating Final Feature Set

After regions contained in features are given final opinions by the intelligent agents, feature sets may be combined into one final feature set. This is done by finding which features have high expected probabilities and which features do not. The features with higher expected probabilities are selected while the features with lower expected probabilities are discarded. When feature extraction algorithms extract features that are the same feature then the feature with the highest expected probability is selected and the other feature is discarded.

Currently, this process is not automated but rather done by visual interpretation. It can be automated by implementing a search criterion that locates the same feature in feature sets created by separate extraction algorithms and then compares those features and selects the feature with the highest expected probability. The search criterion could be a simple distance threshold where if a feature in a separate feature set lies within some spatial bounding box then it is the same feature. For those features that do not have the same feature extracted by a separate algorithm then they may be selected or discarded based on some combination of their belief tuple values and their probability expectation.

CHAPTER 4. VORTEX CORE EXTRACTION METHOD

While the general feature extraction method using intelligent software agents defined in Chapter 3 works for any feature, this chapter applies the method to vortex core line extraction.

4.1 Extracting and Filtering Vortex Cores

Two vortex core line extraction algorithms are used: RP and SH. These two algorithms were described in Section 2.3. From these algorithms two feature sets are created. Each feature set is a collection of points connected into lines. Some lines have been extracted as vortex cores but are clearly extraneous lines. Figure 4.1 gives an example of a data set that has had vortex core lines extracted where some lines are clearly extraneous. They are clearly extraneous because the flow is moving from the bottom right corner to the top left corner and the inlet boundary condition is steady, uniform flow with low freestream turbulence. Vortices are not likely to form in these conditions upstream of a delta wing. Three filters are used to filter extraneous cores from vortex core line feature sets: point count, strength, and quality.

4.1.1 Point Count Filter

The point count filter removes lines that have fewer points than a specified minimum value. This filter removes cores that have low point counts because they are highly unlikely to be coherent vortex structures. This is true especially in Reynolds-averaged Navier-Stokes (RANS) simulations that do not resolve turbulent eddies but rather model turbulence. This research applies to RANS

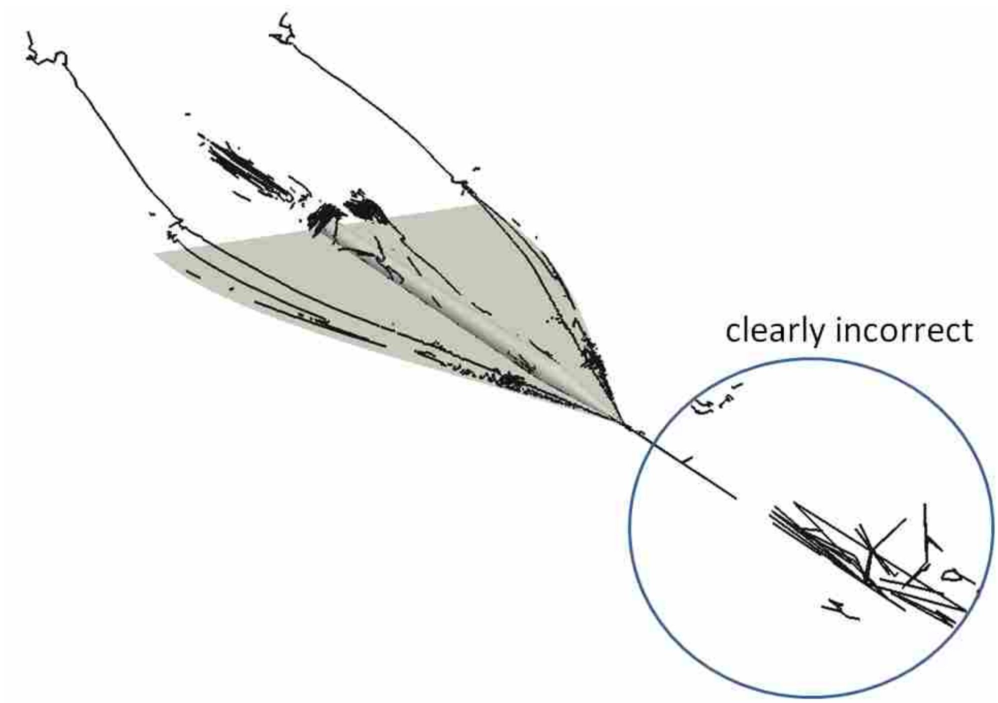


Figure 4.1: Unfiltered delta wing data set showing vortex cores with some cores clearly extraneous.

simulations only so the intelligent agents have not extracted features from CFD data sets that are not RANS simulations. The behavior of the point count filter may change if a simulation is a Large Eddy Simulation (LES) where some turbulent eddies are resolved or a Direct Numerical Simulation (DNS) where all turbulent eddies at or above the Kolmogorov scale are resolved.

A suitable point count minimum threshold value depends upon the density of the grid used to compute the CFD solution. There is a common trend that as the grid density increases the minimum threshold value increases and as the grid density decreases the minimum threshold value decreases. For the delta wing data set explained in Section 5.2 with a relatively high grid density the minimum point count value is 5. This value could be set as high as 10, but it is better not to filter out possible cores before they get to the agents. For the blunt fin data set explained in Section 5.1 with a relatively low grid density the minimum point count value is 2. While the minimum point count values were set constant in these two simulations they may be changed by a user as the user sees fit. If the proper value for the point count filter is not known it is best to set it low as it may filter out possible cores before they get to the agents.

4.1.2 Strength Filter

This filter removes points contained in vortex core lines that have a strength value below the minimum strength value where vortex strength is defined in Section 2.4.2. This is done because strength is synonymous with the swirling motion around the core and a highly probable vortex will have high swirling motion around the core.

While it would be nice to have a value for vortex strength that could be applied across many data sets this is not the case. Vortex strength relies heavily on the local flow velocity which can have a large variation between data sets. There is a common trend that as the velocity increases the minimum vortex strength increases and as the velocity decreases the minimum vortex strength decreases. For the delta wing data set that has a freestream Mach of 0.3, the minimum value for vortex strength is set at 50.

One thing to keep in mind is that for numerical stability some codes will divide the velocity, pressure/density, and temperature values by a reference value. This makes these values close to one giving the code more numerical stability. If this is the case, then the vortex strength value must be set very low because velocities are close to 1. This happens in many old plot3D data sets that are used to validate feature extraction codes [30].

In this research the vortex strength filter is implemented before the two feature extraction algorithms are complete. In the SH and RP algorithms it is found if the conditions of Eq. 2.6 and Eq. 2.14 hold and the result is a set of points that must be aggregated into lines. Before the lines are aggregated the points with a vortex strength below the minimum value are removed.

4.1.3 Quality Filter

The quality filter filters out any vortex core line that has an average quality above the maximum quality value. Recall that quality is defined in Section 2.4.1 as the angle between a vortex core line and its associated velocity vector. Since quality is a pointwise variable the average quality is computed from all points contained in the vortex core line under consideration. Unlike vortex strength and point count, the quality filter does have a threshold value that is constant across data sets. Roth [7] gives a suggested maximum quality threshold between 30° and 45°. A maximum quality value of 35° is used in this research which is sufficient to filter out many extraneous cores.

4.2 Forming Opinions on Extracted Vortex Cores

After the two feature sets have been filtered, four opinions need to be formed at each point contained in each line of both feature sets: $\omega_{\mathrm{AA}_1}^{\mathrm{MA}}$, $\omega_{\mathrm{AA}_2}^{\mathrm{MA}}$, $\omega_{\mathrm{R}}^{\mathrm{AA}_1}$, and $\omega_{\mathrm{R}}^{\mathrm{AA}_2}$. Here R refers to a point contained in a vortex core line and AA_1 and AA_2 refer to the algorithm agents containing the SH and RP algorithms respectively. In practice only three of the four opinions must actually be formed as the opinions $\omega_{\mathrm{AA}_1}^{\mathrm{MA}}$ and $\omega_{\mathrm{AA}_2}^{\mathrm{MA}}$ are the same. Recall from Section 3.2.3 that the information used to set the master agent opinion does not depend on the feature extraction algorithm in use. The master agent opinion depends on the feature displacement and the change in feature displacement which are measures of feature movement through a number of simulation iterations. This allows the opinions to be the same without any loss of information.

To define the algorithm agent opinions the information in Table 3.1 needs to be defined and quantified. Roth [7] gives an excellent comparison of both the SH and RP vortex extraction algorithms along with their respective strengths and weaknesses. The strengths and weaknesses information used here is taken from his work.

4.2.1 Sujudi-Haimes Strengths, Weaknesses and Feature Characteristics

Table 4.1 gives the strengths, weaknesses and feature characteristics used for the SH vortex core extraction algorithm. The SH algorithm is specifically designed to extract straight vortex cores which is why a straight core is part of its strengths. The SH algorithm also works well when there is a strong rotational velocity around the core. This situation is quantified by the vortex strength so a high vortex strength is added as an algorithm strength. Quality is independent of extraction algorithm but is used in the belief value because vortices with a low quality correspond with a probable vortex core line.

Table 4.1: $\mathrm{AA_E}$ opinion values set for the SH vortex core extraction algorithm.

$\mathrm{AA_E}$	Set by	Sujudi-Haimes
b	Strengths	straight core, high strength, low quality
d	Weaknesses	curved core, low strength, high quality
u	Feature characteristics	distance from possible trip point

The weakness characteristics for the SH algorithm are the exact opposite of the strength characteristics. Curved core, low strength, and high quality are all characteristics that negatively affect the correct extraction of vortex core lines.

While there are many possible feature characteristics the only feature characteristic used in this research is the distance from a possible vortex trip point. Other feature characteristics include the 2π criterion where a core line must contain a streamline that rotates at least one revolution around the core and a low pressure at the vortex core when compared to the vortical flow further away from the core. Future research could implement other vortex feature characteristics into the extracting algorithm agent opinion $\omega_{\mathrm{R}}^{\mathrm{AA_E}}$.

4.2.2 Belief Tuple Values for Sujudi-Haimes Extracting Algorithm Agent

For each value in AA_E's belief tuple when SH is the feature extraction algorithm the information from Table 4.1 is used to set the value. This information is quantified and input into a linear function which sets each belief tuple value. The linear functions are shown in Eq.'s 4.1–4.3.

$$b = 0.4 \cdot NormalAverage + 0.6 \tag{4.1}$$

$$d = -0.4 \cdot NormalAverage + 0.4 \tag{4.2}$$

$$u = 0.5 \cdot DistanceFromVortexTripPoint \tag{4.3}$$

where

$$NormalAverage = \frac{NormalVortexStrength + NormalCurvature + NormalQuality}{3} \tag{4.4}$$

and

$$NormalVortexStrength = \begin{cases} \left|\frac{VortexStrength}{VortexStrengthMax}\right|, & |VortexStrength| < VortexStrengthMax \\ 1, & |VortexStrength| \geq VortexStrengthMax \end{cases} \tag{4.5}$$

$$NormalCurvature = \begin{cases} \left|\frac{Curvature}{CurvatureMax} - 1\right|, & Curvature < CurvatureMax \\ 0, & Curvature \geq CurvatureMax \end{cases} \tag{4.6}$$

$$NormalQuality = \begin{cases} \left|\frac{Quality}{QualityMax} - 1\right|, & Quality < QualityMax \\ 0, & Quality \geq QualityMax. \end{cases} \tag{4.7}$$

Each of the three strength values has a corresponding maximum value: $VortexStrengthMax$, $CurvatureMax$, and $QualityMax$. These maximum values allow each strength value to be put on a scale from zero to one. Zero meaning that the algorithm is operating away from its strengths and one meaning that the algorithm is operating at its strongest point. When all three values are put on the same scale they may be averaged and combined into a single value called the $NormalAverage$ which is then used in Eqs. 4.1 and 4.2. This is useful because it allows Eqs. 4.1 and 4.2 to be used regardless of how many strength values are defined.

Eqs. 4.1, 4.2, and 4.3 were chosen to accurately represent the belief, disbelief, and uncertainty a researcher familiar with the SH algorithm would have when it extracted vortex cores in given situations. They are not set in stone and may be changed if it is found that different values more accurately reflect a researcher's opinion. When $NormalAverage = 1$, corresponding to a situation where SH extracts a feature where all of its strength characteristics are present, then $b = 1$ and $d = 0$. This makes intuitive sense because the algorithm should be believed when it extracts features where all its strengths are present. When $NormalAverage = 0$ corresponding to SH extracting features where none of its strengths are present then $b = 0.6$ and $d = 0.4$. Why then is $b \neq 0$? Belief is not equal to zero here because SH did extract the core line. If SH does detect a feature there must be some belief that it is operating correctly. Now let's look at a situation between these two extremes. What if $NormalAverage = 0.5$ which corresponds to the SH algorithm extracting cores where some of its strengths are present and some are not. This gives $b = 0.8$ and $d = 0.2$ which says that there is a significant amount of belief that the SH algorithm extracted the core properly, but it might not have.

While it is nice to have $b + d = 1$ the condition for the belief tuple is $b + d + u = 1$ given in Eq. 2.17. So clearly when $u > 0$ then $b + d + u > 1$. When $u > 0$ belief is held constant and disbelief and uncertainty are equally decreased until the condition holds. In Eq. 4.3 the uncertainty is defined only by $DistanceFromVortexTripPoint$ and if $DistanceFromVortexTripPoint = 0$ then $u = 0$. As it increases the uncertainty will increase proportionately. So $u > 0$ will be all cases except when a vortex core starts exactly on a solid boundary.

From Eq. 4.5 it can be seen that condition two gives a $NormalVortexStrength = 1$ which corresponds to SH extracting a vortex with a high rotational velocity around the core which is one of its strengths. If $VortexStrength = 0$, then $NormalVortexStrength = 0$ meaning that there is no rotational velocity around the core so it is obviously extraneous.

Eq. 4.6 allows the $NormalCurvature$ value to be one when there is zero curvature as SH was designed to extract cores with zero curvature. The $CurvatureMax$ value may be set according to the definition of curvature used. In this research Eq. 2.16 is used to calculate curvature and $CurvatureMax = 0.30$ is an acceptable value for this definition.

Eq. 4.7 is used to calculate $NormalQuality$ and it is set similar to $NormalCurvature$ because a small quality value leads to a more probable extraction of a vortex core. $QualityMax = 50^\circ$ is used as the maximum value. Recall from Section 4.1.3 that our minimum quality filter value is 35°. Then why isn't $QualityMax = 35^\circ$? It is 50° because the quality filter filters out lines, not points, that have an average value above 35°. While most points contained in the lines will have a quality value below 35° there are still quality values at points that may be 50° or higher.

One consequence of Eqs. 4.1 and 4.2 is with $NormalAverage = 1$ which corresponds to SH operating at an optimal value for all of its strengths then $b_{\mathrm{R}}^{\mathrm{AA}_1} = 1$ and $d_{\mathrm{R}}^{\mathrm{AA}_1} = 0$. This says that $\mathrm{AA_E}$ has a complete belief in the extracted feature and no disbelief. This is appropriate because the algorithm should be believed when it is operating at optimal conditions. When $NormalAverage = 0$ this corresponds to the condition that SH is operating at all of its weaknesses. This sets $b_{\mathrm{R}}^{\mathrm{AA}_1} = 0$

and $d_{\mathrm{R}}^{\mathrm{AA}_1} = 1$ saying that $\mathrm{AA_E}$ has no belief in the extract feature and has total disbelief. This is an appropriate opinion because the feature extraction algorithm should not be believed when it is operating in areas for which it has not been designed.

4.2.3 Roth-Peikert Strengths, Weaknesses and Feature Characteristics

Table 4.2 gives the strengths, weaknesses and feature characteristics used for the RP vortex core extraction algorithm. The RP algorithm was designed specifically to extract curved vortex core lines as outlined in Section 2.3.2 so curved core is added to the algorithm's strengths. The RP algorithm also works well with core lines that have a low rotational velocity around the core, or low vortex strength. It is not that the RP algorithm does not extract correctly vortices with a high rotational strength because it does. It just also works well with vortices that have a low strength. Once again quality is used as a strength even though it is algorithm independent.

Table 4.2: $\mathrm{AA_E}$ opinion values set for the RP vortex core extraction algorithm.

$\mathrm{AA_E}$	Set by	Roth-Peikert
b	Strengths	curved core, low strength, low quality
d	Weaknesses	straight core, near zero strength, high quality
u	Feature characteristics	distance from possible trip point

Two of the weakness characteristics for the RP algorithm are the opposite of the strength characteristics: straight core and high quality. Setting a straight core as a weakness characteristic might be misleading because the RP algorithm does not extraneously extract straight vortex core lines. A straight core is a weakness characteristic because when it comes to straight core lines there is more belief that the SH algorithm will extract them correctly than the RP algorithm. Using the SH and the RP algorithms together in this fashion helps us to match each algorithms strengths with the flow situations for which they were designed. The last weakness for the RP algorithm is a near zero strength. This simply means that there is some minimum threshold on vortex strength for which the RP algorithm correctly extracts vortices.

The feature characteristic used for the RP algorithm is the same as the feature characteristic used for the SH algorithm which is distance from a possible vortex trip point. When using multiple feature extraction algorithms the same feature characteristics are used for all algorithms since feature characteristics are not algorithm dependent.

4.2.4 Belief Tuple Values for Roth-Peikert Extracting Algorithm Agent

For each value in AA_E's belief tuple when RP is the feature extraction algorithm the information from Table 4.2 is used to set the value. This information is quantified and input into a linear function which sets each belief tuple value. The linear functions are shown in Eq.'s 4.8–4.10.

$$b = 0.4 \cdot NormalAverage + 0.6 \tag{4.8}$$

$$d = -0.4 \cdot NormalAverage + 0.4 \tag{4.9}$$

$$u = 0.5 \cdot DistanceFromVortexTripPoint \tag{4.10}$$

where

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

and

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

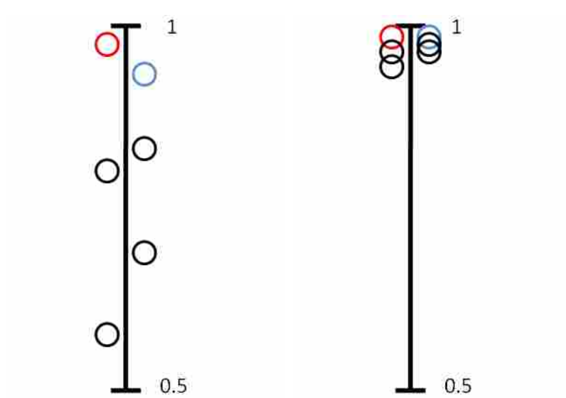
$$NormalCurvature = \begin{cases} \left|\frac{Curvature}{CurvatureMax}\right|, & Curvature < CurvatureMax \\ 1, & Curvature \geq CurvatureMax \end{cases} \tag{4.13}$$

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

In Eq. 4.13 the definition of *NormalCurvature* is different for RP than SH. This is because RP is defined to extract curved cores so the *NormalCurvature* value should be one when $Curvature \geq CurvatureMax$. The definition for *NormalQuality* is the same for RP as SH. There is a slight difference between Eqs. 4.5 and 4.12 found in the *NormalVortexStrength* value. For SH *VortexStrengthMax* is set higher than RP. This is done because RP works better than SH for cores that have a low strength, but a zero strength will still correspond to an extraneous vortex.

While it might seem that maximum values such as *VortexStrengthMax* and the constants from Eqs. 4.8–4.10 must be set exactly they do not. These values are applied equally to all feature sets serving to lower or higher each probability expectation value given in the final opinion. One problem when setting these values is that if the *VortexStrengthMax* value is set too low and if the *QualityMax* value is set too low than the opinions bunch around a belief of one and a probability expectation of one. Also, if the constant in Eq. 4.8 is changed from 0.6 to 0.8 then the belief values will bunch around one.

Figure 4.2 gives a graphical representation of vortex core opinions with good and poor spacing. A circle represents a vortex core opinion and the scale of the figures may represent the belief value of each opinion or the probability expectation value for the entire opinion. It is important to have well spaced opinions so decisions about the most probable cores are simpler. In Figure 4.2a the red circle is clearly the vortex core with the highest belief or probability expectation and the blue circle is below it with the second highest value. In Figure 4.2b this behavior is not as easily distinguished. It looks as if the red circle still has the highest value but it is not as clear. Also, are all the vortex cores bunched around one probable or just the red and blue circles as in Figure 4.2a? It is hard to tell. Maximum values and constants need to be defined such that there is good spacing of the belief and expected probability. With a good spacing it is much simpler to decide which vortex cores are more probable. This behavior is shown in Section 5.1.3 and Figure 5.6 where there is a comparison of belief, disbelief, uncertainty, and probability expectation between the same extracted core by the SH and RP algorithms.



(a) (b)

Figure 4.2: (a) Vortex core opinions with good spacing. (b) Vortex core opinions with poor spacing. The scale on either figure may represent belief or expected probability.

The maximum value that plays the most importance is *CurvatureMax* because essentially it is selecting between the two algorithms. There is a trend when setting the *CurvatureMax* value that when it is set too high there is a bias to believe all cores extracted by RP and disbelieve all cores extracted by SH. When *CurvatureMax* is set too close to zero then the bias shifts to believing all cores extracted by SH and disbelieving all cores extracted by RP. $CurvatureMax = 0.3$ has been found to give a good estimate of belief and disbelief to vortex cores extracted by the SH and RP algorithms. This value corresponds to using the curvature definition of Eq. 2.16.

4.2.5 Non-extracting Algorithm Agent Opinion

For each value in $\mathrm{AA_{NE}}$'s belief tuple the information from Table 3.2 is used to set the value. This information is quantified and input into a linear function which sets each belief tuple value. The linear functions are shown in Eq.'s 4.15–4.17.

$$b = 0.8 \cdot NormalAverage + 0.2 \tag{4.15}$$

$$d = -0.8 \cdot NormalAverage + 0.8 \tag{4.16}$$

$$u = 0.5 \cdot NormalMinimumDistance \tag{4.17}$$

where *NormalAverage* is computed from Eqs. 4.4 and 4.11 based on what algorithm extracted the vortex cores and

$$NormalMinimumDistance = \left| \frac{MinimumDistance}{MinimumDistanceMax} \right|. \tag{4.18}$$

The *NormalMinimumDistance* value is a measure of how close a point is from $\mathrm{AA_{NE}}$ to the point under consideration from $\mathrm{AA_{E}}$. $\mathit{MinimumDistanceMax} = 2$ is used for the blunt fin data set and $\mathit{MinimumDistanceMax} = 0.5$ is used for the delta wing data set because the length scales are different. These values were chosen to give good spacing for the final opinions $\omega_{\mathrm{R}}^{\mathrm{MA}}$. To maintain good spacing for other simulations as the length scales increase *MinimumDistanceMax* will increase and as the length scales decrease *MinimumDistanceMax* should decrease.

The constants in Eqs. 4.15 and 4.16 were defined in a similar fashion to the constants from Eqs. 4.1 and 4.2. Here the agent forming the opinion did not extract the region so it starts with less belief that the region is correct. This is shown by the constant 0.2 in Eq. 4.15 when before in Eq. 4.1 it was 0.6. If $\mathit{NormalAverage} = 1$ then $b = 1$ and $d = 0$ similar to setting the extracting algorithm agent opinion. Also, these equations have the condition that $b + d = 1$ and when $u > 0$ then $b + d + u > 1$ which is in direct contradiction with the condition of Eq. 2.17. This leads to a significant difference between the extracting and non-extracting opinions. In the extracting opinion if $b + d + u > 1$ then the disbelief and uncertainty are adjusted until $b + d + u = 1$. For the non-extracting agent opinion if $b + d + u > 1$ then the uncertainty is left unchanged and the belief and disbelief are adjusted until the condition $b + d + u = 1$ is met.

4.2.6 Master Agent Opinion

For each value in the MA's belief tuple the information from Table 3.3 is used to set the value. This information is quantified and input into a function which sets the value. The functions are shown in Eqs. 4.19–4.21. Eqs. 4.20 and 4.21 are linear functions while Eq. 4.19 is nonlinear. Linear functions were used to simplify the resulting equations but a planar function was needed for Eq. 4.19 to use information from both the feature displacement and the change in feature displacement. Recall that the FD and the ΔFD are defined for line-type features in Section 3.2.3 and measure the movement of features between iterations and the rate of change in feature movement between iterations.

$$b = \frac{-2.25 \cdot \Delta FD - 0.02 \cdot FD}{2} + 1 \tag{4.19}$$

$$d = 0.02 \cdot FD \tag{4.20}$$

$$u = 2.25 \cdot \Delta FD \tag{4.21}$$

The constants 2.25 and 0.2 from Eqs. 4.19–4.21 were determined to give the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$ good spacing for the probability expectation value. Different values for the constants were tried and a visual inspection was done of the extracted cores. The constants giving the best spacing for the probability expectation value were selected. Eq. 4.21 shows that when $\Delta FD = 0$ then $u = 0$ meaning that there is no uncertainty when FD does not change between iterations. When there is a large change in FD between iterations then the uncertainty value will be high. Eq. 4.20 shows that when $FD = 0$ then $d = 0$ meaning that when a vortex core does not move between iterations the MA has no disbelief in the extracted core. As the FD value increases, or as the extracted core lines are extracted at different spatial locations, the disbelief in the extracted cores will increase. Eq. 4.19 gives the belief value as a function of FD and ΔFD. When $FD = \Delta FD = 0$ corresponding to no feature movement between iterations and no rate of change in feature movement then the MA will have total belief in the extracted features. This corresponds to a CFD simulation being fully converged and when a CFD simulation is fully converged the MA should have a full belief in extracted features. When FD and ΔFD have large values this corresponds to extracted features moving between iterations meaning that the vortex cores are still converging so the MA has less belief in the extracting algorithm. The behavior of the FD value in a converging simulation can be seen in Section 5.1.1 and Figure 5.4.

4.3 Aggregating Final Vortex Core Feature Set

When aggregating a final feature set of vortex core lines there are two feature sets to select from. Each feature set contains vortex core lines that have been filtered using the three filters explained in Sections 4.1.1–4.1.3. At each point in each core line in both feature sets there is a final opinion that gives the belief, disbelief, and uncertainty of the point. From these values the expected probability of each point can be found using Eq. 2.19. The probability expectation value gives the expected probability of the point contained in each feature. From the expected probability of each point the expected probability of the entire line can be found by taking an average.

Currently, when forming the final feature set the two feature sets are searched through to determine if any features have been extracted in the same place. In other words, it is found if the same feature has been extracted by both algorithms. It happens frequently that the same vortex core has been extracted by both algorithms but the position and length of the core lines are not the same. There usually is a noticeable displacement between a core extracted by SH and RP even if it is the same core. Once similar cores are found, the core line with the highest average expected probability is selected for the final feature set and the other core is not.

After selecting the more probable similar cores, each feature set is manually searched to find if any other vortex cores have high expected probabilities. There is no fast and hard rule that defines when a feature is correct based on the expected probability or the individual belief tuple values. A good rule of thumb is that a vortex core should have an average expected probability value around 0.85.

Keeping in mind that this vortex feature extraction method is to be incorporated into the CAFÉ concept it is useful to consider how a researcher will work with a final feature set. If the vortex cores selected for the final feature set are ranked in order of decreasing expected probability with the first feature having the highest expected probability, then a researcher will have the best starting points for evaluating his CFD data set.

CHAPTER 5. METHOD VALIDATION

Two CFD simulations were run on separate geometries to verify that concurrent feature extraction is possible and to validate the vortex core extraction method described in Chapter 4. The two geometries are common in CFD feature extraction research: a blunt fin and a delta wing. The blunt fin was selected as an initial test case that had well defined vortex cores and was simple to grid and solve using a standard desktop computer. The delta wing data set was selected as it had a more complex flow field that would help validate the vortex core extraction method.

One crucial piece of information needs to be clear for a proper interpretation of results. When agents form opinions on extracted cores they have information from the current iteration of the simulation and previous iterations only. They do not use information from the fully converged simulation, or any iterations greater than the current iteration, to form opinions on extracted cores. Belief, disbelief, uncertainty, and expected probability of vortex cores can be determined without requiring a final converged solution giving information about a final simulation's expected vortex cores before a simulation is 100% converged.

5.1 Blunt Fin

A CFD simulation was run of a blunt fin [31] geometry using the steady RANS equations solved using Fluent 6.3. The computational domain was generated as a structured curvilinear grid with 44,000 nodes and is displayed in Figure 5.1. The Reynolds number based on the fin diameter was 630,000 and the one equation Spalart-Allmaras method was used to model turbulence. The inlet boundary condition was a pressure-inlet condition with the flow velocity constrained to the downstream direction. The inlet velocity profile from the Hung and Buning blunt fin was used as the input velocity profile for the pressure-inlet boundary condition with a freestream $M = 2.95$. The outlet boundary condition was a pressure-outlet condition, the top boundary condition was a symmetry condition. The fin and the lower boundary were modeled as walls. The flow solver was a compressible pressure-based solver and the flow field was initialized using a small velocity in the downstream direction. The solution reached full convergence at 900 iterations and the simulation residuals are displayed in Figure 5.2. Concurrent feature extraction was replicated by exporting

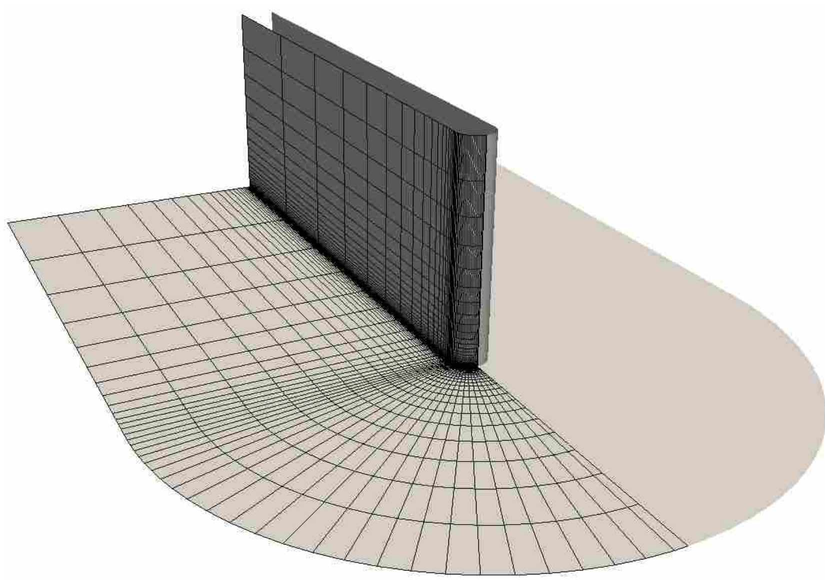


Figure 5.1: Computational domain for the blunt fin simulation.

and saving to hard disk the entire flow field data set every 45 iterations throughout the flow solution. Each of these saved data sets were then input into the vortex core extraction method described in Chapter 4 where vortex core lines were extracted from each of the saved data sets using the RP and the SH algorithms (see Section 2.3) resulting in two feature sets per saved data set. Agents then produced final opinions on all features in both feature sets and a final feature set was selected per data set. This simulation is not of the magnitude where concurrent feature extraction is required, but does yield a good starting point for concept verification.

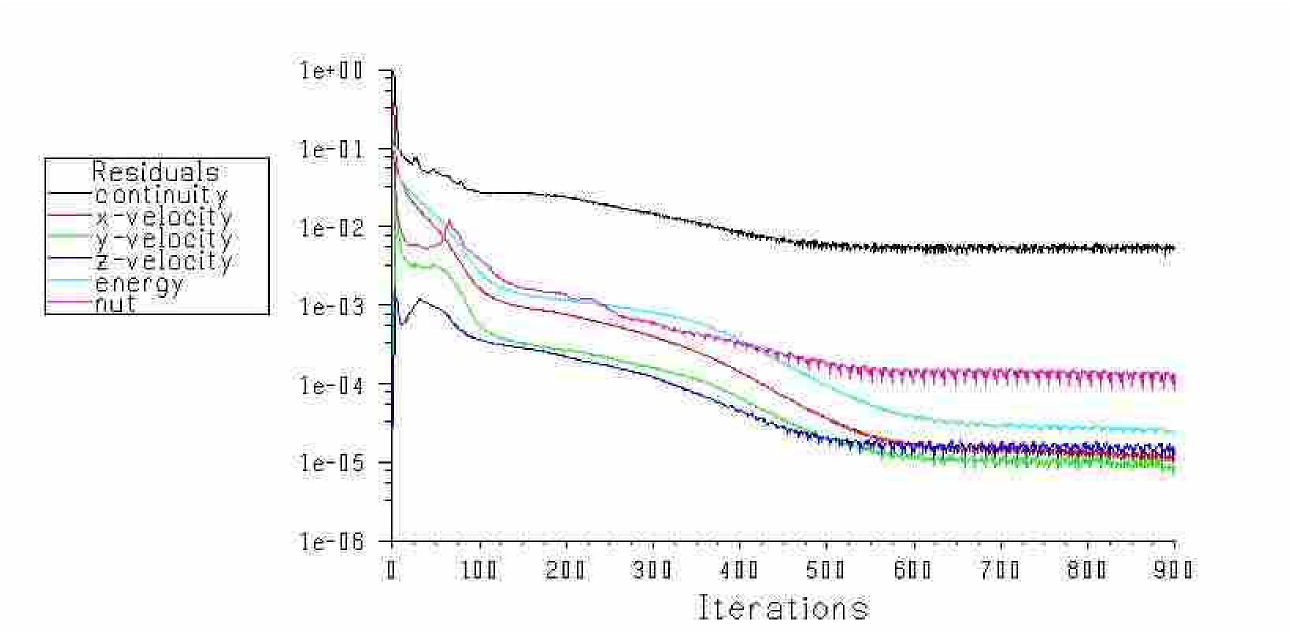
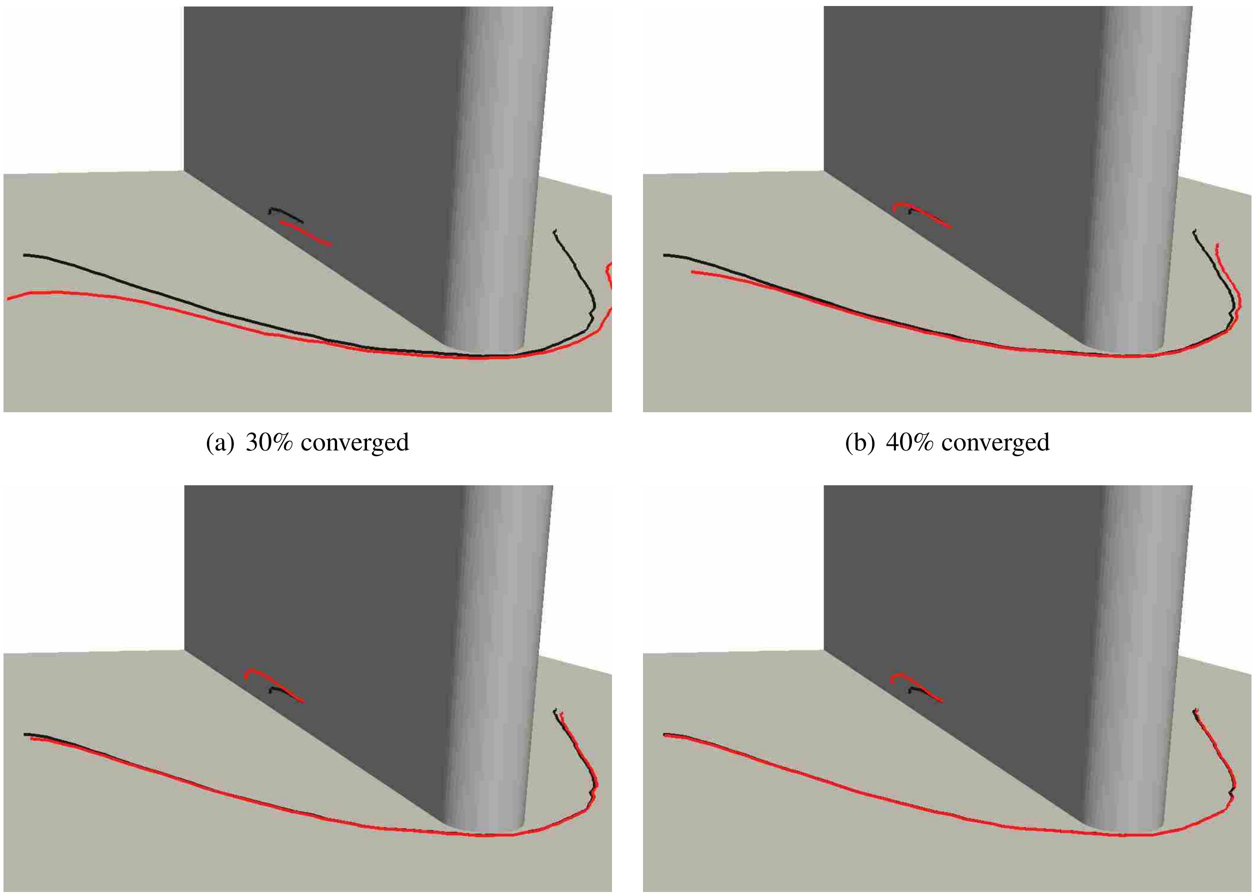


Figure 5.2: Residual plot for the blunt fin simulation.

5.1.1 Vortex Cores in Converging Data Sets

Figures 5.3a–d display the vortex core extraction results obtained from the RP algorithm. Extraneous cores have already been filtered out to make the images easier to understand. The black lines represent extracted vortex core lines from the converged data set and the red lines represent extracted core lines from the converging data sets. The percent convergence is obtained by dividing the iteration containing the converging cores by the number of iterations at full solution convergence and multiplying by one hundred. In order to give a visual comparison, the core lines extracted from the converged solution, i.e. the algorithms correct features, are displayed with the core lines extracted at intermediate steps.



(a) 30% converged

(b) 40% converged

(c) 50% converged

(d) 60% converged

Figure 5.3: Comparison of RP extracted vortex core lines from the converged data set (black) and converging data sets (red). (a) At 30% converged the horseshoe line begins to take shape upstream. (b) At 40% converged the horseshoe line and the fin line are almost correctly resolved. (c) At 50% converged the end point of the fin line moves downstream. (d) At 60% converged the horseshoe line is spatially correct but the fin line is not.

There are two vortex core lines for the blunt fin data set: the horseshoe vortex core line and the fin vortex core line. The core line that forms around the front of the fin in a horseshoe like shape is called the horseshoe line. The core line near the side of the fin is called the fin line. The horseshoe vortex forms as a result of flow separation upstream of the blunt fin. The fin vortex forms as a result of a high pressure region near the middle of the fin and a low pressure region near the bottom of the fin. The interaction of these two pressure regions causes the flow to swirl creating the fin vortex. The low pressure region is a result of the horseshoe vortex and the flow expanding around the fin while the location of the high pressure region corresponds with flow stagnation. Experimental results of a blunt fin have shown the formation of these two features. [32]

Figure 5.4 shows a graph of the feature displacement for the endpoints of the horseshoe core line and the fin core line extracted by the RP algorithm and displayed in Figures 5.3a–d. The start point is defined as the farthest upstream point and the end point is defined as the farthest downstream point. At 60% converged, all but the end point of the fin line has a non-negligible feature displacement. This shows that at 60% converged the entirety of the horseshoe vortex core line is very close to the same position it will be in at full solution convergence which can be seen in Figure 5.3d. The start point of the fin line has the same behavior as the horseshoe line. The end point of the fin line has a feature displacement within 2.5% and 10% from 55% converged to full solution convergence. This tells us that the end point of the fin line does not find a fixed position, but rather continues to move slightly every 45 iterations between 55% converged and fully converged. This behavior suggests that the simulation is not fully converged as the RP algorithm takes two spatial derivatives of velocity to locate vortex cores which makes it very sensitive to variations in the velocity field solution. Making sure that the feature displacement is zero for all features in the spatiotemporal flow domain extracted by the RP algorithm could aid in determining if a CFD simulation has reached full solution convergence. If the feature displacement is not zero then the features are continuing to move suggesting that the flow solution has not converged. The two vortex core lines exhibit similar behavior when they are extracted by the SH algorithm.

While it is possible to monitor features and their corresponding feature displacement to aid in checking for solution convergence, a simulation should not be considered complete as soon as features are converged. Extracting features is just a starting point to massive data set post-processing and certainly more information beyond features such as coefficient of lift, coefficient of drag, boundary layer profile, vorticity, etc. are needed to thoroughly post-process CFD data sets. If the blunt fin simulation were terminated at 60% converged when the horseshoe core line was spatially converged then useful information could be lost such as an exact value for coefficient of drag and the simulation would be inaccurate.

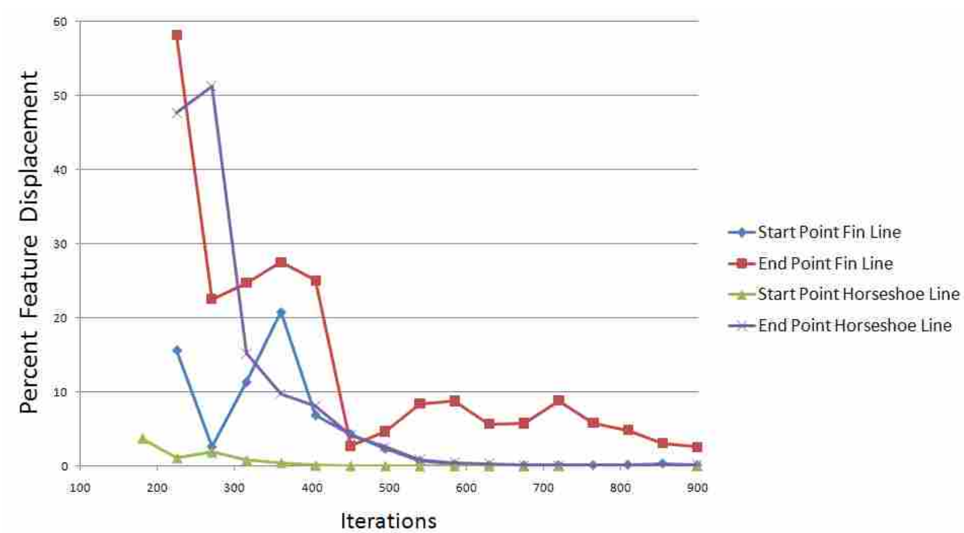
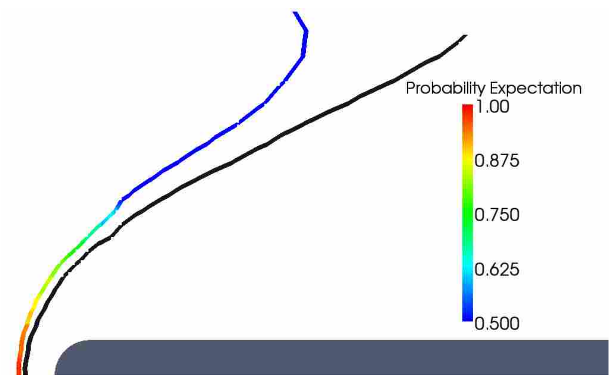


Figure 5.4: Percent feature displacement for the endpoints of the horseshoe line and the fin line extracted by the RP algorithm.

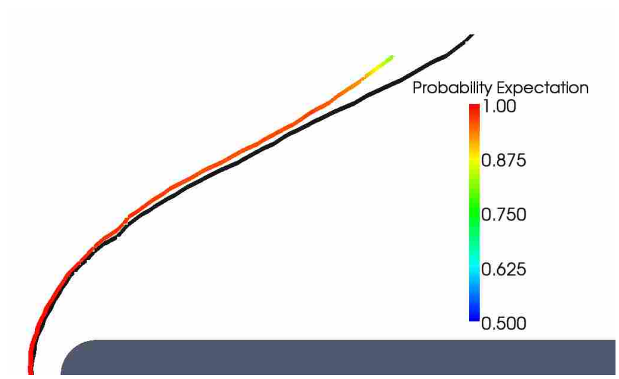
It is interesting to note that the upstream start point moves to its final location sooner than the downstream end point for both core lines. Recall from Section 3.2.3 that FD is defined by Eq. 3.7 as the displacement of a region between iterations nondimensionalized by the length of the line for line-type features. For the horseshoe line the FD at the start point is 0.8% at 35% converged and the FD at the end point is 0.8% at 60% converged. This suggests that the vortex core lines are convected downstream as the solution converges. This convection can also be seen in Figure 5.5.

5.1.2 Vortex Cores in Converging Data Sets Processed by Agents

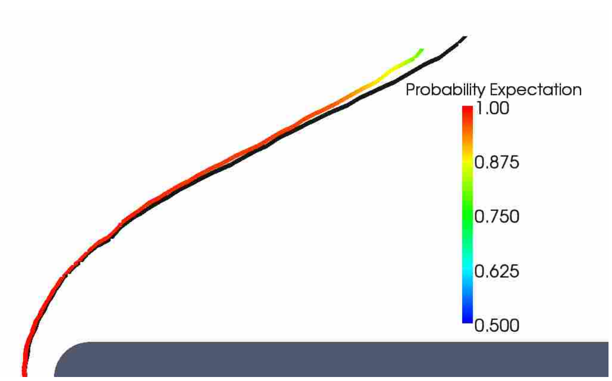
Figures 5.5a–d are a comparison of the probability expectation between four separate core lines extracted by RP at 30%, 40%, 50% and 60% converged. Recall that the probability expectation defined in Eq. 2.19 gives what one would expect the probability of a feature to be. The converged line is colored black to represent the exact location of the final core line. It is this core that we are trying to match. In these figures the flow is moving from left to right.



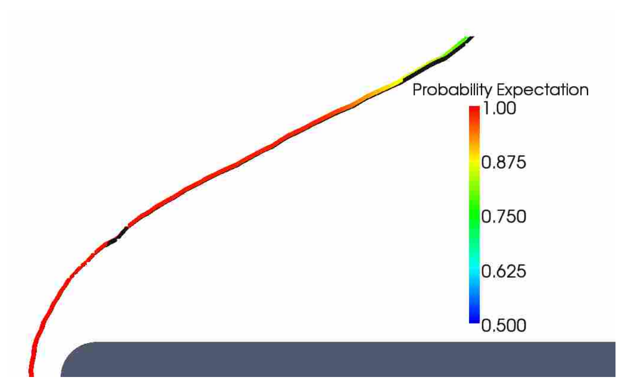
(a) 30% converged vs. converged



(b) 40% converged vs. converged



(c) 50% converged vs. converged



(d) 60% converged vs. converged

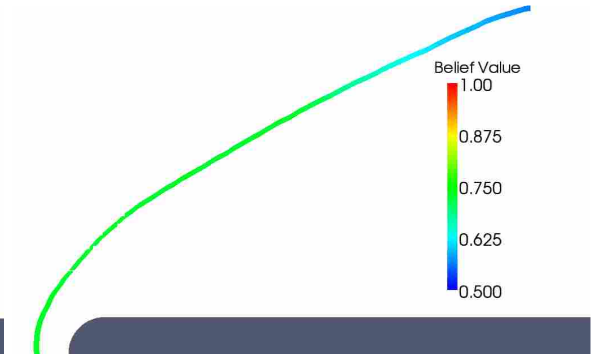
Figure 5.5: Comparison of horseshoe core lines extracted by RP algorithm at 10% convergence increments. The black line represents the extracted core line from the final converged solution. Flow is moving from left to right.

At 30% converged the probability expectation value is close to 1 at the start point and then quickly transitions to 0.5 at the downstream end point which tells us that only the area near the start point has a high expected probability. A high expected probability is approximately 0.85 and above. At 40% and 50% converged the probability expectation value is close to 1 at the start point and stays close to 1 until near the end point which indicates that these lines are highly probable. This is a correct analysis by the agents since both lines are close to the final line. The 60% converged core line is almost identical spatially to the fully converged core line but the agents have only given most of the line an expected probability of 0.90 or above and the rest is lower with the end reaching an expected probability of 0.75. The reason for this is that near the end point of the horseshoe line the vortex strength has a low value which is one of the input criterion for belief in a feature. This low value drives the belief down at the end of the horseshoe line and therefore drives the probability expectation value down. In these cases agents correctly identify the core line at 30% convergence as having a low expected probability and correctly identify the core lines at 40%, 50% and 60% as having a high expected probability.

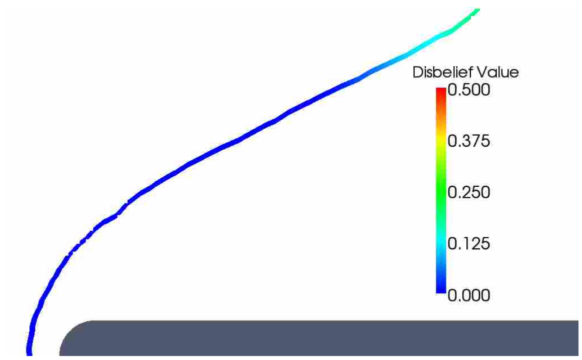
5.1.3 Comparison of Vortex Cores Processed by Agents from Converged Solution

Figure 5.6 is a comparison of the horseshoe line extracted at full solution convergence by the RP algorithm and the SH algorithm after agents have formed final opinions. This particular situation represents a case where both feature extraction algorithms have extracted a feature in a similar location so one more probable feature must be selected. Referring to Table 4.2 it can be seen that a belief criteria for the RP algorithm was curved. This core line contains a high curvature so the corresponding belief value for AA_E when RP extracts the line will be high which is shown in Figure 5.6a. Weaknesses for the RP algorithm were not found in the extracted core which makes

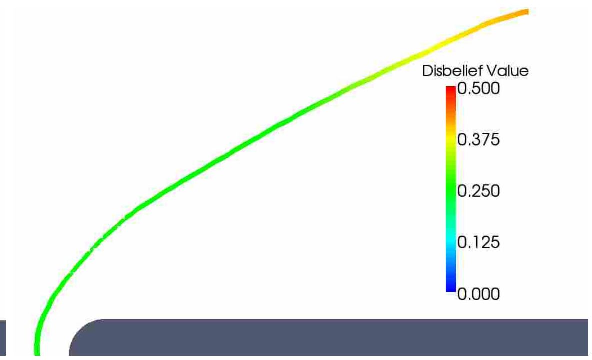
(a) RP belief value



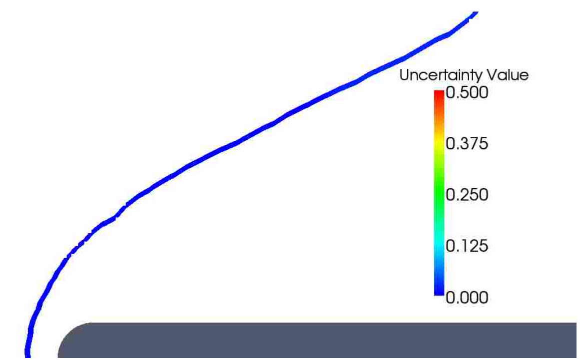
(b) SH belief value



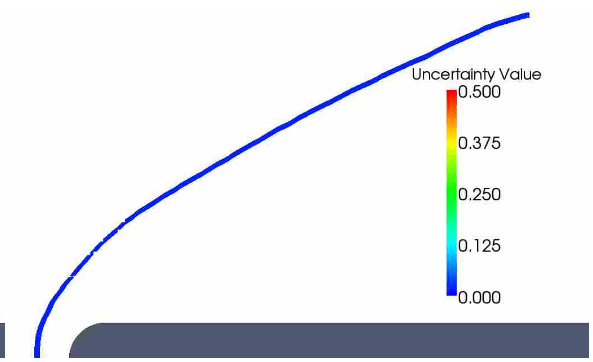
(c) RP disbelief value



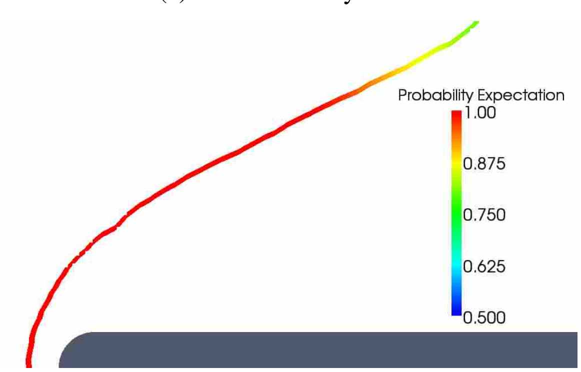
(d) SH disbelief value



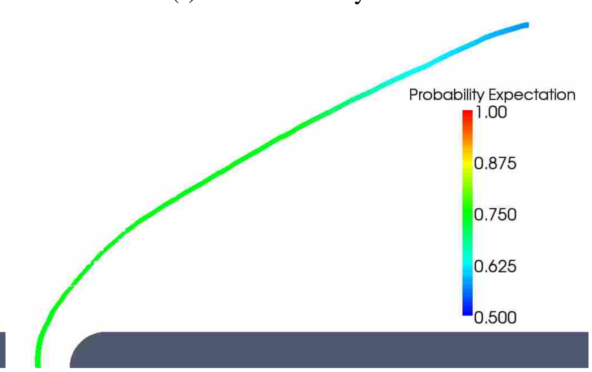
(e) RP uncertainty value



(f) SH uncertainty value



(g) RP probability expectation value



(h) SH probability expectation value

Figure 5.6: Comparison of the belief tuple values and probability expectation value for the horseshoe core line from the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$ of the converged data set extracted by the RP and SH algorithms at full solution convergence. Flow is moving from left to right.

for the low disbelief in Figure 5.6c. From Table 4.1 a criterion to set the disbelief for SH is curved so the corresponding disbelief for AA_E when SH extracts the line will be high which is shown in Figure 5.6d. Only some strengths were found in the SH extracted core line giving a belief around 0.75 for most of the horseshoe core line. The uncertainty values for both algorithms in Figures 5.6e & f are low showing that the distance from a vortex trip point is low and that the horseshoe core line is converged as the FD and ΔFD must also be low for the uncertainty to be low.

For the horseshoe core line the RP horseshoe line was selected as most probable because the probability expectation value throughout the line was higher as well as the belief value. The probability expectation values are shown in Figures 5.6g & h. Based on the strengths and weaknesses input criteria, agents correctly selected the RP horseshoe line as the feature with the highest expected probability.

5.2 Delta Wing

A CFD simulation was run of a delta wing using the steady Reynolds-averaged Navier-Stokes (RANS) equations solved using Fluent 12.0. The computational domain was generated as an unstructured tetrahedral mesh with 6,065,247 nodes. The inlet mach number was 0.3 and the wing was at $\alpha = 10°$. The inlet boundary condition was a pressure-inlet condition, the outlet boundary condition was a pressure-outlet condition, and the delta wing was modeled as a wall. The Fluent Full Multigrid Initialization (FMG) [33] technique was employed to generate an initial condition. FMG initialization uses iterations on computationally cheap coarse levels and a few on computationally costly fine levels to provide a better initial condition with lower computational cost. The k-ω SST model was used to model turbulence and the solver was a pressure based compressible solver. The simulation reached convergence at 1900 iterations and the simulation residuals are displayed in Figure 5.7.

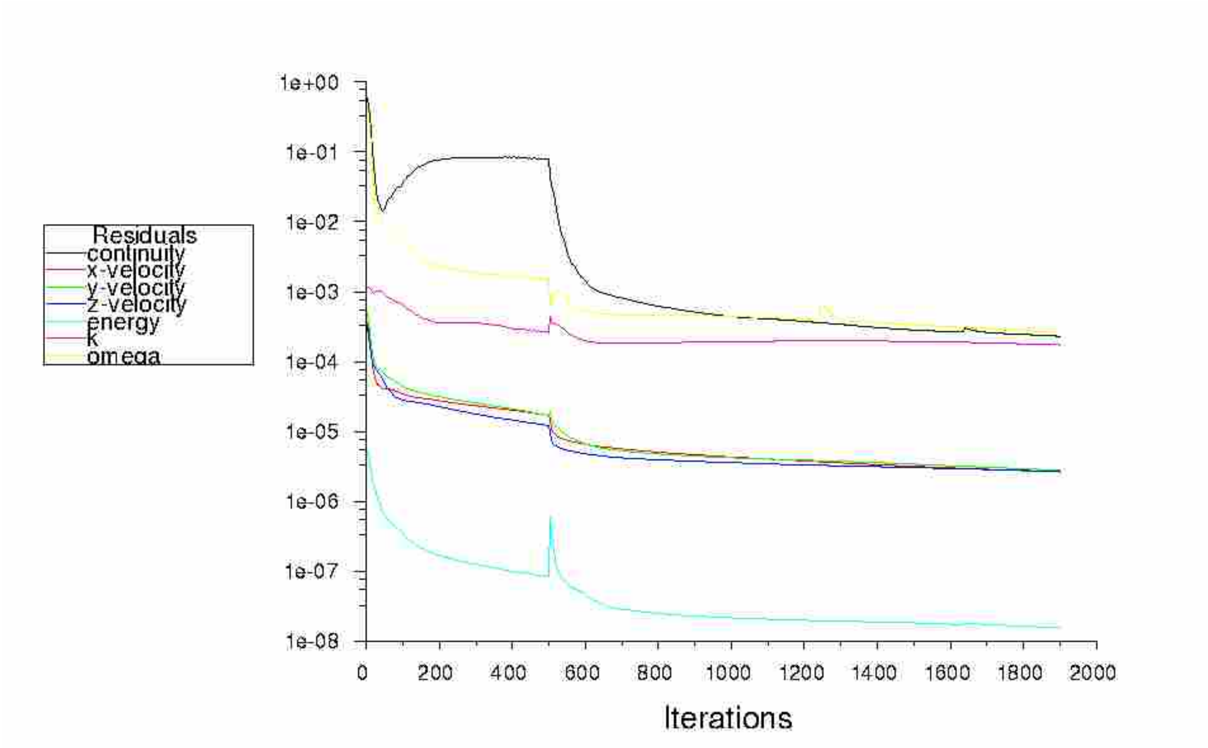
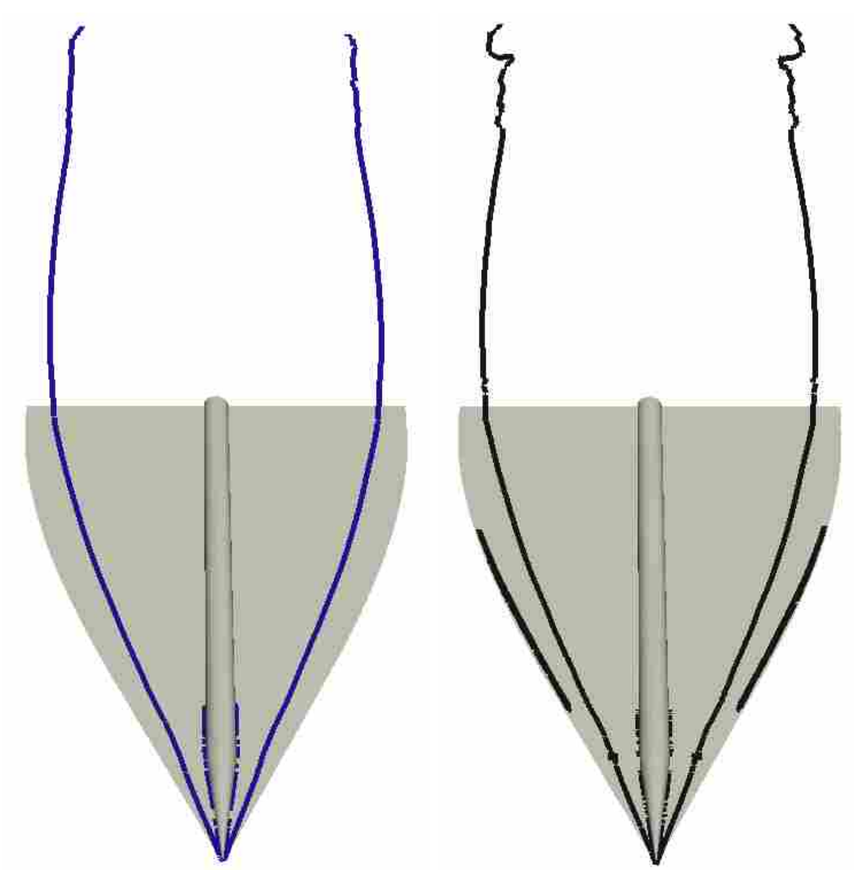


Figure 5.7: Residual plot for the delta wing simulation.

Concurrent feature extraction was replicated by exporting and saving to hard disk the entire flow field data set every 200 iterations, starting at 100 iterations, throughout the flow solution. Each of these saved data sets were then input into the vortex core extraction method described in Chapter 4 where vortex core lines were extracted from each of the saved data sets using the RP and the SH algorithms (see Section 2.3) resulting in two feature sets per saved data set. Agents then produced final opinions on all features in both feature sets and a final feature set was selected per data set. Like the blunt fin, this simulation is not of the magnitude where concurrent feature extraction is required but does yield a good test point for validating the feature extraction method.

5.2.1 Definition of Extracted Vortex Cores

Figure 5.8 shows the vortex core lines extracted by the RP and SH extraction algorithms. In this figure flow is moving from the bottom to the top of the page. The core lines that extend from the nose of the delta wing well beyond the trailing edge are called the primary lines. They can be seen in both the RP and SH feature sets. In the RP feature set there are two lines that are outboard of the primary lines laying almost directly on the edge of the delta wing. These lines are called the secondary lines and are not contained in the SH feature set. Near the leading edge running along the intersection of the fuselage and the wing are two lines called the tertiary lines. The tertiary lines are contained in both the SH and RP feature sets.

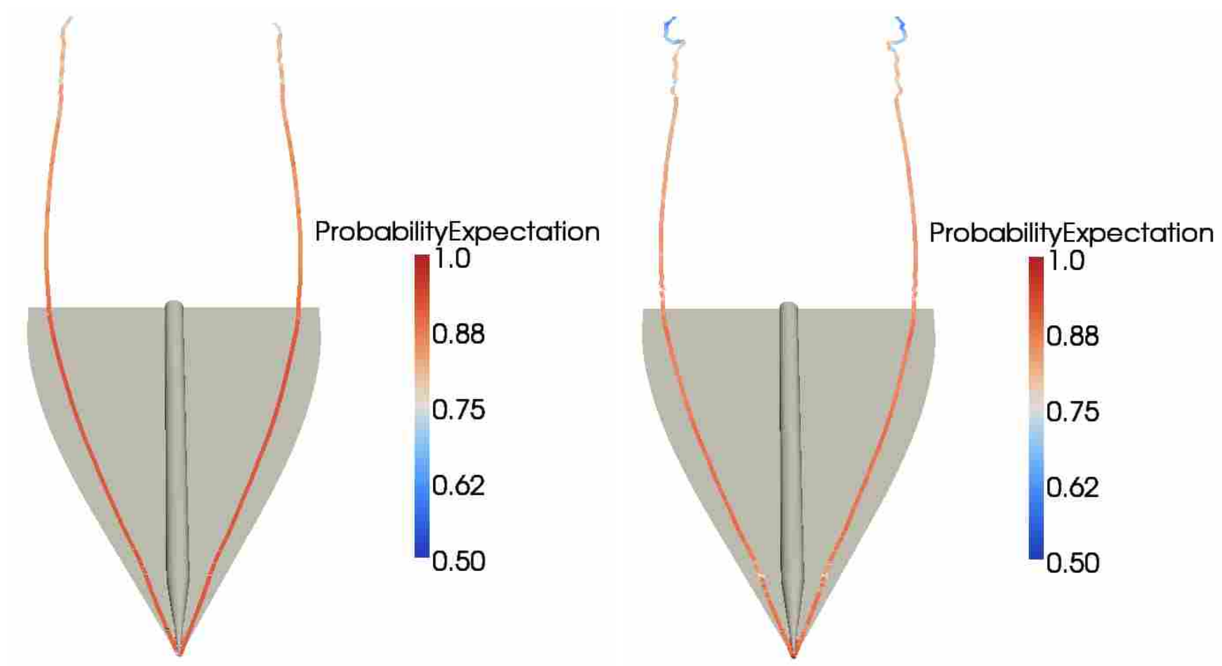


(a) Cores extracted by SH (b) Cores extracted by RP

Figure 5.8: Display of vortex cores extracted by RP and SH algorithms at full solution convergence.

5.2.2 Comparison of Vortex Cores Processed by Agents from Converged Solution

Figure 5.9 shows the primary vortex cores extracted by the RP and SH algorithms. Each of the primary cores is colored by the probability expectation value from the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$. The expected probability is high for both algorithms across most of the primary cores. At the downstream sections of the primary cores for both algorithms the probability expectation value is substantially lower than the upstream portions. For the RP algorithm the probability expectation value is around 0.65 at the farthest downstream section. This decrease in expected probability is due to a large value for vortex strength at the upstream portion of the primary cores and then a decreasing value for vortex strength as the core extends downstream as shown in Figures 5.10c & d. Physically the primary cores are dissipating as they travel downstream which is giving the decrease in vortex strength leading to a smaller belief value set in the opinions $\omega_{\mathrm{R}}^{\mathrm{AA}_1}$ and $\omega_{\mathrm{R}}^{\mathrm{AA}_2}$.



(a) Primary cores extracted by SH

(b) Primary cores extracted by RP

Figure 5.9: Display of primary cores extracted by RP and SH algorithms colored by the probability expectation value from the final opinion at full solution convergence.

Figures 5.10a & b show the quality of the primary cores from the converged solution. The quality of the primary cores for both algorithms is low which leads to a higher expected probability. For the downstream section of the primary cores for the RP algorithm the quality is high which also helps to decrease the expected probability with the decreased value for vortex strength. This high quality for the primary cores at the extreme downstream section is not seen for the SH algorithm where the quality is low throughout the entire core.

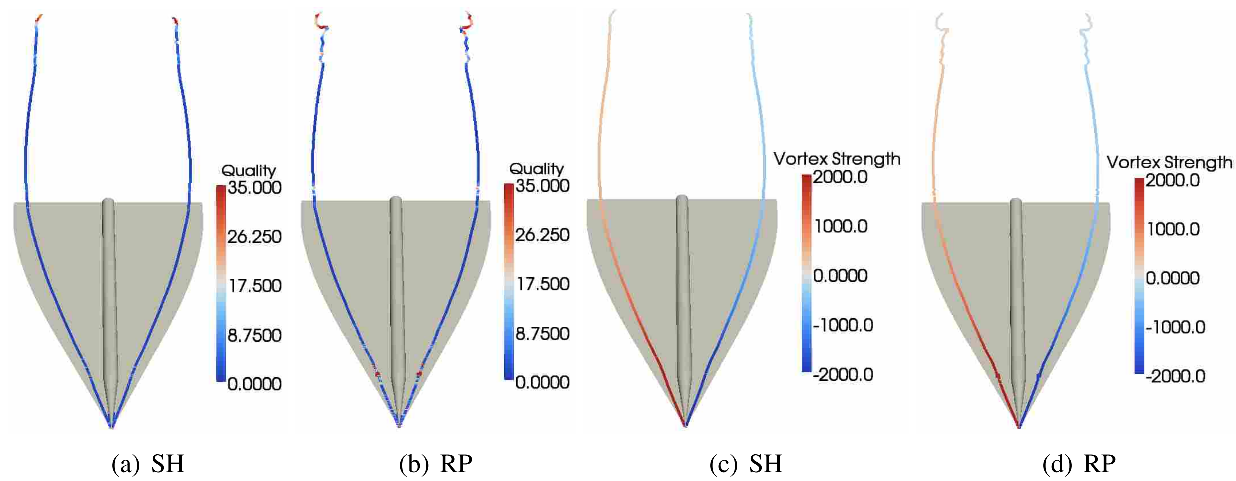


Figure 5.10: Display of primary cores extracted by RP and SH algorithms colored by quality and vortex strength from the final opinion $\omega_{\mathrm{R}}^{\mathrm{MA}}$ at full solution convergence.

The difference in expected probability of the primary cores for the SH and RP algorithms is slight, but overall the expected probability is higher for SH than RP. The main reason that the expected probabilities are so close is that the strength and quality values are nearly identical across the majority of the primary lines making the curvature value the main cause of the difference. The curvature value for the two sets of primary lines lies almost directly between the zero curvature and high curvature conditions. Recall from Tables 4.1 & 4.2 the strength condition for the SH algorithm is a straight core line and the strength for the RP algorithm is a curved core line. For this simulation $CurvatureMax = 0.3$, $Curvature = 0.100$ for SH, and $Curvature = 0.095$ for RP. In Section 4.2.4 selecting *CurvatureMax* is explained. These values lead to setting the belief value for the opinion $\omega_{\mathrm{R}}^{\mathrm{AA}_1}$ higher than the belief value for the $\omega_{\mathrm{R}}^{\mathrm{AA}_2}$ opinion which gives a higher belief value in the final opinion for the primary cores extracted by SH and therefore a higher expected probability. This higher overall expected probability for SH leads to selecting the primary cores extracted by SH for the final feature set.

Do the agents make a correct decision when selecting the primary cores extracted by SH over the primary cores extracted by RP? Based on the selection criteria, yes. While it may be alarming that the primary cores have expected probabilities that are so similar, it should not be. It may be expected to have one core with a much higher probability than the other when they are extracted in such a similar spatial location. In some situations this may be the case, but not in all situations. Since the only real difference between the extracted cores is the curvature value, which is between the straight and curved core conditions, the expected probabilities are similar. Recall that the idea behind subjective logic is not to make a strict yes or no decision about a situation, but rather to make a human estimate of a situation. Applying this idea to the primary cores there is a human estimate that both sets of primary cores have high expected probabilities, but the primary cores that are the most probable are the cores extracted by SH. This is the reason SH was appropriately selected.

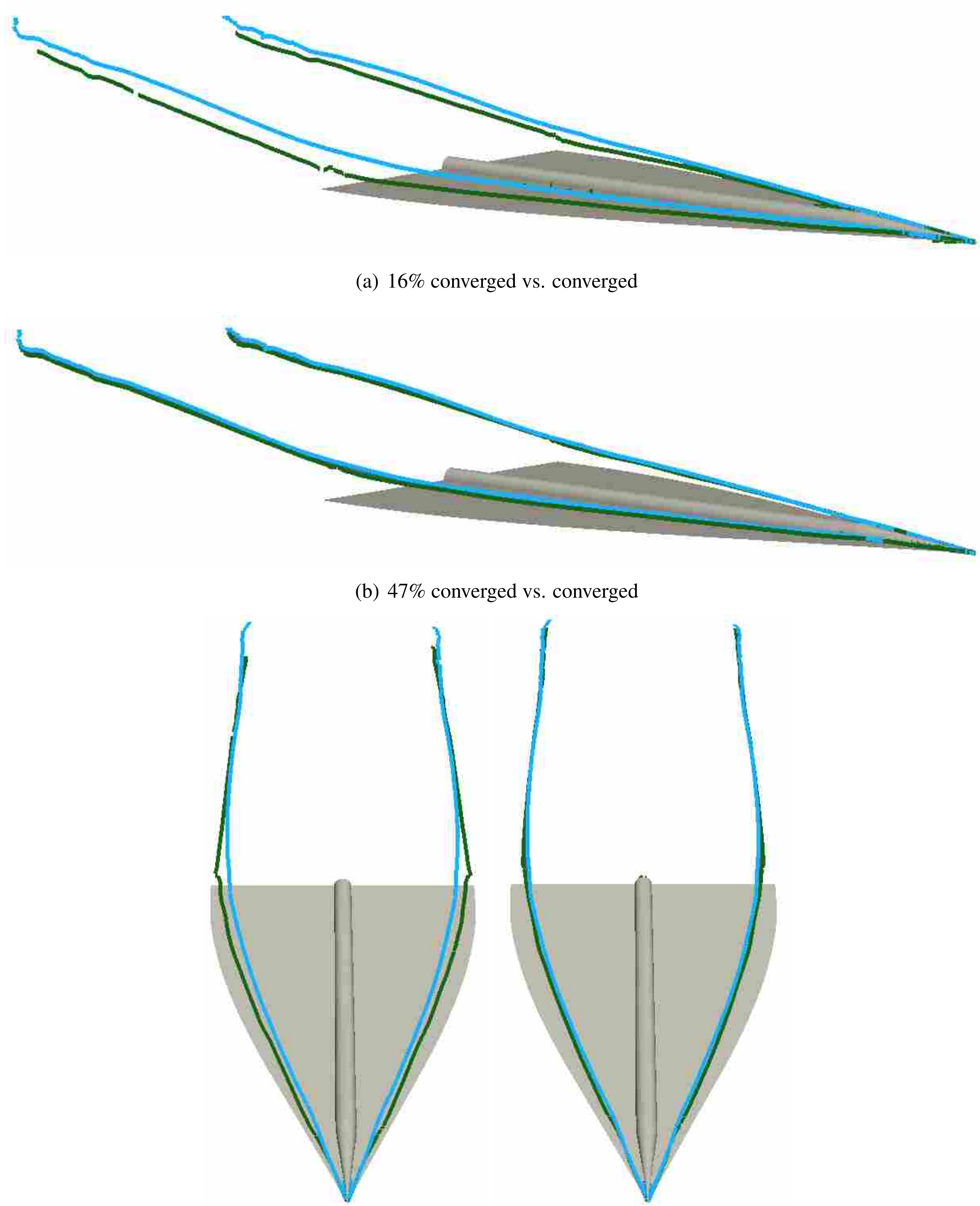
5.2.3 Vortex Cores in Converging Data Sets

In the simple case of the blunt fin it was seen that vortex cores could be detected early enough in a simulation to warrant concurrent feature extraction. It was also seen that the vortex cores were convected downstream as the solution approached convergence. In this section it is shown how the delta wing's vortex cores behave as its simulation converges.

Figure 5.11 displays the primary cores from the converged simulation (cyan) with the primary cores from two converging data sets (green). In Figures 5.11a and 5.11b the flow is moving from right to left. At 300 iterations, or 16% of solution convergence, the primary cores can be extracted by the SH algorithm. When extracted at 16% converged the primary cores are noticeably downstream of their converged location but still relatively close. At 47% of solution convergence the primary cores are extracted close to their final converged spatial location. No iterations after 47% converged were visualized as it is difficult to visually distinguish the converging cores from the converged cores.

Recall that the delta wing is flying at $\alpha = 10°$. A visual inspection reveals that the cores convect downstream with the flow near this angle of attack. There is some movement of the cores along the length of the wing as seen in Figures 5.11c & 5.11d as well as some movement normal to the top of the delta wing as seen in Figures 5.11a & 5.11b. This corroborates the findings from the blunt fin that vortex cores are convected downstream.

One reason for the cores being present and well defined so early on in the simulation is due to the FMG initialization performed to obtain a better initial guess. What FMG does is compute solutions on a coarser grid and then set that coarse grid solution as a starting point for the fine grid. This method helps to resolve the cores before the iteration count is started. If the FMG initialization were not used it would take more iterations for the cores to develop.



(a) 16% converged vs. converged

(b) 47% converged vs. converged

(c) 16% converged vs. converged (d) 47% converged vs. converged

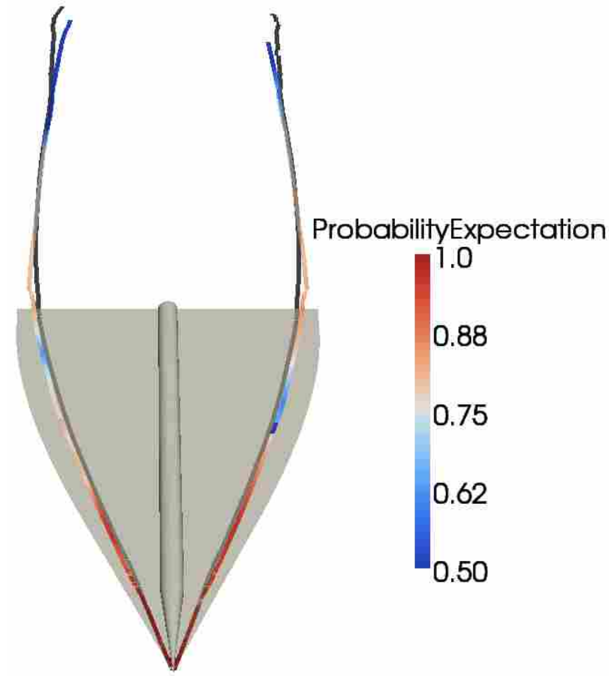
Figure 5.11: Display of converging vortex cores extracted by SH algorithm.

5.2.4 Expected Probability of Converging Cores

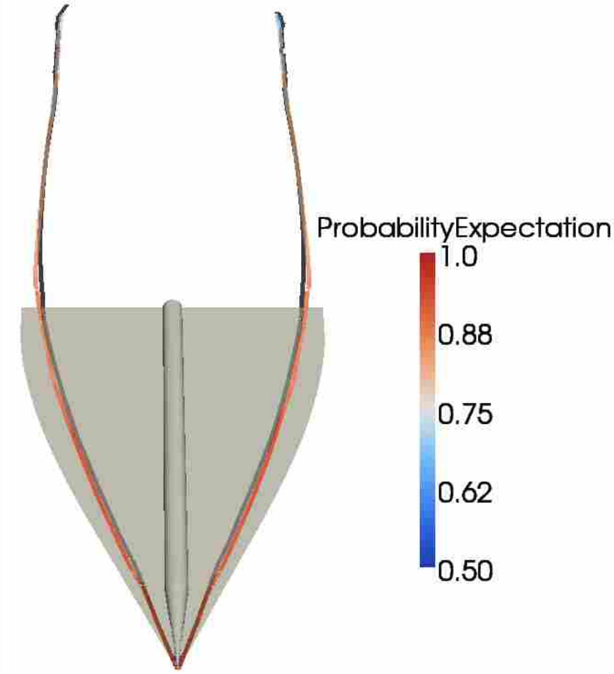
Extracting vortex cores early on in CFD simulations has limited value until a measure can be made about the extracted cores. Are the extracted cores in the correct spatial location? Are there portions of the cores that have been extracted correctly and portions that are spurious? The probability expectation value gives a measure about the expected probability of vortex cores extracted from converging data sets which can answer these questions.

Figure 5.12 shows the expected probabilities of the primary cores extracted by the SH algorithm at specific iterations before the solution has converged. Also, on each subfigure there is an overlain opaque image of the primary cores extracted from the converged simulation. In Figure 5.12a there is a comparison between the primary cores extracted at 26% converged and the cores extracted from the fully converged solution. The portions of the primary cores with the lowest expected probabilities, around 0.50, are at the downstream ends of the cores. This is due partly to the low vortex strength but mainly to a large feature displacement. Portions of the primary cores with large values for FD and ΔFD will have the lowest expected probabilities. Recall that in Section 4.2.6 the belief tuple for MA is set based on FD and ΔFD. When FD and ΔFD are large the belief will be low and the disbelief and uncertainty will be high for the opinion $\omega_{\mathrm{AA}_1}^{\mathrm{MA}}$.

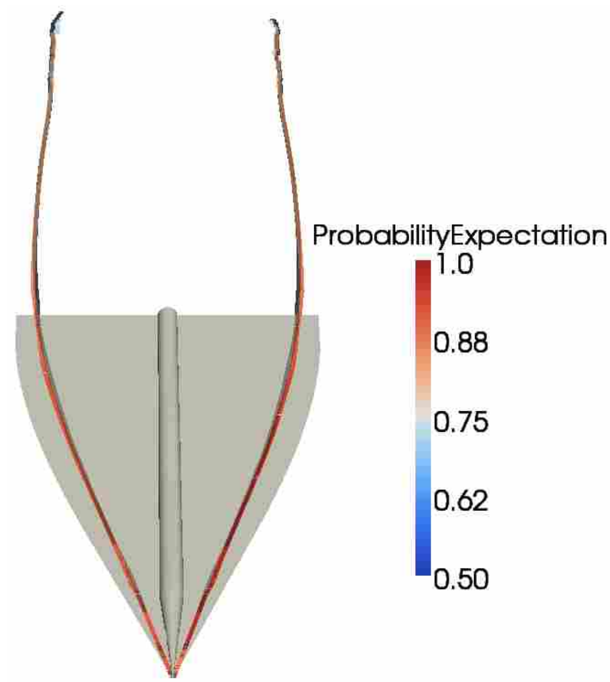
In Figures 5.12b & 5.12c there is only slight spatial variation between the converging primary cores and the converged primary cores. The probability expectation value for both primary cores is high as anticipated when FD is low. In Figure 5.12d there is close to no spatial variation between the primary cores at 89% converged and fully converged. The 89% converged primary cores have close to the same expected probability as the fully converged primary cores shown in Figure 5.9a. These are correct interpretations of the converging primary cores by the agents.



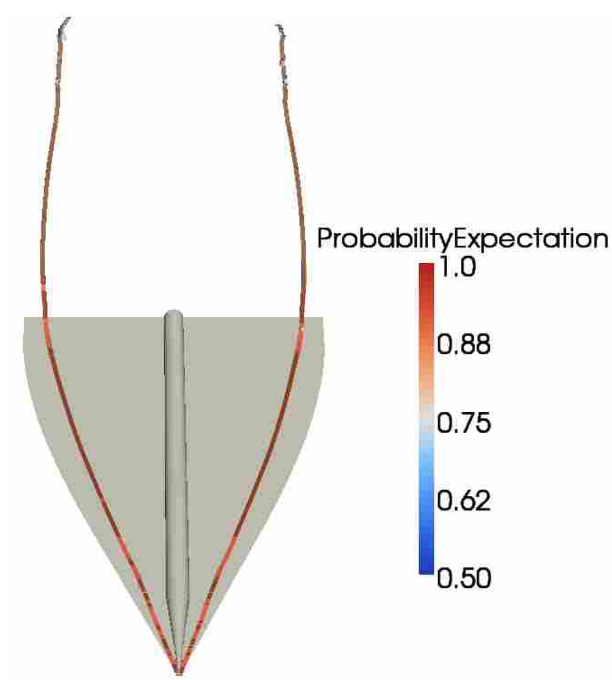
(a) 26% converged vs. converged



(b) 47% converged vs. converged



(c) 68% converged vs. converged



(d) 89% converged vs. converged

Figure 5.12: Display of expected probability for converging vortex cores extracted by the SH algorithm with primary cores extracted from the converged simulation overlain.

CHAPTER 6. RECOMMENDATIONS

This research and the CAFÉ concept are still in development. This chapter gives recommendations for future research and development.

6.1 CAFÉ and the General Feature Extraction Method Recommendations

Thus far vortex cores have been chosen to demonstrate extracting features from converging data sets. Two other features that are common in high-fidelity CFD applications are shock waves and separation and attachment lines. One new research direction would be to extract these two features from converging data sets to find if they behave similar to, or different from, vortex cores. Also, with their corresponding feature extraction algorithms they may be input into the general feature extraction method from Chapter 4.

In Section 3.2.2 the opinion for the non-extracting algorithm agent, $\omega_{\mathrm{R}}^{\mathrm{AA}}$, is given. This opinion is based on the extracting algorithm strengths and weaknesses and the distance from a region extracted by the non-extracting algorithm agent. Essentially this opinion adds or subtracts uncertainty to the final opinion. It could be possible to cut the non-extracting algorithm agent opinion out of the agent structure altogether. This would result in a trust network that is a line as shown on the left side of Figure 3.2. While this would make the method simpler, the final opinion would not take into account the features extracted by other feature extraction algorithms. Further development would find if there is any significant difference between the two trust networks and if so, which trust network is superior.

As explained in Section 3.3 the process to aggregate a final feature set is not automated. When implemented in CAFÉ the end user may not want to select the final feature set, but rather have the final feature set already selected. This aggregation process may be implemented by using a simple search criterion to locate cores in common between feature sets and then selecting the cores with the highest expected probabilities. Also, a threshold criterion for cores that are not similar between feature sets must be set based on probability expectation, belief, disbelief, uncertainty or some combination of these values. Future research would find how to locate the common cores, select between them and then parse through the remaining feature sets to select the remaining most probable cores in an automated fashion.

6.2 Vortex Extraction Method Recommendations

The main limitation to the vortex feature extraction method is that maximum values used to set the opinion $\omega_{\mathrm{R}}^{\mathrm{AA}}$ such as $VortexStrengthMax$ are not properly defined across data sets with varying flow conditions. The $VortexStrengthMax$ value is set based on each data set and the range of values for vortex strength seen in that data set. There needs to be a way to find $VortexStrengthMax$ based upon the simulation Reynolds number, inlet mach number, or some other common flow value. If this is not the case then a different measure of vortex strength may be used. Future research would find a function based only on standard CFD values that could be used with any CFD data set to set $VortexStrengthMax$.

Currently CAFÉ has concentrated on RANS and URANS simulations, but it has the ability to aid in post-processing other high-fidelity CFD simulations such as LES and DNS. To do this the general and vortex feature extraction processes need to be validated on LES and DNS data sets. As turbulent eddies are either partially or fully resolved in these codes it might be difficult to distinguish turbulent eddies from vortices. Future research would extract vortex cores from LES and DNS data sets to find how to distinguish turbulent eddies from vortices.

Also as codes scale up in grid resolution, other parts of the extraction method such as filters may be affected. The point count filter explained in Section 4.1.1 may need a higher minimum point count to threshold extraneous cores as the number of points contained in cores increases as the grid resolution increases. The quality filter will be affected with grid resolution as well. As the grid is increased the quality of the extracted cores should decrease as the flow vector and the direction vector of the core line will be more closely aligned. Future research would find the affect that grid density has on setting the minimum threshold values for filters such as the point count filter and the quality filter.

In the extracting algorithm agent opinion the uncertainty value is defined by feature specific characteristics. Currently the only vortex characteristic implemented is the distance from a vortex trip point. For the agents to make more intelligent decisions other feature characteristics need to be inserted. One possible vortex characteristic is the 2π criterion used in the Evita [4] concept which defines a vortex core as having a streamline that rotates one full revolution around the core. Future research would add more vortex characteristics to the extracting algorithm agent opinion.

Lastly one challenge has been to find a proper definition for the curvature of a line and then its proper software implementation. Curvature is currently calculated by taking the two line endpoints and midpoint to find the radius of the containing circle. The inverse of the radius is then the curvature. While this definition is sufficient to calculate curvature it would be better to have a local definition of curvature rather than a single global curvature value for the entire core line as opinions are calculated in a pointwise fashion. Future research would find an accurate local definition of curvature and then implement it.

CHAPTER 7. CONCLUSIONS

This research has developed a general method to extract fluid flow features from converging and converged CFD simulations using intelligent software agents governed by subjective logic. The general feature extraction method contains five basic steps which may be applied to any CFD flow feature with corresponding feature extraction algorithms. These five basic steps are as follows:

1. Extract features using feature extraction algorithms
2. Filter obviously extraneous features
3. Create agent opinions at regions contained in each extracted feature
4. Combine agent opinions to form final opinions of features
5. Aggregate one final feature set from all available feature sets

After defining the general feature extraction method, the method was applied specifically to vortex core lines. Three specific filters were used to filter out extraneous core lines: point count, quality, and vortex strength. The SH and RP algorithms were used to extract vortex core lines as they are robust algorithms with strengths and weaknesses that are complimentary. The information and functions necessary to set each component in each agent belief tuple was given along with an explanation of the methods for setting the components.

Before agents were applied to converging simulations it was found if vortex cores could coherently be extracted from CFD data sets that were still converging. Results from the blunt fin simulation showed that the horseshoe core line could be extracted coherently as early as 30% into a converging simulation and that at 60% of the converged solution the horseshoe core line had little to no spatial variation from the horseshoe core line extracted from the fully converged solution. The delta wing simulation helped to confirm the results of the blunt fin by showing coherent primary cores at 16% of solution convergence. These results showed that concurrent feature extraction from CFD data sets is possible.

Application of intelligent agents to fully converged data sets showed that a human estimate of the probability of extracted features could be made. The blunt fin simulation showed that the horseshoe core line was extracted by both the RP and SH algorithms so one algorithm's feature needed to be selected as most probable. Based on the strengths and weaknesses of each algorithm agents formed a final opinion at each point contained in the horseshoe cores. This final opinion aided in selecting the RP extracted core as the most probable. This decision corresponds with feature extraction literature as the RP algorithm is designed specifically to extract curved vortex cores. The fully converged delta wing simulation showed that the primary cores were extracted by both the RP and SH algorithms so a decision needed to be made between the two sets of primary cores. Based on a curvature value for SH of $Curvature = 0.100$ and a curvature value for RP of $Curvature = 0.095$ agents correctly selected the SH algorithm's primary cores as most probable.

Agents were then tested on their abilities to find the expected probability of features in converging data sets. When forming opinions on and making decisions about vortex core lines agents do not have any information about the fully converged simulation or any following iterations. Agents do have information about features extracted at previous iterations in the simulation. This means that agents can select features from converging data sets as being highly probable to be in the same spatial location at the end of the simulation. The blunt fin simulation showed that as early as 40% into the simulation the horseshoe core line was found to have an expected probability above 0.90 for most of the feature. This was a correct analysis by the agents as the feature at 40% was close to its final spatial position in the fully converged data set. The delta wing simulation showed that at 47% and 68% converged the extracted primary cores by the SH algorithm had expected probability values near 0.9 for the majority of the cores except at the downstream ends as the vortex strength and quality values were low. This expected probability was correct as the primary cores from the fully converged data set showed little to no spatial variation between the primary cores at 47% and 68% converged.

Subjective logic provides an effective vehicle for analysis of concurrent feature extraction. In concurrent feature extraction it is difficult to make a concrete statement about the convergence of extracted features such as yes the extracted features are spatially correct. Subjective logic provides three logic values so intermediate opinions can be made about converging features such as there is a high belief with some uncertainty and low disbelief giving a high expected probability. While it may be uncomfortable without a clear cut yes or no to extracted features, concurrent feature extraction is inherently a gray area. Features are in the process of converging. This grayness, rather than black and white, is effectively quantified with subjective logic. Also, because subjective logic does not give a clear cut yes or no for cores it can give a researcher some flexibility based on previous experience as to what cores to analyze and visualize.

When extracting features, specifically vortex cores, from converged data sets it is not always clear if extracted cores are vortices or if they are extraneous cores. Even when tracing streamlines to visualize flow rotation it can still be difficult. This is especially true for cores with weak rotation. Subjective logic can effectively present the expected probability of extracted cores based on their characteristics. The developed method does not take away the need to visualize a data set and to visualize extracted features but rather it can provide an effective starting point for visualization and possibly find highly probable features that may have gone unnoticed. The most effective starting point for flow visualization will be the feature with the highest expected probability then moving to the next feature with the highest expected probability.

A weakness of using the developed method is it can be cumbersome to set all of the values appropriately. There are three opinions which make for nine belief tuple values that need to be set before final opinions may be evaluated. For each of these belief tuple values there is usually a linear function that sets each value so the constants in the linear functions must be found. Once the constants are found then the value input into the constant must be found which includes more variables and equations. Luckily, most of the values to be found such as constants in the linear equations are constant across any given CFD data sets so once they are found they stay the same. With all of these values to be set it is difficult to find which values are influencing the outcome of the final opinion and which values have little influence on the final opinion.

The unique contributions of this research is a method to analyze CFD flow features in converging data sets. Previously there has not been a method to analyze features from converging data sets. Also, the method can combine feature sets created by two separate algorithms into one feature set containing only features with high expected probabilities. It has been a problem in vortex core extraction that algorithms were designed to extract features in specific flow conditions and did not produce acceptable results in other flow conditions. Also, the research provided some basic information on how features behaved in converging data sets. Features were shown to convect downstream as the solution converged and some features found their final spatial location before the overall CFD solution was converged.

REFERENCES

[1] Mortensen, C., Woodley, R., and Gorrell, S., 2009. "Concurrent Agent-enabled Extraction of Computational Fluid Dynamics (CFD) Features in Simulation." In *Proceedings of The 2009 International Conference on Data Mining*, pp. 90–96.

[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] 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." *ASME Paper GT2008-51220*, June.

[4] Thompson, D., Nair, J., Venkata, S., Machiraju, R., Jiang, M., and Craciun, G., 2002. "Physics-Based Feature Mining for Large Data Exploration." *IEEE Computing in Science & Engineering,* **4**(4), July, pp. 22–30.

[5] Post, F., Vrolijk, B., Hauser, H., Laramee, R., and Doleisch, H., 2003. "The State of the Art in Flow Visualisation: Feature Extraction and Tracking." *Computer Graphics Forum,* **22**(4), 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] Jøsang, A., 2001. "A Logic for Uncertain Probabilites." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems,* **9**(3), June, pp. 279–311.

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

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

[11] Robinson, S., 1989. "A Review of Vortex Structures and Associated Coherent Motions in Turbulent Boundary Layers." In *Proceedings of Second IUTAM Symposium on Structure of Turbulence and Drag Reduction*.

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

[13] 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.

[14] Levy, Y., Degani, D., and Seginer, A., 1990. "Graphical Visualization of Vortical Flows by Means of Helicity." *AIAA Journal,* **28**(8), pp. 1347–1352.

[15] Yates, L., and Chapman, G., 1991. "Streamlines, Vorticity Lines, and Vortices." *AIAA 91-0731.*

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

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

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

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

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

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

[22] Jiang, M., Machiraju, R., and Thompson, D., 2002. "A Novel Approach to Vortex Core Region Detection." In *VISSYM '02: Proceedings of the Symposium on Data Visualisation 2002*, Eurographics Association, pp. 217–225.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] NASA, 2010. Sample Curvilinear Mesh, CFD Datasets `http://www.nas.nasa.gov/Resources/datasets.html`, June.

[31] Hung, C., and Buning, P., 1985. "Simulation of Blunt-fin-induced Shock-wave and Turbulent Boundary-layer Interaction." *Journal of Fluid Mechanics,* **154**, pp. 163–185.

[32] Dolling, D., Cosad, C., and S., B., 1979. "An Examination of Blunt Fin-induced Shock Wave Turbulent Boundary Layer Interactions." *AIAA 79-0068*.

[33] ANSYS, 2006. *FLUENT 6.3 User's Guide*.

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

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

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

A.1 User's Guide

The code that runs the intelligent vortex core extraction and includes the 'main' method is contained in Section A.2. The easiest way to go through the code will be to walk through it step-by-step. Before this code will compile the VTK 5.4 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 9.10 (Karmic Koala) using g++ and cmake 2.6 to create make files.

The #include statements on lines 1–38 include all libraries required for execution of the code. The system call on line 48 is used to remove the files that the program writes and will throw an error that doesn't matter if there are no files in the directory to delete. On lines 50–56 some values are defined that will be used as inputs to objects later. In lines 60–67 an array iterations is set up to hold each iteration number that data sets have been saved for.

On line 70 the for loop is started which performs the actual feature extraction using the Sujudi-Haimes and Roth-Peikert extraction algorithms. What this for loop does is read in data sets that are saved to the hard disk (currently the data sets are Fluent case files), then put the data sets into the Roth-Peikert and Sujudi-Haimes algorithms where feature sets are created containing only vortex core lines, these feature sets are then saved to disk and the unused objects are deleted. The extraction algorithms have been provided by Dr. Rhonda Vickery and John van der Zwaag.

Lines 72–81 of the for loop define strings for the names of the files that are to be read in and the files that are to be saved to disk later. Lines 85–87 are the vtkFLUENTReader that read in the saved Fluent data sets to a vtkUnstructuredGrid object. If other data sets besides Fluent are saved to disk then the appropriate vtk reader can be substituted into these lines. Lines 91–93 just change the vtkUnstructuredGrid as having cell data to point data. When the Fluent data file is read in the data is read in as cell data rather than point data so it needs to be changed because the feature extraction algorithms work with point data rather than cell data. Lines 96–106 use the class vtkArrayCalculator to add an array to the vtkUnstructuredGrid called 'Velocity' that is an array with three components. This is done because as input into the vtkRothPeikert and vtkSujudiHaimes classes a velocity vector needs to be input rather than three scalar velocity components. Lines 109–116 remove extraneous arrays that are not needed. Lines 121–127 instantiate and input the data object into the Roth-Peikert vortex core extraction algorithm. This class takes an unstructured grid as an input and outputs a poly data set which is composed of polylines. Lines 136–139 delete unused objects. Lines 144–150 instantiate and input the data object into the Sujudi-Haimes vortex core extraction algorithm. This class takes an unstructured grid as an input and outputs a poly data set which is composed of polylines. Lines 130–133 and 153–156 write the extracted core lines to file and lines 159–161 delete the rest of the unused objects. This completes the for loop.

Lines 166–168 really don't matter. I put them in there because at the time I was wondering how long it was taking the code to complete and about how much time left the code would take to complete when it was running. These lines correspond with the other lines that use the object stopWatch.

The next for loop starts on line 171. This for loop takes the feature sets created from the first for loop, cleans the data sets, computes all the values needed to form the final opinion, computes the final opinion and then writes the data sets to disk. The for loop in this case ends at 10 because there are only ten data sets. The condition i<11 should be changed to reflect how many data sets are currently available to process. The for loop starts at i=2 because the first feature set can't form an opinion because there is no previous data set. The condition on line 175 if(i==2) is there because even the second feature set can't form a final opinion but certain values such as feature displacement need to be calculated in order for the third feature set to form its final opinion. For this condition the Sujudi-Haimes data sets are read in using the vtkPolyDataReader, they are cleaned using vtkCleanPolyData, the quality is computed using vtkQuality, each of the lines is paramaterized from 0 to 1 using vtkParamaterizeLineFilter, the feature displacement is computed using vtkFeatureDisplacement and then the feature sets are saved to disk. The feature sets need to be cleaned because when they come out of the classes vtkRothPeikert and vtkSujudiHaimes sometimes there are stray points that are not core lines that need to be removed. This procedure is then followed for the Roth-Peikert feature sets.

The else statement is for all feature sets past the second feature set. It is fairly similar to the i==2 condition except for change in feature displacement is calculated and final opinions are calculated. So lines 377–392 deal with getting and setting the proper file names to read and then write files at the end of the else statement. Lines 394–457 read in the Sujudi-Haimes feature sets, clean the newest feature set (this would be 3 if i=3, 4 if i=4 etc.), computes the quality of the newest feature set, paramaterizes the newest feature set, computes the curvature of the newest feature set, finds same lines for the newest feature set, calculates the feature displacement and change in feature displacement and then deletes all of the unused objects. Lines 461–512 read in the Roth-Peikert feature sets, cleans, computes quality for, paramaterizes, computes curvature for, finds same lines for, calculates feature displacment and change in feature displacement for the newest feature set. Lines 516–526 find the minimum distance between points in feature sets once for Sujudi-Haimes and once for Roth-Peikert. Lines 529–538 is where everything is all tied together and a final opinion is computed for each point contained in each vortex core line. Then lines 541–550 write the complete feature sets to file and the remaining lines delete objects and deal with timing the code.

A.2 Source Code

```
1  #include <vtkPLOT3DReader.h>
2  #include <vtkStructuredGrid.h>
3  #include <vtkPolyDataReader.h>
4  #include <vtkPolyData.h>
5  #include <vtkPolyDataWriter.h>
6  #include <vtkAppendPolyData.h>
7  #include <vtkFLUENTReader.h>
8  #include <vtkMultiBlockDataSet.h>
9  #include <vtkCellDataToPointData.h>
10 #include <vtkArrayCalculator.h>
11 #include <vtkVectorsGradientFilter.h>
12 #include <vtkParallelVectors.h>
13 #include <vtkCallbackCommand.h>
14 #include <vtkVortexStrength.h>
15 #include <vtkThresholdPoints.h>
16 #include <vtkConnectLines.h>
17 #include <vtkDoubleArray.h>
18 #include <vtkPointData.h>
19 #include <vtkMath.h>
20 #include <vtkCell.h>
21 #include "vtkParamaterizeLineFilter.h"
22 #include "vtkCurvature.h"
23 #include "vtkFeatureDisplacement.h"
24 #include "vtkSameLine.h"
25 #include "vtkQuality.h"
26 #include "vtkMinimumDistance.h"
27 #include "vtkCreateOpinion.h"
28 #include "vtkRothPeikert.h"
29 #include "vtkSujudiHaimes.h"
30 #include <vtkGradientFilter.h>
31 #include "vtkExtractCells.h"
32 #include "vtkUnstructuredGrid.h"
33 #include "vtkPolyDataConnectivityFilter.h"
34 #include "vtkCleanPolyData.h"
35 #include <math.h>
36 #include <sstream>
37 #include <iostream>
38 #include "hr_time.h"
39
40 using namespace std;
41
```

```cpp
int main()
{

//---------------------------------------------------------------------------------

    // removing unneeded files
    system("rm ./dataSets/deltaWing/Complete*");

    bool verbose = true;
    int cpu = 4;
    double vortexStrengthThreshold = 10;
    double qualityThresholdValue = 40;  // Roth says this value is typically
                                        // between 30 and 45 degrees.
    bool thresholdLines = true;
    int minimumCorePoints = 5;  //min value 2

//---------------------------------------------------------------------------------

    int iterations[10];
    iterations[0] = 100;
    cout << iterations[0] << endl;
    int q;
    for(q=1 ; q<10 ; q++){
        iterations[q] = iterations[q-1] + 200;
        cout << iterations[q] << "\n" ;
    }

    int j;
   for(j=9 ; j<10 ; j++){

        // getting correct names for the files
        string inputFileName, outputFileNameSH, outputFileNameRP;
        stringstream out;

        out << "./dataSets/MattsGrids/deltaWing/delta_10_deg_iter_" << iterations[j] << ".cas" << endl;
        getline(out, inputFileName);
        out << "./dataSets/MattsGrids/deltaWing/delta_10_deg_iter_" << iterations[j] << "_RP.vtk" << endl;
        getline(out, outputFileNameRP);
        out << "./dataSets/MattsGrids/deltaWing/delta_10_deg_iter_" << iterations[j] << "_SH.vtk" << endl;
        getline(out, outputFileNameSH);

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

        // Changing Fluent's cell data to point data
        vtkCellDataToPointData *c2p = vtkCellDataToPointData::New();
        c2p->SetInput(fluent->GetOutput()->GetBlock(0));
        c2p->Update();

        // Creating the 'Velocity' array
        vtkArrayCalculator *arrayCalc = 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("Velocity");
        arrayCalc->SetFunction("iHat*(X_Velocity) +"
                               "jHat*(Y_Velocity) +"
                             "kHat*(Z_Velocity)");
        arrayCalc->SetInput(c2p->GetOutput());
        arrayCalc->SetAttributeModeToUsePointData();
        arrayCalc->Update();
```

```
        // removing unrequired arrays
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("X_VELOCITY");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("Y_VELOCITY");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("Z_VELOCITY");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("NUT");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("BODY_FORCES");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("MU_LAM");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("MU_TURB");
        arrayCalc->GetOutput()->GetPointData()->RemoveArray("WALL_DIST");

        // 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.
        vtkRothPeikert *rothPeikert = vtkRothPeikert::New();
        rothPeikert->SetInput(arrayCalc->GetOutput());
        rothPeikert->SetVelocityArrayName("Velocity");
        rothPeikert->SetVortexStrengthThreshold(vortexStrengthThreshold);
        rothPeikert->SetMinimumNumberOfPoints(minimumCorePoints);
        rothPeikert->SetVerbose(verbose);
        rothPeikert->Update();

        // writing the connected lines to RP.vtk
        vtkPolyDataWriter *writer2 = vtkPolyDataWriter::New();
        writer2->SetInput(rothPeikert->GetOutput());
        writer2->SetFileName(outputFileNameRP.c_str());
        writer2->Write();

        // deleting unused objects
        fluent->Delete();
        c2p->Delete();
        rothPeikert->Delete();
        writer2->Delete();

        // 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.
        vtkSujudiHaimes *sujudiHaimes = vtkSujudiHaimes::New();
        sujudiHaimes->SetInput(arrayCalc->GetOutput());
        sujudiHaimes->SetVelocityArrayName("Velocity");
        sujudiHaimes->SetVortexStrengthThreshold(vortexStrengthThreshold);
        sujudiHaimes->SetMinimumNumberOfPoints(minimumCorePoints);
        sujudiHaimes->SetVerbose(verbose);
        sujudiHaimes->Update();

        // writing extracted lines from Sujud-Haimes
        vtkPolyDataWriter *writer3 = vtkPolyDataWriter::New();
        writer3->SetInput(sujudiHaimes->GetOutput());
        writer3->SetFileName(outputFileNameSH.c_str());
        writer3->Write();

        // deleting unused objects
        arrayCalc->Delete();
        sujudiHaimes->Delete();
        writer3->Delete();
}

//-----------------------------------------------------------------------------

    CStopWatch stopWatch = CStopWatch::CStopWatch();
    stopWatch.startTimer();
    double timeToCompletion, oldTime;

    int i;
   for(i=2 ; i<11 ; i++){ //**need to change these i values so they can accurately reflect how
        many data sets we have :p

        // for i==2 we don't need to use subjective logic because belief, disbelief,
```

```
        // and uncertainty values can't be computed until the third data set.
        if(i==2){

        // getting correct names for the files
        string activeFileNameSH, passiveFileNameSH, outputFileNameSH,
                activeFileNameRP, passiveFileNameRP, outputFileNameRP;
        stringstream out;
        out << "./dataSets/deltaWing/" << i << "SH.vtk" << endl;
        getline(out, activeFileNameSH);
        out << "./dataSets/deltaWing/" << i-1 << "SH.vtk" << endl;
        getline(out, passiveFileNameSH);
        out << "./dataSets/deltaWing/Complete" << i << "SH.vtk" << endl;
        getline(out, outputFileNameSH);

        out << "./dataSets/deltaWing/" << i << "RP.vtk" << endl;
        getline(out, activeFileNameRP);
        out << "./dataSets/deltaWing/" << i-1 << "RP.vtk" << endl;
        getline(out, passiveFileNameRP);
        out << "./dataSets/deltaWing/Complete" << i << "RP.vtk" << endl;
        getline(out, outputFileNameRP);

        ////****Sujudi-Haimes Section****////
        // Reading in Sujudi-Haimes vortex core lines
        vtkPolyDataReader *polyReader1 = vtkPolyDataReader::New();
        polyReader1->SetFileName(activeFileNameSH.c_str());
        polyReader1->Update();

        // Reading in Sujudi-Haimes vortex core lines from previous extraction
        vtkPolyDataReader *polyReader2 = vtkPolyDataReader::New();
        polyReader2->SetFileName(passiveFileNameSH.c_str());
        polyReader2->Update();

        // cleaning the input data set
        vtkCleanPolyData *clean1 = vtkCleanPolyData::New();
        clean1->SetInput(polyReader1->GetOutput());
        clean1->Update();

        // cleaning the input data set
        vtkCleanPolyData *clean2 = vtkCleanPolyData::New();
        clean2->SetInput(polyReader2->GetOutput());
        clean2->Update();

        // Computing the quality of the vortices
        vtkQuality *quality1 = vtkQuality::New();
        quality1->SetInput(clean1->GetOutput());
        quality1->SetThresholdLines(thresholdLines);
        quality1->SetQualityThresholdValue(qualityThresholdValue);
        quality1->Update();

        // Computing the quality of the vortices
        vtkQuality *quality2 = vtkQuality::New();
        quality2->SetInput(clean2->GetOutput());
        quality2->SetThresholdLines(thresholdLines);
        quality2->SetQualityThresholdValue(qualityThresholdValue);
        quality2->Update();

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

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

```
// finding same lines in other data set for previous error filter
vtkSameLine *sameLine1 = vtkSameLine::New();
sameLine1->AddInputConnection(plf1->GetOutputPort());
sameLine1->AddInputConnection(plf2->GetOutputPort());
sameLine1->Update();
vtkIntArray *sameLineArray1 = sameLine1->GetSameLine();

// Computing previous error of the data set
vtkFeatureDisplacement *pe1 = vtkFeatureDisplacement::New();
// the first input is the input that the previous error is calculated for
// i.e. newer data set
pe1->AddInputConnection(plf1->GetOutputPort());
// the second input is used to calculate previous error for the first input
// i.e. older data set
pe1->AddInputConnection(plf2->GetOutputPort());
pe1->SetSameLineArray(sameLineArray1);
pe1->ComputeChangeInErrorOff();
pe1->ClosestPointOn();
pe1->Update();

// Writing file to check it
vtkPolyDataWriter *pdWriter1 = vtkPolyDataWriter::New();
pdWriter1->SetInput(pe1->GetOutput());
pdWriter1->SetFileName(outputFileNameSH.c_str());
pdWriter1->Write();

// deleting unused objects
polyReader1->Delete();
polyReader2->Delete();
clean1->Delete();
clean2->Delete();
quality1->Delete();
quality2->Delete();
plf1->Delete();
plf2->Delete();
sameLine1->Delete();
pe1->Delete();
pdWriter1->Delete();
sameLineArray1->Delete();

////****Roth-Peikert Section****////
// Reading in Roth-Peikert vortex core lines
vtkPolyDataReader *polyReader3 = vtkPolyDataReader::New();
polyReader3->SetFileName(activeFileNameRP.c_str());
polyReader3->Update();

// Reading in Roth-Peikert vortex core lines from previous extraction
vtkPolyDataReader *polyReader4 = vtkPolyDataReader::New();
polyReader4->SetFileName(passiveFileNameRP.c_str());
polyReader4->Update();

// cleaning the input data set
vtkCleanPolyData *clean3 = vtkCleanPolyData::New();
clean3->SetInput(polyReader3->GetOutput());
clean3->Update();

// cleaning the input data set
vtkCleanPolyData *clean4 = vtkCleanPolyData::New();
clean4->SetInput(polyReader4->GetOutput());
clean4->Update();

// Computing the quality of the vortices
vtkQuality *quality3 = vtkQuality::New();
quality3->SetInput(clean3->GetOutput());
quality3->SetThresholdLines(thresholdLines);
quality3->SetQualityThresholdValue(qualityThresholdValue);
quality3->Update();
```

```
        // Computing the quality of the vortices
        vtkQuality *quality4 = vtkQuality::New();
        quality4->SetInput(clean4->GetOutput());
        quality4->SetThresholdLines(thresholdLines);
        quality4->SetQualityThresholdValue(qualityThresholdValue);
        quality4->Update();

        // Paramaterizing line segments
        // each line segment has an a,b,c,d,e,f and l associated value
        vtkParamaterizeLineFilter *plf3 = vtkParamaterizeLineFilter::New();
        plf3->SetInput(quality3->GetOutput());
        plf3->Update();

        // Paramaterizing line segments
        // each line segment has an a,b,c,d,e,f and l associated value
        vtkParamaterizeLineFilter *plf4 = vtkParamaterizeLineFilter::New();
        plf4->SetInput(quality4->GetOutput());
        plf4->Update();

        // finding same lines in other data set for previous error filter
        vtkSameLine *sameLine2 = vtkSameLine::New();
        sameLine2->AddInputConnection(plf3->GetOutputPort());
        sameLine2->AddInputConnection(plf4->GetOutputPort());
        sameLine2->Update();
        vtkIntArray *sameLineArray2 = sameLine2->GetSameLine();

        // Computing previous error of the data set
        vtkFeatureDisplacement *pe2 = vtkFeatureDisplacement::New();
        // the first input is the input that the previous error is calculated for
        // i.e. newer data set
        pe2->AddInputConnection(plf3->GetOutputPort());
        // the second input is used to calculate previous error for the first input
        // i.e. older data set
        pe2->AddInputConnection(plf4->GetOutputPort());
        pe2->SetSameLineArray(sameLineArray2);
        pe2->ComputeChangeInErrorOff();
        pe2->Update();

        // Writing file to check it
        vtkPolyDataWriter *pdWriter2 = vtkPolyDataWriter::New();
        pdWriter2->SetInput(pe2->GetOutput());
        pdWriter2->SetFileName(outputFileNameRP.c_str());
        pdWriter2->Write();

        // deleting unused objects
        polyReader3->Delete();
        polyReader4->Delete();
        clean3->Delete();
        clean4->Delete();
        quality3->Delete();
        quality4->Delete();
        plf3->Delete();
        plf4->Delete();
        sameLine2->Delete();
        pe2->Delete();
        pdWriter2->Delete();
        sameLineArray2->Delete();

        stopWatch.stopTimer();
        oldTime = stopWatch.getElapsedTime();
        cout << "Completed " << i << " in Time = " << stopWatch.getElapsedTime() << " s" << endl;
        }
////////////////////////////////////////////////////////////////////////////////

        else{

        // getting correct names for the files
        string activeFileNameSH, passiveFileNameSH, outputFileNameSH,
```

```
            activeFileNameRP, passiveFileNameRP, outputFileNameRP;
    stringstream out;
    out << "./dataSets/deltaWing/" << i << "SH.vtk" << endl;
    getline(out, activeFileNameSH);
    out << "./dataSets/deltaWing/Complete" << i-1 << "SH.vtk" << endl;
    getline(out, passiveFileNameSH);
    out << "./dataSets/deltaWing/Complete" << i << "SH.vtk" << endl;
    getline(out, outputFileNameSH);

    out << "./dataSets/deltaWing/" << i << "RP.vtk" << endl;
    getline(out, activeFileNameRP);
    out << "./dataSets/deltaWing/Complete" << i-1 << "RP.vtk" << endl;
    getline(out, passiveFileNameRP);
    out << "./dataSets/deltaWing/Complete" << i << "RP.vtk" << endl;
    getline(out, outputFileNameRP);

    ////****Sujudi-Haimes Section****////
    // Reading in Sujudi-Haimes vortex core lines
    vtkPolyDataReader *polyReader1 = vtkPolyDataReader::New();
    polyReader1->SetFileName(activeFileNameSH.c_str());
    polyReader1->Update();

    // Reading in another data set
    vtkPolyDataReader *polyReader2 = vtkPolyDataReader::New();
    polyReader2->SetFileName(passiveFileNameSH.c_str());
    polyReader2->Update();

    // cleaning the input data set
    vtkCleanPolyData *clean1 = vtkCleanPolyData::New();
    clean1->SetInput(polyReader1->GetOutput());
    clean1->Update();

    // Computing the quality of the vortices
    vtkQuality *quality1 = vtkQuality::New();
    quality1->SetInput(clean1->GetOutput());
    quality1->SetThresholdLines(thresholdLines);
    quality1->SetQualityThresholdValue(qualityThresholdValue);
    quality1->Update();

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

    // calculating the curvature of the line
    vtkCurvature *curvature1 = vtkCurvature::New();
    curvature1->SetInput(plf1->GetOutput());
    curvature1->SingleCurvatureValueOn();
    curvature1->TwoSegmentCurvatureOff();
    curvature1->Update();

    // finding same lines in other data set for previous error filter
    vtkSameLine *sameLine1 = vtkSameLine::New();
    sameLine1->AddInputConnection(curvature1->GetOutputPort());
    sameLine1->AddInputConnection(polyReader2->GetOutputPort());
    sameLine1->Update();
    vtkIntArray *sameLineArray1 = sameLine1->GetSameLine();

    // Computing previous error of the data set
    vtkFeatureDisplacement *pe1 = vtkFeatureDisplacement::New();
    // the first input is the input that the previous error is calculated for
    // i.e. newer data set
    pe1->AddInputConnection(curvature1->GetOutputPort());
    // the second input is used to calculate previous error for the first input
    // i.e. older data set
    pe1->AddInputConnection(polyReader2->GetOutputPort());
    pe1->SetSameLineArray(sameLineArray1);
```

```
        pe1->ComputeChangeInErrorOn();
        pe1->Update();

        // deleting unused objects
        polyReader1->Delete();
        polyReader2->Delete();
        clean1->Delete();
        quality1->Delete();
        plf1->Delete();
        curvature1->Delete();
        sameLine1->Delete();
        sameLineArray1->Delete();

        ////****Roth-Peikert Section****////
        // Reading in Roth-Peikert vortex core lines
        vtkPolyDataReader *polyReader3 = vtkPolyDataReader::New();
        polyReader3->SetFileName(activeFileNameRP.c_str());
        polyReader3->Update();

        // Reading in another data set
        vtkPolyDataReader *polyReader4 = vtkPolyDataReader::New();
        polyReader4->SetFileName(passiveFileNameRP.c_str());
        polyReader4->Update();

        // cleaning the input data set
        vtkCleanPolyData *clean3 = vtkCleanPolyData::New();
        clean3->SetInput(polyReader3->GetOutput());
        clean3->Update();

        // Computing the quality of the vortices
        vtkQuality *quality3 = vtkQuality::New();
        quality3->SetInput(clean3->GetOutput());
        quality3->SetThresholdLines(thresholdLines);
        quality3->SetQualityThresholdValue(qualityThresholdValue);
        quality3->Update();

        // Paramaterizing line segments
        // each line segment has an a,b,c,d,e,f and l associated value
        vtkParamaterizeLineFilter *plf3 = vtkParamaterizeLineFilter::New();
        plf3->SetInput(quality3->GetOutput());
        plf3->Update();

        // calculating the curvature of the line
        vtkCurvature *curvature3 = vtkCurvature::New();
        curvature3->SetInput(plf3->GetOutput());
        curvature3->SingleCurvatureValueOn();
        curvature3->TwoSegmentCurvatureOff();
        curvature3->Update();

        // finding same lines in other data set for previous error filter
        vtkSameLine *sameLine2 = vtkSameLine::New();
        sameLine2->AddInputConnection(curvature3->GetOutputPort());
        sameLine2->AddInputConnection(polyReader4->GetOutputPort());
        sameLine2->Update();
        vtkIntArray *sameLineArray2 = sameLine2->GetSameLine();

        // Computing previous error of the data set
        vtkFeatureDisplacement *pe2 = vtkFeatureDisplacement::New();
        // the first input is the input that the previous error is calculated for
        // i.e. newer data set
        pe2->AddInputConnection(curvature3->GetOutputPort());
        // the second input is used to calculate previous error for the first input
        // i.e. older data set
        pe2->AddInputConnection(polyReader4->GetOutputPort());
        pe2->SetSameLineArray(sameLineArray2);
        pe2->ComputeChangeInErrorOn();
        pe2->Update();

```

```
        // Computing minimum distance between points in Sujudi-Haimes data set
        // and points in Roth-Peikert data set.
        vtkMinimumDistance *minimumDistance1 = vtkMinimumDistance::New();
        minimumDistance1->AddInputConnection(pe1->GetOutputPort());
        minimumDistance1->AddInputConnection(pe2->GetOutputPort());
        minimumDistance1->Update();

        // Computing minimum distance between points in Roth-Peikert data set
        // and points in Sujudi-Haimes data set.
        vtkMinimumDistance *minimumDistance2 = vtkMinimumDistance::New();
        minimumDistance2->AddInputConnection(pe2->GetOutputPort());
        minimumDistance2->AddInputConnection(pe1->GetOutputPort());
        minimumDistance2->Update();

        // Creating the final opinion of the data set
        vtkCreateOpinion *createOpinion1 = vtkCreateOpinion::New();
        createOpinion1->SetInput(minimumDistance1->GetOutput());
        createOpinion1->SujudiHaimesOn();
        createOpinion1->Update();

        // Creating the final opinion of the data set
        vtkCreateOpinion *createOpinion2 = vtkCreateOpinion::New();
        createOpinion2->SetInput(minimumDistance2->GetOutput());
        createOpinion2->RothPeikertOn();
        createOpinion2->Update();

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

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

        // deleting unused objects
        polyReader3->Delete();
        polyReader4->Delete();
        clean3->Delete();
        quality3->Delete();
        plf3->Delete();
        curvature3->Delete();
        sameLine2->Delete();
        pe2->Delete();
        pdWriter2->Delete();
        sameLineArray2->Delete();

            stopWatch.stopTimer();
            timeToCompletion = (stopWatch.getElapsedTime()-oldTime)*(10-i);
            cout << "Completed " << i << " in Time = " << stopWatch.getElapsedTime()-oldTime << " s"
                 << "\tElapsed Time = " << stopWatch.getElapsedTime() << "\tETA = " <<
                timeToCompletion << " s" << endl;
            oldTime = stopWatch.getElapsedTime();

        }
////////////////////////////////////////////////////////////////////////////////

    }

  return 1;
}
```

A.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 5.4 code as superclasses. Two books from Kitware, Inc. explain the vtk object structure [34, 35]. The header files are listed in alphabetical order.

A.3.1 vtkCreateOpinion.h

```
// .NAME vtkCreateOpinion

// .SECTION Description
// vtkCreateOpinion is a filter that computes the final opinion.

#ifndef __vtkCreateOpinion_h
#define __vtkCreateOpinion_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkCreateOpinion : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkCreateOpinion,vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkCreateOpinion *New();

  // Description: Set/Get constant used to find belief,
   // disbelief, and uncertainty values for Master Agent.
  vtkSetMacro(PreviousErrorConstant, double);
  vtkGetMacro(PreviousErrorConstant, double);

  // Description: Set/Get constant used to find belief,
   // disbelief, and uncertainty values for Master Agent.
  vtkSetMacro(ChangeInErrorConstant, double);
  vtkGetMacro(ChangeInErrorConstant, double);

   // 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);

  // Description: Set/Get largest vortex strength value which
   // divides all the vortex strength values.
```

```
  vtkSetMacro(VortexStrengthMax, double);
  vtkGetMacro(VortexStrengthMax, double);

  // Description: Set/Get largest curvature value which
   // divides all the curvature values.
  vtkSetMacro(CurvatureMax, double);
  vtkGetMacro(CurvatureMax, double);

  // Description: Set/Get largest quality value which
   // divides all the quality values.
  vtkSetMacro(QualityMax, double);
  vtkGetMacro(QualityMax, double);

  // Description: Set/Get largest quality value which
   // divides all the quality values.
  vtkSetMacro(MinimumDistanceMax, double);
  vtkGetMacro(MinimumDistanceMax, double);

protected:
  vtkCreateOpinion();
  ~vtkCreateOpinion() {};

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

   double PreviousErrorConstant;
   double ChangeInErrorConstant;
   double VortexStrengthMax;
   double CurvatureMax;
   double QualityMax;
   double MinimumDistanceMax;
   int    SujudiHaimes;
   int    RothPeikert;

private:
  vtkCreateOpinion(const vtkCreateOpinion&);     // Not implemented.
  void operator=(const vtkCreateOpinion&);   // Not implemented.
};

#endif
```

A.3.2 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 VTK_GRAPHICS_EXPORT vtkCurvature : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkCurvature,vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkCurvature *New();
```

```

  // Description:
  // Set number of points to calculate curvature.
  vtkSetMacro(NumberOfCurvatureValues,int);
  vtkGetMacro(NumberOfCurvatureValues,int);

  // Description:
  // Turn on/off calculating curvature using line endpoint,
  // midpoint and startpoint.
  vtkSetMacro(SingleCurvatureValue,     int);
  vtkGetMacro(SingleCurvatureValue,     int);
  vtkBooleanMacro(SingleCurvatureValue, int);  //false is 0

  // Description:
  // Turn on/off the calculation of curvature for a central
  // point using a three point approximation.
  vtkSetMacro(TwoSegmentCurvature,     int);
  vtkGetMacro(TwoSegmentCurvature,     int);
  vtkBooleanMacro(TwoSegmentCurvature, int);  //false is 0

protected:
  vtkCurvature();
  ~vtkCurvature() {};

  int TwoSegmentCurvature;
  int SingleCurvatureValue;
   int NumberOfCurvatureValues;

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

private:
  vtkCurvature(const vtkCurvature&);     // Not implemented.
  void operator=(const vtkCurvature&);   // Not implemented.
};

#endif
```

A.3.3 vtkMinimumDistance.h

```
// .NAME vtkMinimumDistance

// .SECTION Description
// vtkMinimumDistance is a filter that computes the smallest cartesian
// distance between each point in input1 and all the points in input2.

#ifndef __vtkMinimumDistance_h
#define __vtkMinimumDistance_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkMinimumDistance : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkMinimumDistance,vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkMinimumDistance *New();

protected:
  vtkMinimumDistance();
```

```
  ~vtkMinimumDistance() {};

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
   int FillInputPortInformation( int port, vtkInformation* info);

private:
  vtkMinimumDistance(const vtkMinimumDistance&);      // Not implemented.
  void operator=(const vtkMinimumDistance&);          // Not implemented.
};

#endif
```

A.3.4 vtkFeatureDisplacement.h

```
// .NAME vtkFeatureDisplacement - computes feature displacement

// .SECTION Description
// vtkFeatureDisplacement is a filter that computes the feature displacement
// for each line in the first data set. The feature displacement is computed
// based on the closest line in the second data set.

#ifndef __vtkFeatureDisplacement_h
#define __vtkFeatureDisplacement_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkFeatureDisplacement : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkFeatureDisplacement,vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkFeatureDisplacement *New();

   vtkSetMacro(SameLineArray, vtkIntArray*);

   // turning on/off the computation of the change in error array
   vtkSetMacro(ComputeChangeInError,      int);
   vtkGetMacro(ComputeChangeInError,      int);
   vtkBooleanMacro(ComputeChangeInError, int);

   // turning on/off the computation of previous error using closest point
   vtkSetMacro(ClosestPoint,      int);
   vtkGetMacro(ClosestPoint,      int);
   vtkBooleanMacro(ClosestPoint, int);

   // turning on/off the computation of previous error using same line
   vtkSetMacro(SameLine,      int);
   vtkGetMacro(SameLine,      int);
   vtkBooleanMacro(SameLine, int);

protected:
  vtkFeatureDisplacement();
  ~vtkFeatureDisplacement() {};

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
   int FillInputPortInformation( int port, vtkInformation* info);

   vtkIntArray *SameLineArray;
```

```
    int ComputeChangeInError;
    int ClosestPoint;
    int SameLine;

private:
  vtkFeatureDisplacement(const vtkFeatureDisplacement&);      // Not implemented.
  void operator=(const vtkFeatureDisplacement&);    // Not implemented.
};

#endif
```

A.3.5 vtkQuality.h

```
// .NAME vtkQuality

// .SECTION Description
// vtkQuality is a filter that computes the vortex quality
// at each point of a vortex core line.

#ifndef __vtkQuality_h
#define __vtkQuality_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkQuality : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkQuality,vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkQuality *New();

  // Description:
  // Turn on/off quality thresholding.
  vtkSetMacro(ThresholdLines,      int);
  vtkGetMacro(ThresholdLines,      int);
  vtkBooleanMacro(ThresholdLines, int);  //false is 0

  // Description:
  // Set/Get value for quality threshold.
  vtkSetMacro(QualityThresholdValue, double);
  vtkGetMacro(QualityThresholdValue, double);

protected:
  vtkQuality();
  ~vtkQuality() {};

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);

    int ThresholdLines;
    double QualityThresholdValue;

private:
  vtkQuality(const vtkQuality&);      // Not implemented.
  void operator=(const vtkQuality&);    // Not implemented.
};

#endif
```

A.3.6 vtkSameLine.h

```
// .NAME vtkSameLine - locates closest line in separate data set

// .SECTION Description
// vtkSameLine is a filter that locates the closest line in the second
// data set to all lines in the first data set

#ifndef __vtkSameLine_h
#define __vtkSameLine_h

#include "vtkPolyDataAlgorithm.h"

class vtkFloatArray;
class vtkIdList;
class vtkPolyData;

class VTK_GRAPHICS_EXPORT vtkSameLine : public vtkPolyDataAlgorithm
{
public:
  vtkTypeRevisionMacro(vtkSameLine, vtkPolyDataAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkSameLine *New();

   vtkGetMacro(SameLine, vtkIntArray*);

protected:
  vtkSameLine();
  ~vtkSameLine() {};

  // Usual data generation method
  int RequestData(vtkInformation *, vtkInformationVector **, vtkInformationVector *);
   int FillInputPortInformation( int port, vtkInformation* info);

   vtkIntArray *SameLine;

private:
  vtkSameLine(const vtkSameLine&);     // Not implemented.
  void operator=(const vtkSameLine&);   // Not implemented.
};

#endif
```

A.4 Source Files

In this section source files are listed for each of the header files in Section A.3. Source files have not been listed for code I did not write like vtkRothPeikert and vtkSujudiHaimes. All of the code uses VTK 5.4 code as superclasses. The source files are listed in alphabetical order.

A.4.1 vtkCreateOpinion.cxx

```
#include "vtkCreateOpinion.h"

```

```
#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include "vtkMath.h"
#include "vtkThreshold.h"
#include "vtkUnstructuredGrid.h"
#include "vtkGeometryFilter.h"
#include <math.h>

vtkCxxRevisionMacro(vtkCreateOpinion, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkCreateOpinion);

//----------------------------------------------------------------------------
vtkCreateOpinion::vtkCreateOpinion()
{
   this->PreviousErrorConstant = 0.02;
   this->ChangeInErrorConstant = 0.05;
   this->SujudiHaimes          = true;
   this->RothPeikert           = false;
   this->VortexStrengthMax     = 600; // I like 600
   this->CurvatureMax          = 0.3; // I like 0.3
   this->QualityMax            = 80; // I like 80
   this->MinimumDistanceMax    = 0.1;
}

//----------------------------------------------------------------------------
int vtkCreateOpinion::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 Master Agent opinion array
   vtkDoubleArray *MAArray = 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
   vtkDoubleArray *AARPArray = 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
   vtkDoubleArray *AASHArray = vtkDoubleArray::New();
   AASHArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
   AASHArray->SetNumberOfComponents(3);
   AASHArray->SetNumberOfTuples(input->GetNumberOfPoints());
   AASHArray->SetName("AASH");

   // Creating array to store final opinion
```

```
    vtkDoubleArray *finalOpinionArray = vtkDoubleArray::New();
    finalOpinionArray->SetNumberOfValues(input->GetNumberOfPoints()*3);
    finalOpinionArray->SetNumberOfComponents(3);
    finalOpinionArray->SetNumberOfTuples(input->GetNumberOfPoints());
    finalOpinionArray->SetName("FinalOpinion");

    // Creating array to store probability expectation value
    vtkDoubleArray *probExpArray = vtkDoubleArray::New();
    probExpArray->SetNumberOfValues(input->GetNumberOfPoints());
    probExpArray->SetNumberOfComponents(1);
    probExpArray->SetNumberOfTuples(input->GetNumberOfPoints());
    probExpArray->SetName("ProbabilityExpectation");

    // Belief is set based on previous error and change in error
    // a small previous error and small change in error yields
    // belief values of approximately one.
    // Disbelief is set based on previous error. A small previous
    // error yields a low disbelief.
    // Uncertainty is set based on change in error. A small
    // change in error yields a low uncertainty.
    double b, d, u, CE, PE, tupleCheck, equalizer;
    int i;
    for(i=0 ; i<input->GetNumberOfPoints() ; i++){
      PE = input->GetPointData()->GetArray("PreviousError")->GetComponent(i,0);
      CE = input->GetPointData()->GetArray("ChangeInError")->GetComponent(i,0);
      b  = (-ChangeInErrorConstant * CE - PreviousErrorConstant * PE)/2 + 1;
      if(b<0){b = 0;}
      d  = PreviousErrorConstant * PE;
      if(d>1){d = 1;}
      u  = ChangeInErrorConstant * CE;
      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);
      // cout << "b = " << b << "\td = " << d << "\tu = " << u << "\tSum = " << b+d+u << endl;
    }

    // initializing variables
    double vortexStrength, curvature, quality, minimumDistance, 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(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);
```

```
            normalVortexStrength = fabs(vortexStrength/VortexStrengthMax);
            if(normalVortexStrength>1){normalVortexStrength=1;}

            // putting curvature value in proper form
            curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
            if(curvature>CurvatureMax){curvature=CurvatureMax;}
            normalCurvature = fabs(curvature/CurvatureMax - 1);

            // putting quality value in proper form
            quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
            if(quality>QualityMax){quality=QualityMax;}
            normalQuality = fabs(quality/QualityMax - 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);
            normalMinimumDistance = fabs(minimumDistance/MinimumDistanceMax);
            if(normalMinimumDistance>1){normalMinimumDistance=1;}

            // the function that sets the belief value
            b = 0.8*normalAverage + 0.2;                    //<------------------------
            if(b>1){b=1;}
            // the function that sets the disbelief value
            d = -0.8*normalAverage + 0.8;                   //<------------------------
            if(d<0){d=0;}
            // the function that sets the uncertainty value
            u = normalMinimumDistance*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);
        // cout << "b = " << b << "\td = " << d << "\tu = " << u << "\tSum = " << b+d+u << endl;
        }

////////////////////////////////////////////////////////////////////////////////

        for(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);
            normalVortexStrength = fabs(vortexStrength/VortexStrengthMax);
            if(normalVortexStrength>1){normalVortexStrength=1;}

            // putting curvature value in proper form
            if(curvature>CurvatureMax){curvature=CurvatureMax;}
            normalCurvature = fabs(curvature/CurvatureMax - 1);
            curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);

            // putting quality value in proper form
            quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
```

```
            if(quality>QualityMax){quality=QualityMax;}
            normalQuality = fabs(quality/QualityMax - 1);

            // finding the average of the three values
            normalAverage = (normalVortexStrength+normalCurvature+normalQuality) / 3;

            // the function that sets the b-value
            b = 0.4*normalAverage + 0.6;                              //<-----------------------------
            if(b>1){b=1;}
            // the function that sets the d-value
            d = -0.4*normalAverage + 0.4;                             //<-----------------------------
            if(d<0){d=0;}
            // the function that sets the u-value
            u = 0.05;   //////////**********we are setting this to a low value
                        //////////**********maybe replace this with other vortex factors

            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);
        // cout << "b = " << b << "\td = " << d << "\tu = " << u << "\tSum = " << b+d+u << endl;
        }
    }

//////////**********************************************************//////////
//////////**********************************************************//////////

    // calculating belief tuple values as if RothPeikert was the
    // extraction algorithm for the set of vortex cores.
    if(RothPeikert){
        for(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);
            normalVortexStrength = fabs(vortexStrength/VortexStrengthMax);
            if(normalVortexStrength>1){normalVortexStrength=1;}

            // putting curvature value in proper form
            curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
            normalCurvature = curvature/CurvatureMax;
            if(normalCurvature>1){normalCurvature=1;}

            // putting quality value in proper form
            quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
            if(quality>QualityMax){quality=QualityMax;}
            normalQuality = fabs(quality/QualityMax - 1);

            // finding the average of the three values
            normalAverage = (normalVortexStrength+normalCurvature+normalQuality) / 3;

            // the function that sets the belief value.
            b = 0.4*normalAverage + 0.6;
            if(b>1){b=1;}
```

```
            // the function that sets the disbelief value.
            d = -0.4*normalAverage + 0.6;
            if(d<0){d=0;}
            // the function that sets the uncertainty value.
            u = 0.05;  //////////**********we are setting this to a low value
                       //////////**********maybe replace this with distance from extracted point

            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);
         // cout << "b = " << b << "\td = " << d << "\tu = " << u << "\tSum = " << b+d+u << endl;
         }

/////////////////////////////////////////////////////////////////////////////////

         for(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);
            normalVortexStrength = fabs(vortexStrength/VortexStrengthMax);
            if(normalVortexStrength >1){normalVortexStrength=1;}

            // putting curvature value in proper form
            curvature = input->GetPointData()->GetArray("Curvature")->GetComponent(i,0);
            normalCurvature = fabs(curvature/CurvatureMax);
            if(normalCurvature >1){normalCurvature=1;}

            // putting quality value in proper form
            quality = input->GetPointData()->GetArray("Quality")->GetComponent(i,0);
            if(quality >QualityMax){quality=QualityMax;}
            normalQuality = fabs(quality/QualityMax - 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);
            normalMinimumDistance = fabs(minimumDistance/MinimumDistanceMax);
            if(normalMinimumDistance >1){normalMinimumDistance=1;}

            // the function that sets the belief value
            b = 0.8*normalAverage + 0.2;                    //<------------------------
            if(b>1){b=1;}
            // the function that sets the disbelief value
            d = -0.8*normalAverage + 0.8;                   //<------------------------
            if(d<0){d=0;}
            // the function that sets the uncertainty value
            u = normalMinimumDistance*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 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);
            // cout << "b = " << b << "\td = " << d << "\tu = " << u << "\tSum = " << b+d+u << endl;
        }
    }

    // Combining all the opinions into the final opinion.
    double MA[3], AARP[3], AASH[3], MAxAASH[3], MAxAARP[3], k, finalOpinion[3], gamma;
    for(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;
        // cout << "b=" << finalOpinion[0] << "\td=" << finalOpinion[1] << "\tu=" << finalOpinion[2] << "\tError=" << 1 -finalOpinion[0] -finalOpinion[1] -finalOpinion[2] << endl;
        finalOpinionArray->SetTuple(i,finalOpinion);
    }

    // calculating the probability expectation value
    for(i=0 ; i<input->GetNumberOfPoints() ; i++){
        probExpArray->SetValue(i, finalOpinionArray->GetComponent(i,0)+0.5*finalOpinionArray->GetComponent(i,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());

    return 1;
}

//----------------------------------------------------------------------------
void vtkCreateOpinion::PrintSelf(ostream& os, vtkIndent indent)
{
    this->Superclass::PrintSelf(os,indent);
    os << indent << "PreviousErrorConstant: " << (this->PreviousErrorConstant) << "\n";
    os << indent << "ChangeInErrorConstant: " << (this->ChangeInErrorConstant) << "\n";
}
```

A.4.2 vtkCurvature.cxx

```
#include "vtkCurvature.h"

#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include "vtkMath.h"

#include "vtkIdList.h"

#include <vector>
#include <math.h>

vtkCxxRevisionMacro(vtkCurvature, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkCurvature);

vtkCurvature::vtkCurvature()
{
    this->SingleCurvatureValue = true;
    this->TwoSegmentCurvature = false;
    this->NumberOfCurvatureValues = 4;
}

//----------------------------------------------------------------------------
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(SingleCurvatureValue){
    // Calculating the radius and the curvature
    double ff,gg,mm,x1,x2,x3,y1,y2,y3;
    double cc,dd,hh,ee,kk,ss,radius,curvature;
    double xyzFirst[3], xyzLast[3], xyzMiddle[3];
    std::vector <int> iPointList;
```

```
    /* Initializing the curvature array to add to polydata */
    vtkDoubleArray *curvatureArray = vtkDoubleArray::New();
    curvatureArray->SetNumberOfComponents(1);
    curvatureArray->SetNumberOfTuples(input->GetNumberOfPoints());
    curvatureArray->SetName("Curvature");

    // initializing values
    double p1[3], p2[3], p3[3];
    double v1[3], v2[3], v3[3], v4[3], v5[3], v6[3];
    double n1[3], n2[3], n3[3];

    int p, i;
    for(p=0 ; p<input->GetNumberOfLines() ; p++){
      /* Putting cell point ids into an array because the pointers kept getting screwed up */
      vtkIdList *cellPtIds;
      cellPtIds = input->GetCell(p)->GetPointIds();
      iPointList.resize(cellPtIds->GetNumberOfIds());
      for(i=0 ; i<cellPtIds->GetNumberOfIds() ; i++){
        iPointList[i] = cellPtIds->GetId(i);
      }
      /* Getting point locations at end and beginning of line */
      input->GetCell(p)->GetPoints()->GetPoint(0,xyzFirst);
      input->GetCell(p)->GetPoints()->GetPoint(input->GetCell(p)->GetNumberOfPoints()-1,xyzLast);

      /* Finding the right a,b,c etc for t=0.5 */
      int tcounter = 0;
      double checkt = 0;
      float findt = 0.5;
      while(findt > checkt){
        tcounter = tcounter + 1;
        checkt = input->GetPointData()->GetArray("t")->GetComponent(iPointList[tcounter],0);
      }
      /*Now tcounter is equal to the number of the line that holds the a,b,c,d,e,f values */
      xyzMiddle[0] = input->GetPointData()->GetArray("a")->GetComponent(iPointList[tcounter],0)
           *0.5 + input->GetPointData()->GetArray("d")->GetComponent(iPointList[tcounter],0);
      xyzMiddle[1] = input->GetPointData()->GetArray("b")->GetComponent(iPointList[tcounter],0)
           *0.5 + input->GetPointData()->GetArray("e")->GetComponent(iPointList[tcounter],0);
      xyzMiddle[2] = input->GetPointData()->GetArray("c")->GetComponent(iPointList[tcounter],0)
           *0.5 + input->GetPointData()->GetArray("f")->GetComponent(iPointList[tcounter],0);

      int q;
      for(q=0 ; q<3 ; q++){
      p1[q] = xyzFirst[q];
        p2[q] = xyzMiddle[q];
        p3[q] = xyzLast[q];
      }

      // finding two vectors between the points
      int i;
      for(i=0 ; i<3 ; i++){
        v1[i] = p2[i]-p1[i];
        v3[i] = p3[i]-p1[i];
      }

      // Crossing the vectors to find a normal vector to a plane
      // containing the three points.
      vtkMath::Cross(v1,v3,v4);

      // making the vectors unit vectors
      for(i=0 ; i<3 ; i++){
        v2[i] = v1[i]/vtkMath::Norm(v1);
        v5[i] = v4[i]/vtkMath::Norm(v4);
      }

      // crossing two vectors to find the last orthogonal component
      vtkMath::Cross(v2,v5,v6);

```

```
        // finding the new point values
        // n1=L_mn*p1
        n1[0] = p1[0]*v2[0] + p1[1]*v2[1] + p1[2]*v2[2];
        n1[1] = p1[0]*v6[0] + p1[1]*v6[1] + p1[2]*v6[2];
        n1[2] = p1[0]*v5[0] + p1[1]*v5[1] + p1[2]*v5[2];
        // n2=L_mn*p2
        n2[0] = p2[0]*v2[0] + p2[1]*v2[1] + p2[2]*v2[2];
        n2[1] = p2[0]*v6[0] + p2[1]*v6[1] + p2[2]*v6[2];
        n2[2] = p2[0]*v5[0] + p2[1]*v5[1] + p2[2]*v5[2];
        // n3=L_mn*p3
        n3[0] = p3[0]*v2[0] + p3[1]*v2[1] + p3[2]*v2[2];
        n3[1] = p3[0]*v6[0] + p3[1]*v6[1] + p3[2]*v6[2];
        n3[2] = p3[0]*v5[0] + p3[1]*v5[1] + p3[2]*v5[2];

        /*Separating the x and y values*/
        x1 = n1[0]; y1 = n1[1];
        x2 = n2[0]; y2 = n2[1];
        x3 = n3[0]; y3 = n3[1];

        /*Computing the equation of a circle containing the three points*/
        ff = x3*x3-x3*x2-x1*x3+x1*x2+y3*y3-y3*y2-y1*y3+y1*y2;
        gg = x3*y1-x3*y2+x1*y2-x1*y3+x2*y3-x2*y1;

        if(gg==0){mm=0;}
        else{mm=(ff/gg);}

        cc = (mm*y2)-x2-x1-(mm*y1);
        dd = (mm*x1)-y1-y2-(x2*mm);
        ee = (x1*x2)+(y1*y2)-(mm*x1*y2)+(mm*x2*y1);

        hh = (cc/2);
        kk = (dd/2);
        ss = (((hh)*(hh))+((kk)*(kk))-ee);

        /*radius is equal to the radius of the computed circle*/
        radius = pow(ss,.5);
        curvature = 1/radius;

        /*Setting curvature array*/
        /*Curvature is the same for every point on the line*/
        for(i=0 ; i<input->GetCell(p)->GetNumberOfPoints() ; i++){
            curvatureArray->SetValue(iPointList[i],curvature);
        }
    }
    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());
    }

    else if(TwoSegmentCurvature){
        double c[3], c1[3], c5[3];

        // initializing values
        double p1[3], p2[3], p3[3];
        double v1[3], v2[3], v3[3], v4[3], v5[3], v6[3];
        double n1[3], n2[3], n3[3];

        double ff,gg,mm,x1,x2,x3,y1,y2,y3;
        double cc,dd,hh,ee,kk,ss,radius,curvature;

        // Initializing the curvature array to add to polydata
        vtkDoubleArray *curvatureArray = vtkDoubleArray::New();
        curvatureArray->SetNumberOfComponents(1);
        curvatureArray->SetNumberOfTuples(input->GetNumberOfPoints());
        curvatureArray->SetName("Curvature");
```

```
    int i,j;
    for(i=0 ; i<input->GetNumberOfLines() ; i++){

        // getting cell ids
        vtkIdList *cellPtIds;
        cellPtIds = input->GetCell(i)->GetPointIds();

        // getting the endpoints
        input->GetCell(i)->GetPoints()->GetPoint(0,c1);
        input->GetCell(i)->GetPoints()->GetPoint(input->GetCell(i)->GetNumberOfPoints()-1,c5);

        double findt[5];
        findt[0] = 0;       findt[1] = 0.25;
        findt[2] = 0.5;     findt[3] = 0.75;
        findt[4] = 1;

        std::vector<int> tHolder;
        tHolder.push_back(0);
        std::vector<double> cHolder;
        std::vector<double> holder;
        holder.push_back(c1[0]);
        holder.push_back(c1[1]);
        holder.push_back(c1[2]);

        int p;
        for(p=1 ; p<4 ; p++){

            // Finding the right a,b,c etc for given t
            int tcounter  = 0;
            double checkt = 0;

            while(findt[p] > checkt){
                tcounter = tcounter + 1;
                checkt   = input->GetPointData()->GetArray("t")->GetComponent(cellPtIds->GetId(
                    tcounter),0);
            }
            tHolder.push_back(tcounter);
            // Now tcounter is equal to the number of the line that holds the a,b,c,d,e,f values
            c[0] = input->GetPointData()->GetArray("a")->GetComponent(cellPtIds->GetId(tcounter)
                ,0)*findt[p] + input->GetPointData()->GetArray("d")->GetComponent(cellPtIds->
                GetId(tcounter),0);
            c[1] = input->GetPointData()->GetArray("b")->GetComponent(cellPtIds->GetId(tcounter)
                ,0)*findt[p] + input->GetPointData()->GetArray("e")->GetComponent(cellPtIds->
                GetId(tcounter),0);
            c[2] = input->GetPointData()->GetArray("c")->GetComponent(cellPtIds->GetId(tcounter)
                ,0)*findt[p] + input->GetPointData()->GetArray("f")->GetComponent(cellPtIds->
                GetId(tcounter),0);
            holder.push_back(c[0]);
            holder.push_back(c[1]);
            holder.push_back(c[2]);
        }

        tHolder.push_back(input->GetCell(i)->GetNumberOfPoints()-1);
        holder.push_back(c5[0]);
        holder.push_back(c5[1]);
        holder.push_back(c5[2]);

        for(p=0 ; p<2 ; p++){
            // finding two vectors between the points
            int f;
            for(f=0 ; f<3 ; f++){
                p1[f] = holder[f+3*p];
                p2[f] = holder[f+3+3*p];
                p3[f] = holder[f+6+3*p];
                v1[f] = p2[f]-p1[f];
                v3[f] = p3[f]-p1[f];
            }
```

```
            // Crossing the vectors to find a normal vector to a plane
            // containing the three points.
            vtkMath::Cross(v1,v3,v4);

            // making the vectors unit vectors
            for(f=0 ; f<3 ; f++){
                v2[f] = v1[f]/vtkMath::Norm(v1);
                v5[f] = v4[f]/vtkMath::Norm(v4);
            }

            // crossing two vectors to find the last orthogonal component
            vtkMath::Cross(v2,v5,v6);

            // finding the new point values
            // n1=L_mn*p1
            n1[0] = p1[0]*v2[0] + p1[1]*v2[1] + p1[2]*v2[2];
            n1[1] = p1[0]*v6[0] + p1[1]*v6[1] + p1[2]*v6[2];
            // n2=L_mn*p2
            n2[0] = p2[0]*v2[0] + p2[1]*v2[1] + p2[2]*v2[2];
            n2[1] = p2[0]*v6[0] + p2[1]*v6[1] + p2[2]*v6[2];
            // n3=L_mn*p3
            n3[0] = p3[0]*v2[0] + p3[1]*v2[1] + p3[2]*v2[2];
            n3[1] = p3[0]*v6[0] + p3[1]*v6[1] + p3[2]*v6[2];

            // Separating the x and y values
            x1 = n1[0]; y1 = n1[1];
            x2 = n2[0]; y2 = n2[1];
            x3 = n3[0]; y3 = n3[1];

            // Computing the equation of a circle containing the three points
            ff = x3*x3-x3*x2-x1*x3+x1*x2+y3*y3-y3*y2-y1*y3+y1*y2;
            gg = x3*y1-x3*y2+x1*y2-x1*y3+x2*y3-x2*y1;

            if(gg==0){mm=0;}
            else{mm=(ff/gg);}

            cc = (mm*y2)-x2-x1-(mm*y1);
            dd = (mm*x1)-y1-y2-(x2*mm);
            ee = (x1*x2)+(y1*y2)-(mm*x1*y2)+(mm*x2*y1);

            hh = (cc/2);
            kk = (dd/2);
            ss = (((hh)*(hh))+((kk)*(kk))-ee);

            // radius is equal to the radius of the computed circle
            radius = pow(ss,.5);
            curvature = 1/radius;
            cHolder.push_back(curvature);
        }

        cHolder.push_back((cHolder[0]+cHolder[1])/2);

        int f = 0;
        while(f<input->GetCell(i)->GetNumberOfPoints()){
            if(f<tHolder[1] || f==tHolder[1]){
                curvatureArray->SetComponent(cellPtIds->GetId(f),0,cHolder[0]);
            }
            else if(f>tHolder[1] & f<tHolder[3] || f==tHolder[3]){
                curvatureArray->SetComponent(cellPtIds->GetId(f),0,cHolder[2]);
            }
            else{
                curvatureArray->SetComponent(cellPtIds->GetId(f),0,cHolder[1]);
            }
            f++;
        }

    }
```

```
        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());
    }

    return 1;
}

//----------------------------------------------------------------------------
void vtkCurvature::PrintSelf(ostream& os, vtkIndent indent)
{
    this->Superclass::PrintSelf(os,indent);
    os << indent << "SingleCurvatureValue: "    << (this->SingleCurvatureValue ? "On\n" : "Off\n");
    os << indent << "TwoSegmentCurvature: "     << (this->TwoSegmentCurvature ? "On\n" : "Off\n");
    os << indent << "NumberOfCurvatureValues: " << (this->NumberOfCurvatureValues);
}
```

A.4.3 vtkMinimumDistance.cxx

```
#include "vtkMinimumDistance.h"

#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include <math.h>

vtkCxxRevisionMacro(vtkMinimumDistance, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkMinimumDistance);

//----------------------------------------------------------------------------
vtkMinimumDistance::vtkMinimumDistance()
{
    this->SetNumberOfInputPorts(1);
    this->SetNumberOfOutputPorts(1);
}

//----------------------------------------------------------------------------
int vtkMinimumDistance::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 vtkMinimumDistance::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 previous error 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()));

    // Obtaining minimum distance at each point of input1
    // Initializing the array
    vtkDoubleArray *minDistanceArray = vtkDoubleArray::New();
    minDistanceArray->SetNumberOfValues(input1->GetNumberOfPoints());
    minDistanceArray->SetNumberOfComponents(1);
    minDistanceArray->SetNumberOfTuples(input1->GetNumberOfPoints());
    minDistanceArray->SetName("MinimumDistance");

    double minDistance, distance, xyz1[3], xyz2[3];

    int i, j;
    for(i=0 ; i<input1->GetNumberOfPoints() ; i++){

        input1->GetPoints()->GetPoint(i,xyz1);
        minDistance = 1000000;

        for(j=0 ; j<input2->GetNumberOfPoints() ; j++){
            input2->GetPoints()->GetPoint(j,xyz2);
            distance = sqrt(pow(xyz1[0]-xyz2[0],2) + pow(xyz1[1]-xyz2[1],2) + pow(xyz1[2]-xyz2[2],2)
                );
            if(distance < minDistance){
                minDistance = distance;
            }

        minDistanceArray->SetValue(i,minDistance);

        }
    }
    input1->GetPointData()->AddArray(minDistanceArray);

    // Copying the input data and structure to the output
    output->CopyStructure(input1);
    output->GetPointData()->PassData(input1->GetPointData());
    output->GetCellData()->PassData(input1->GetCellData());

    return 1;
}

//----------------------------------------------------------------------------
void vtkMinimumDistance::PrintSelf(ostream& os, vtkIndent indent)
{
    this->Superclass::PrintSelf(os,indent);
}
```

A.4.4 vtkFeatureDisplacement.cxx

```
#include "vtkFeatureDisplacement.h"

#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
```

```cpp
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include "vtkSameLine.h"
#include <vector>
#include <math.h>

vtkCxxRevisionMacro(vtkFeatureDisplacement, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkFeatureDisplacement);

//----------------------------------------------------------------------------
vtkFeatureDisplacement::vtkFeatureDisplacement()
{
   this->SetNumberOfInputPorts(1);
   this->SetNumberOfOutputPorts(1);
   this->ComputeChangeInError = true;
   this->ClosestPoint = true;
   this->SameLine = false;
}

//----------------------------------------------------------------------------
int vtkFeatureDisplacement::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 vtkFeatureDisplacement::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()));

   // Obtaining feature displacement at each point
   // Initializing the array and naming variables
   vtkDoubleArray *PEArray = vtkDoubleArray::New();
   PEArray->SetNumberOfValues(input1->GetNumberOfPoints());
   PEArray->SetNumberOfComponents(1);
   PEArray->SetNumberOfTuples(input1->GetNumberOfPoints());
   PEArray->SetName("FeatureDisplacement");

   // Obtaining change in error at each point
   // Initializing the array and naming variables
   vtkDoubleArray *CEArray = vtkDoubleArray::New();
   CEArray->SetNumberOfValues(input1->GetNumberOfPoints());
   CEArray->SetNumberOfComponents(1);
   CEArray->SetNumberOfTuples(input1->GetNumberOfPoints());
   CEArray->SetName("ChangeInError");
```

```
    if(ClosestPoint){
        // array to hold minimum distance value
        vtkDoubleArray *mdArray = vtkDoubleArray::New();
        mdArray->SetNumberOfValues(input1->GetNumberOfPoints());
        mdArray->SetNumberOfComponents(1);
        mdArray->SetNumberOfTuples(input1->GetNumberOfPoints());

        // array to hold number of closest point
        vtkDoubleArray *cpArray = vtkDoubleArray::New();
        cpArray->SetNumberOfValues(input1->GetNumberOfPoints());
        cpArray->SetNumberOfComponents(1);
        cpArray->SetNumberOfTuples(input1->GetNumberOfPoints());

        // initialzing values
        double p0[3], c0[3];
        double distance, length;
        double minDistance = 1000;
        int i, j;

        for(i=0 ; i<input1->GetNumberOfPoints() ; i++){
            input1->GetPoints()->GetPoint(i, p0);

            // resetting minDistance value
            minDistance = 1000;

            for(j=0 ; j<input2->GetNumberOfPoints() ; j++){
                input2->GetPoints()->GetPoint(j, c0);

                // measure distance between the points
                distance = sqrt(pow(p0[0]-c0[0],2)+pow(p0[1]-c0[1],2)+pow(p0[2]-c0[2],2));

                if(distance<minDistance){
                    minDistance = distance;
                    cpArray->SetComponent(i, 0, j);
                }
            }
            mdArray->SetValue(i,minDistance);
        }

        for(i=0 ; i<input1->GetNumberOfLines() ; i++){

            // getting idList for cell points
            vtkIdList *cellPtIds;
            cellPtIds = input1->GetCell(i)->GetPointIds();

            input1->GetCell(i)->GetLength2();

            // Getting the length of the current line to use later
            // length = input1->GetPointData()->GetArray("l")->GetComponent(cellPtIds->GetId(0),0);
            length = sqrt(input1->GetCell(i)->GetLength2());

            for(j=0 ; j<input1->GetCell(i)->GetNumberOfPoints() ; j++){
                PEArray->SetValue(cellPtIds->GetId(j),mdArray->GetValue(cellPtIds->GetId(j))*100/
                    length);
            }
        }

        if(ComputeChangeInError){
            double PE1, PE2, CE;
            int i;
            for(i=0 ; i<input1->GetNumberOfPoints() ; i++){
                PE1 = PEArray->GetValue(i);
                PE2 = input2->GetPointData()->GetArray("FeatureDisplacement")->GetComponent(cpArray->
                    GetValue(i),0);
                CE  = fabs(PE1-PE2);
                CEArray->SetValue(i, CE);
            }
```

```
         }  // end of if(ComputeChangeInError)

      mdArray->Delete();
      cpArray->Delete();
      }  // end of ClosestPoint if statement

      if(SameLine){

      std::vector<int> iPointList, iPointList1;
      double length, xyzi[3], xyzi1[3];
      int i, q;

      // Begin iterating through the lines
      int numLines = input1->GetNumberOfLines();
      int p;
      for(p=0 ; p<numLines ; p++){

         // Putting cell point ids into an array because the pointers kept getting screwed up
         // these ids are for the line which is compared to its previous line
         vtkIdList *cellPtIds;
         cellPtIds = input1->GetCell(p)->GetPointIds();
         iPointList.resize(cellPtIds->GetNumberOfIds());
         for(i=0 ; i<cellPtIds->GetNumberOfIds() ; i++){
            iPointList[i] = cellPtIds->GetId(i);
         }

         // these are the ids for the line to be compared to
         cellPtIds = input2->GetCell(SameLineArray->GetValue(p))->GetPointIds();
         iPointList1.resize(cellPtIds->GetNumberOfIds());
         for(i=0 ; i<cellPtIds->GetNumberOfIds() ; i++){
            iPointList1[i] = cellPtIds->GetId(i);
         }

         // Getting the length of the current line to use later
         length = input1->GetPointData()->GetArray("1")->GetComponent(iPointList[0],0);

         double PE, CE, findt;
         double checkt = 0;
         int tcounter = 0;
         // Begin iterating through the points in each line
         for(q=0 ; q<input1->GetCell(p)->GetNumberOfPoints() ; q++){

            // Obtaining feature displacement at first point in line
            if(q==0){
               input1->GetCell(p)->GetPoints()->GetPoint(q,xyzi);
               input2->GetCell(SameLineArray->GetValue(p))->GetPoints()->GetPoint(q,xyzi1);
               PE = (pow(pow(xyzi[0]-xyzi1[0],2) + pow(xyzi[1]-xyzi1[1],2) + pow(xyzi[2]-xyzi1[2],2)
                    ,0.5)/length)*100;
               PEArray->SetValue(iPointList[q],PE);

               // computing the change in feature displacement if required
               if(ComputeChangeInError){
                  CE = fabs(PE - input2->GetPointData()->GetArray("FeatureDisplacement")->
                       GetComponent(iPointList1[0],0));
                  CEArray->SetValue(iPointList[q],CE);
               }
            }

            // Obtaining feature displacement at last point in line
            else if(q==input1->GetCell(p)->GetNumberOfPoints()-1){
               input1->GetCell(p)->GetPoints()->GetPoint(q,xyzi);
               input2->GetCell(SameLineArray->GetValue(p))->GetPoints()->GetPoint(input2->GetCell(p)
                    ->GetNumberOfPoints()-1,xyzi1);
               PE = (pow(pow(xyzi[0]-xyzi1[0],2) + pow(xyzi[1]-xyzi1[1],2) + pow(xyzi[2]-xyzi1[2],2)
                    ,0.5)/length)*100;
               PEArray->SetValue(iPointList[q],PE);

               // computing the change in feature displacement if required
```

```
                if(ComputeChangeInError){
                    CE = fabs(PE - input2->GetPointData()->GetArray("FeatureDisplacement")->GetComponent(iPointList1[input2->GetCell(SameLineArray->GetValue(p))->GetNumberOfPoints() -1],0));
                    CEArray->SetValue(iPointList[q],CE);
                }
            }

            // Obtaining feature displacement at inbetween points
            else{
                tcounter = 0;
                checkt = 0;
                findt = input1->GetPointData()->GetArray("t")->GetComponent(iPointList[q],0);
                while(findt > checkt){
                    tcounter = tcounter + 1;
                    checkt = input2->GetPointData()->GetArray("t")->GetComponent(iPointList1[tcounter],0);
                }

                // Now tcounter is equal to the number of the line that holds the a,b,c,d,e,f values
                xyzi1[0] = input2->GetPointData()->GetArray("a")->GetComponent(iPointList1[tcounter],0)*findt + input2->GetPointData()->GetArray("d")->GetComponent(iPointList1[tcounter],0);
                xyzi1[1] = input2->GetPointData()->GetArray("b")->GetComponent(iPointList1[tcounter],0)*findt + input2->GetPointData()->GetArray("e")->GetComponent(iPointList1[tcounter],0);
                xyzi1[2] = input2->GetPointData()->GetArray("c")->GetComponent(iPointList1[tcounter],0)*findt + input2->GetPointData()->GetArray("f")->GetComponent(iPointList1[tcounter],0);
                input1->GetCell(p)->GetPoints()->GetPoint(q,xyzi);
                PE = (pow(pow(xyzi[0]-xyzi1[0],2) + pow(xyzi[1]-xyzi1[1],2) + pow(xyzi[2]-xyzi1[2],2),0.5)/length)*100;
                PEArray->SetValue(iPointList[q],PE);

                // computing the change in feature displacement if required
                if(ComputeChangeInError){
                    CE = fabs(PE - input2->GetPointData()->GetArray("FeatureDisplacement")->GetComponent(iPointList1[tcounter],0));
                    CEArray->SetValue(iPointList[q],CE);
                }
            }
        }
    }
    }

    // adding computed arrays to input1
    input1->GetPointData()->AddArray(PEArray);
    if(ComputeChangeInError){
        input1->GetPointData()->AddArray(CEArray);
    }

    // Copying the input data and structure to the output
    output->CopyStructure(input1);
    output->GetPointData()->PassData(input1->GetPointData());
    output->GetCellData()->PassData(input1->GetCellData());

    return 1;
}

//----------------------------------------------------------------------------
void vtkFeatureDisplacement::PrintSelf(ostream& os, vtkIndent indent)
{
    this->Superclass::PrintSelf(os,indent);
}
```

A.4.5 vtkQuality.cxx

```
#include "vtkQuality.h"

#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include "vtkMath.h"
#include "vtkThreshold.h"
#include "vtkUnstructuredGrid.h"
#include "vtkGeometryFilter.h"
#include <math.h>

vtkCxxRevisionMacro(vtkQuality, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkQuality);

//----------------------------------------------------------------------------
vtkQuality::vtkQuality()
{
  this->ThresholdLines = true;
  this->QualityThresholdValue = 27;
}

//----------------------------------------------------------------------------
int vtkQuality::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 quality array
  vtkDoubleArray *qualityArray = vtkDoubleArray::New();
  qualityArray->SetNumberOfValues(input->GetNumberOfPoints());
  qualityArray->SetNumberOfComponents(1);
  qualityArray->SetNumberOfTuples(input->GetNumberOfPoints());
  qualityArray->SetName("Quality");

  // computing the quality
  double theta, theta2;
  double v1[3], v2[3], v3[3], vel[3], nvel[3];
  int i, j;
  for(i=0 ; i<input->GetNumberOfLines() ; i++){
    for(j=0 ; j<input->GetCell(i)->GetNumberOfPoints() ; j++){

      // getting point Ids to use later
      vtkIdList *ptIds = vtkIdList::New();
      input->GetCellPoints(i,ptIds);

      if(j==0){
        // set velocities and position vectors
        vel[0] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),0)
            ;
```

```
        vel[1] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),1);
        vel[2] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),2);
        input->GetCell(i)->GetPoints()->GetPoint(j,v1);
        input->GetCell(i)->GetPoints()->GetPoint(j+1,v2);

        // making the vectors unit vectors
        int q;
        for(q=0 ; q<3 ; q++){
            v1[q] = v1[q]-v2[q];
        }
        for(q=0 ; q<3 ; q++){
            v3[q]   = v1[q]  /  vtkMath::Norm(v1);
            nvel[q] = vel[q] /  vtkMath::Norm(vel);
        }
        theta = acos(vtkMath::Dot(v3,nvel));
        theta = theta*180/3.14159265;  // radians to degrees
        if(theta>90){theta=180-theta;}
        qualityArray->SetComponent(ptIds->GetId(j),0,theta);
    }

    else if(j==input->GetCell(i)->GetNumberOfPoints()-1){
        vel[0] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),0);
        vel[1] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),1);
        vel[2] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),2);
        input->GetCell(i)->GetPoints()->GetPoint(j,v1);
        input->GetCell(i)->GetPoints()->GetPoint(j-1,v2);
        // making the vectors unit vectors
        int q;
        for(q=0 ; q<3 ; q++){
            v1[q] = v1[q]-v2[q];
        }
        for(q=0 ; q<3 ; q++){
            v3[q] = v1[q]/vtkMath::Norm(v1);
            nvel[q] = vel[q]/vtkMath::Norm(vel);
        }
        theta = acos(vtkMath::Dot(v3,nvel));
        theta = theta*180/3.14159265;  // radians to degrees
        if(theta>90){theta=180-theta;}
        qualityArray->SetComponent(ptIds->GetId(j),0,theta);
    }

    else{
        vel[0] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),0);
        vel[1] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),1);
        vel[2] = input->GetPointData()->GetArray("Velocity")->GetComponent(ptIds->GetId(j),2);
        input->GetCell(i)->GetPoints()->GetPoint(j,v1);
        input->GetCell(i)->GetPoints()->GetPoint(j-1,v2);
        // making the vectors unit vectors
        int q;
        for(q=0 ; q<3 ; q++){
            v1[q] = v1[q]-v2[q];
        }
        for(q=0 ; q<3 ; q++){
            v3[q] = v1[q]/vtkMath::Norm(v1);
            nvel[q] = vel[q]/vtkMath::Norm(vel);
        }
        theta = acos(vtkMath::Dot(v3,nvel));
        theta = theta*180/3.14159265;  // radians to degrees
        if(theta>90){theta=180-theta;}
        qualityArray->SetComponent(ptIds->GetId(j),0,theta);
```

```
        // for the interior points we can calculate two quality values
        input->GetCell(i)->GetPoints()->GetPoint(j,v1);
        input->GetCell(i)->GetPoints()->GetPoint(j+1,v2);
        // making the vectors unit vectors
        for(q=0 ; q<3 ; q++){
          v1[q] = v1[q]-v2[q];
        }
        for(q=0 ; q<3 ; q++){
          v3[q] = v1[q]/vtkMath::Norm(v1);
        }
        theta2 = acos(vtkMath::Dot(v3,nvel));
        theta2 = theta2*180/3.14159265;  // radians to degrees
        if(theta2 >90){theta2=180-theta2;}
        if(theta2<theta){
          qualityArray->SetComponent(ptIds->GetId(j),0,theta2);
        }
      }
    }
  }
  // Setting quality array to input
  input->GetPointData()->AddArray(qualityArray);

  // threshold by an average quality value
  double avgQuality = 0;
  if(ThresholdLines){
    // creating Avergae quality array
    vtkDoubleArray *averageQualityArray = vtkDoubleArray::New();
    averageQualityArray->SetNumberOfValues(input->GetNumberOfPoints());
    averageQualityArray->SetNumberOfComponents(1);
    averageQualityArray->SetName("AverageQuality");

    for(i=0 ; i<input->GetNumberOfLines() ; i++){
      // getting point Ids to use later
      vtkIdList *ptIds = vtkIdList::New();
      input->GetCellPoints(i,ptIds);

      // finding the average quality across the line
      for(j=0 ; j<input->GetCell(i)->GetNumberOfPoints() ; j++){
        avgQuality = input->GetPointData()->GetArray("Quality")->GetComponent(ptIds->GetId(j),0) + avgQuality;
      }
      avgQuality = avgQuality / input->GetCell(i)->GetNumberOfPoints();

      for(j=0 ; j<input->GetCell(i)->GetNumberOfPoints() ; j++){
        averageQualityArray->SetComponent(ptIds->GetId(j),0,avgQuality);
      }
    }
  input->GetPointData()->AddArray(averageQualityArray);

  // thresholding based on average vortex quality
  vtkThreshold *threshold = vtkThreshold::New();
  threshold->SetInput(input);
  threshold->ThresholdByLower(QualityThresholdValue);
  threshold->SetInputArrayToProcess(0,0,0,0,"AverageQuality");
  threshold->Update();

  // converting unstructured grid to poly data
  vtkGeometryFilter *geometryFilter = vtkGeometryFilter::New();
  geometryFilter->SetInput(threshold->GetOutput());
  geometryFilter->Update();
  geometryFilter->GetOutput()->GetPointData()->RemoveArray("AverageQuality");

  /*Copying the input data and structure to the output*/
  output->CopyStructure(geometryFilter->GetOutput());
  output->GetPointData()->PassData(geometryFilter->GetOutput()->GetPointData());
  output->GetCellData()->PassData(geometryFilter->GetOutput()->GetCellData());
  }
```

```
  else{
    /*Copying the input data and structure to the output*/
    output->CopyStructure(input);
    output->GetPointData()->PassData(input->GetPointData());
    output->GetCellData()->PassData(input->GetCellData());
  }

  return 1;
}

//----------------------------------------------------------------------------
void vtkQuality::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
  os << indent << "ThresholdLines: " << (this->ThresholdLines ? "On\n" : "Off\n");
  os << indent << "QualityThresholdValue: " << (this->QualityThresholdValue) << "\n";
}
```

A.4.6 vtkSameLine.cxx

```
#include "vtkSameLine.h"

#include "vtkCellArray.h"
#include "vtkCellData.h"
#include "vtkDoubleArray.h"
#include "vtkInformation.h"
#include "vtkInformationVector.h"
#include "vtkObjectFactory.h"
#include "vtkPointData.h"
#include "vtkPolyData.h"
#include <math.h>
#include <iostream>

vtkCxxRevisionMacro(vtkSameLine, "$Revision: 1.70 $");
vtkStandardNewMacro(vtkSameLine);

//----------------------------------------------------------------------------
vtkSameLine::vtkSameLine()
{
  this->SetNumberOfInputPorts(1);
  this->SetNumberOfOutputPorts(1);
}

//----------------------------------------------------------------------------
int vtkSameLine::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 vtkSameLine::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 locating same lines 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()));

  // creating sameLine int array that holds values for lines in input2 that
  // have a minimum distance from lines in input1
  SameLine = vtkIntArray::New();
  SameLine->SetNumberOfComponents(1);
  SameLine->SetNumberOfTuples(input1->GetNumberOfLines());
  SameLine->SetName("SameLine");

  // initialzing values
  double p0[3], p1[3], c0[3], c1[3];
  double distance, distance2;
  double minDistance = 1000;

  // begin iterating through lines in input1
  int i, j;
  for(j=0 ; j<input1->GetNumberOfLines() ; j++){
    // getting endpoints from each line in input1
    input1->GetCell(j)->GetPoints()->GetPoint(0 , p0 );
    input1->GetCell(j)->GetPoints()->GetPoint(input1->GetCell(j)->GetPoints()->
        GetNumberOfPoints()-1 , p1 );

    // resetting minDistance value
    minDistance = 1000;

    for(i=0 ; i<input2->GetNumberOfLines() ; i++){
      // getting endpoints from each line in input2
      input2->GetCell(i)->GetPoints()->GetPoint(0 , c0 );
      input2->GetCell(i)->GetPoints()->GetPoint(input2->GetCell(i)->GetPoints()->
          GetNumberOfPoints()-1 , c1 );

      // Measure distance between the endpoints
      distance  = sqrt(pow(p0[0]-c0[0],2)+pow(p0[1]-c0[1],2)+pow(p0[2]-c0[2],2)) +
                  sqrt(pow(p1[0]-c1[0],2)+pow(p1[1]-c1[1],2)+pow(p1[2]-c1[2],2));
      distance2 = sqrt(pow(p0[0]-c1[0],2)+pow(p0[1]-c1[1],2)+pow(p0[2]-c1[2],2)) +
                  sqrt(pow(p1[0]-c0[0],2)+pow(p1[1]-c0[1],2)+pow(p1[2]-c0[2],2));
      if(distance<minDistance){
        minDistance = distance;
        SameLine->SetComponent(j,0,i);
      }
      if(distance2<minDistance){
        minDistance = distance2;
        SameLine->SetComponent(j,0,i);
      }
    }
  }

  /*Copying the input data and structure to the output*/
  output->CopyStructure(input1);
  output->GetPointData()->PassData(input1->GetPointData());
  output->GetCellData()->PassData(input1->GetCellData());

  return 1;
}
//----------------------------------------------------------------------------
void vtkSameLine::PrintSelf(ostream& os, vtkIndent indent)
{
  this->Superclass::PrintSelf(os,indent);
}
```

APPENDIX B. FLOW VISUALIZATION IMAGES

This appendix contains figures of the delta wing data set at varying degrees of solution convergence. There are eight values displayed for each converging data set: feature displacement, change in feature displacement, vortex strength, quality, belief, disbelief, uncertainty and probability expectation. The first four values help to set the probability expectation and belief tuple values. The scales for the color bars were chosen to give the best understanding of each value.

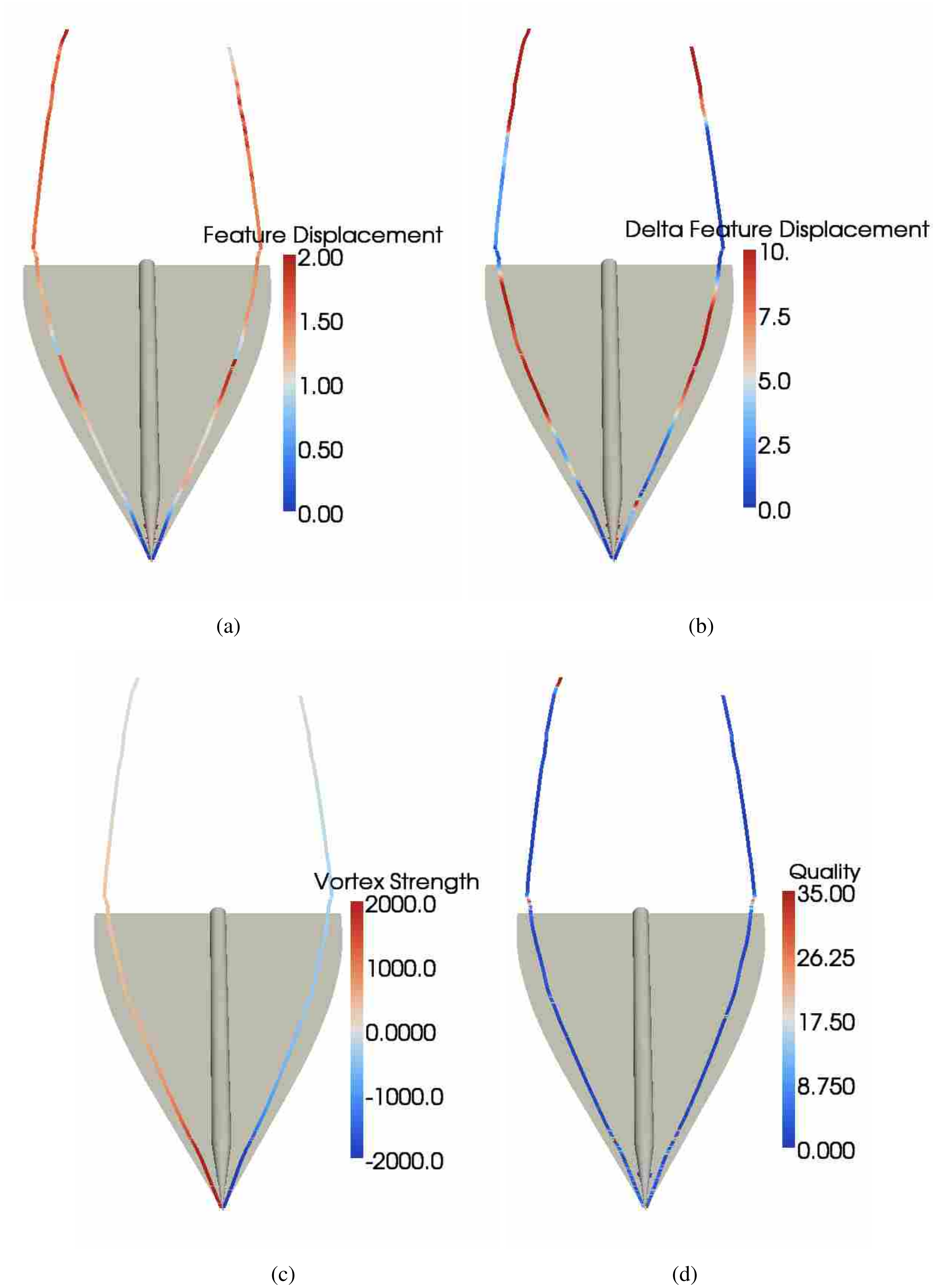


Figure B.1: Values for primary cores extracted by SH from 26% converged simulation.

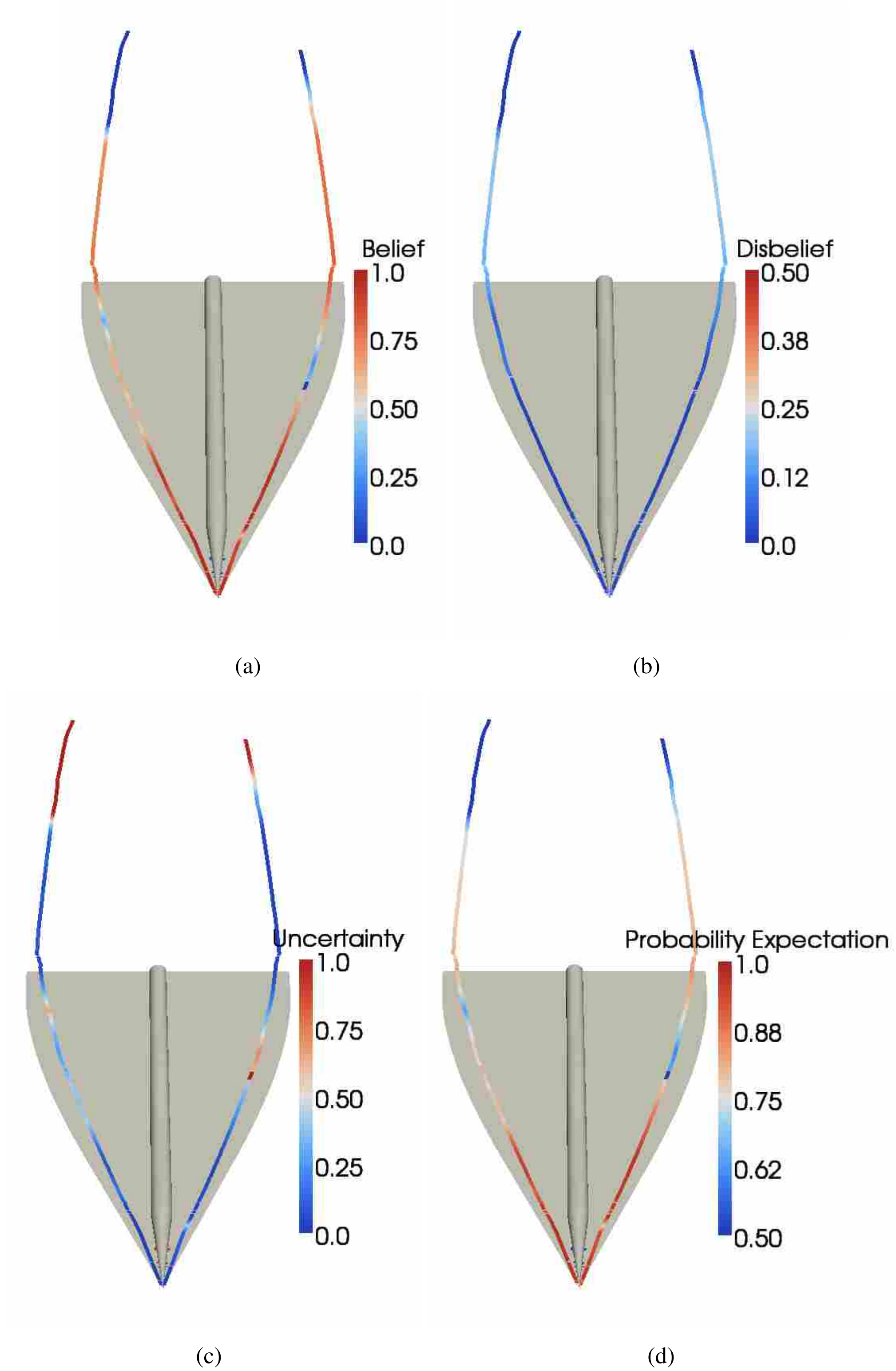


Figure B.2: Probability expectation and belief tuple values for primary cores extracted by SH from 26% converged simulation.

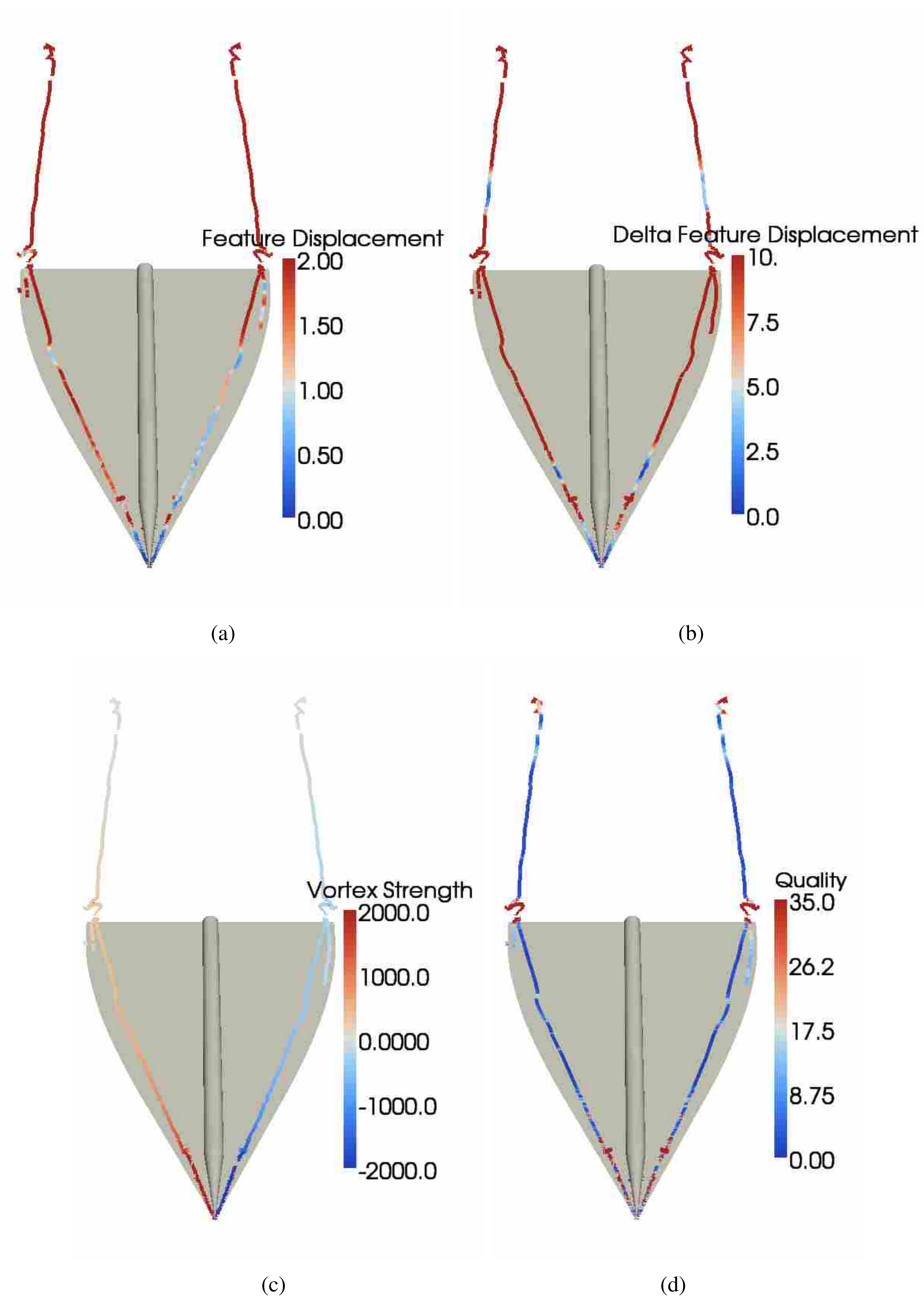
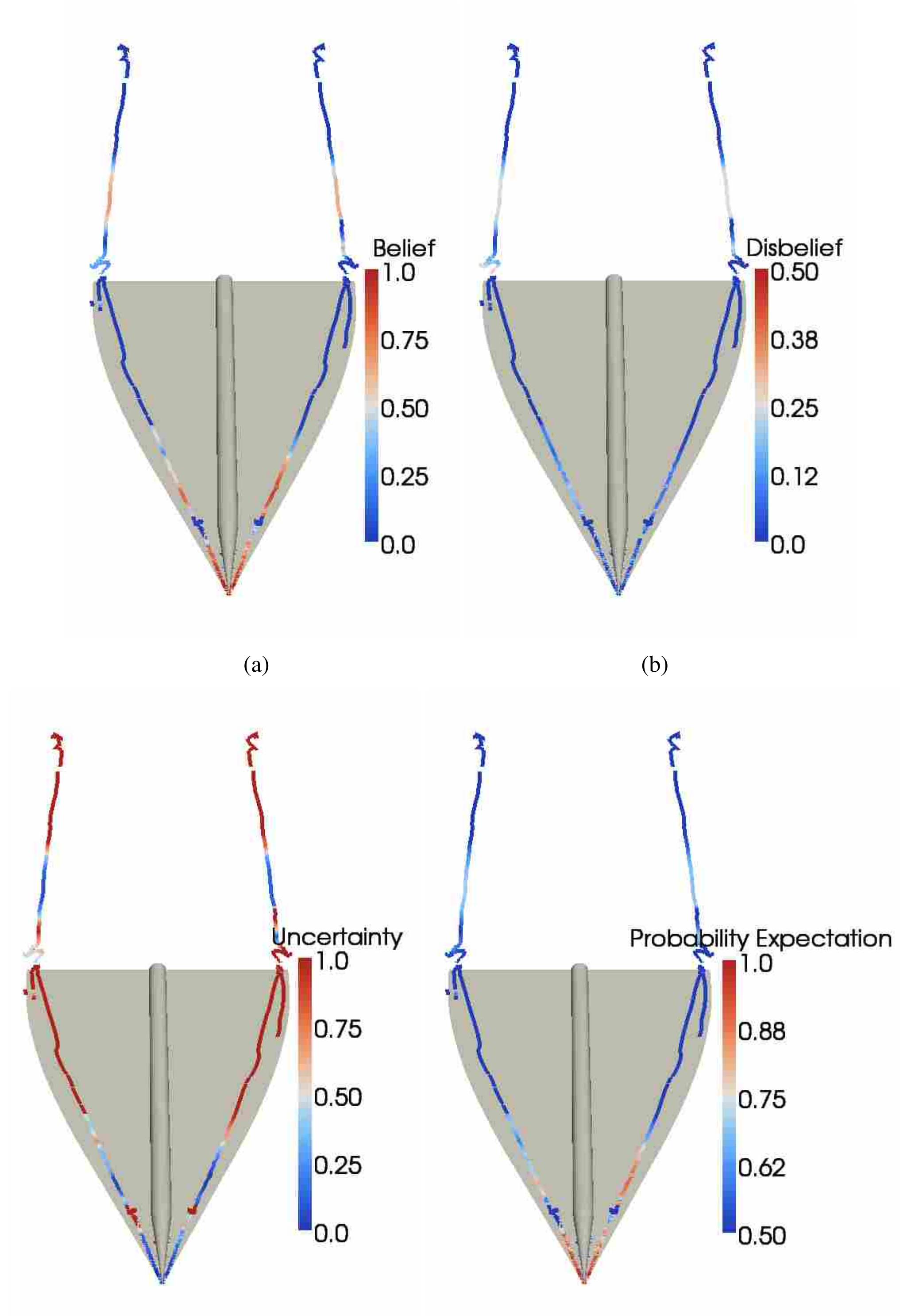
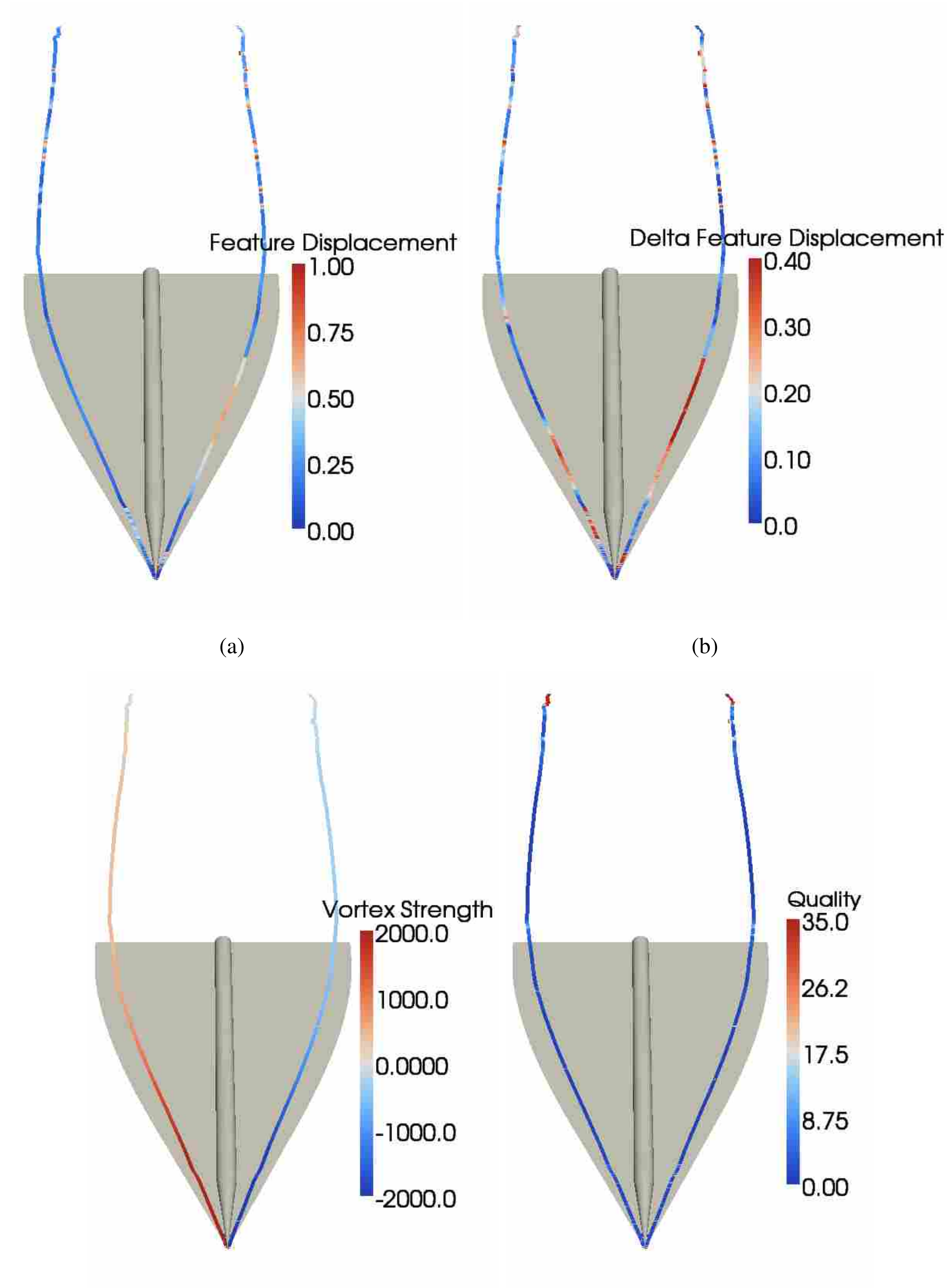


Figure B.3: Values for primary cores extracted by RP from 26% converged simulation.



(a) (b) (c) (d)

Figure B.4: Probability expectation and belief tuple values for primary cores extracted by RP from 26% converged simulation.



(a) (b)

(c) (d)

Figure B.5: Values for primary cores extracted by SH from 68% converged simulation.

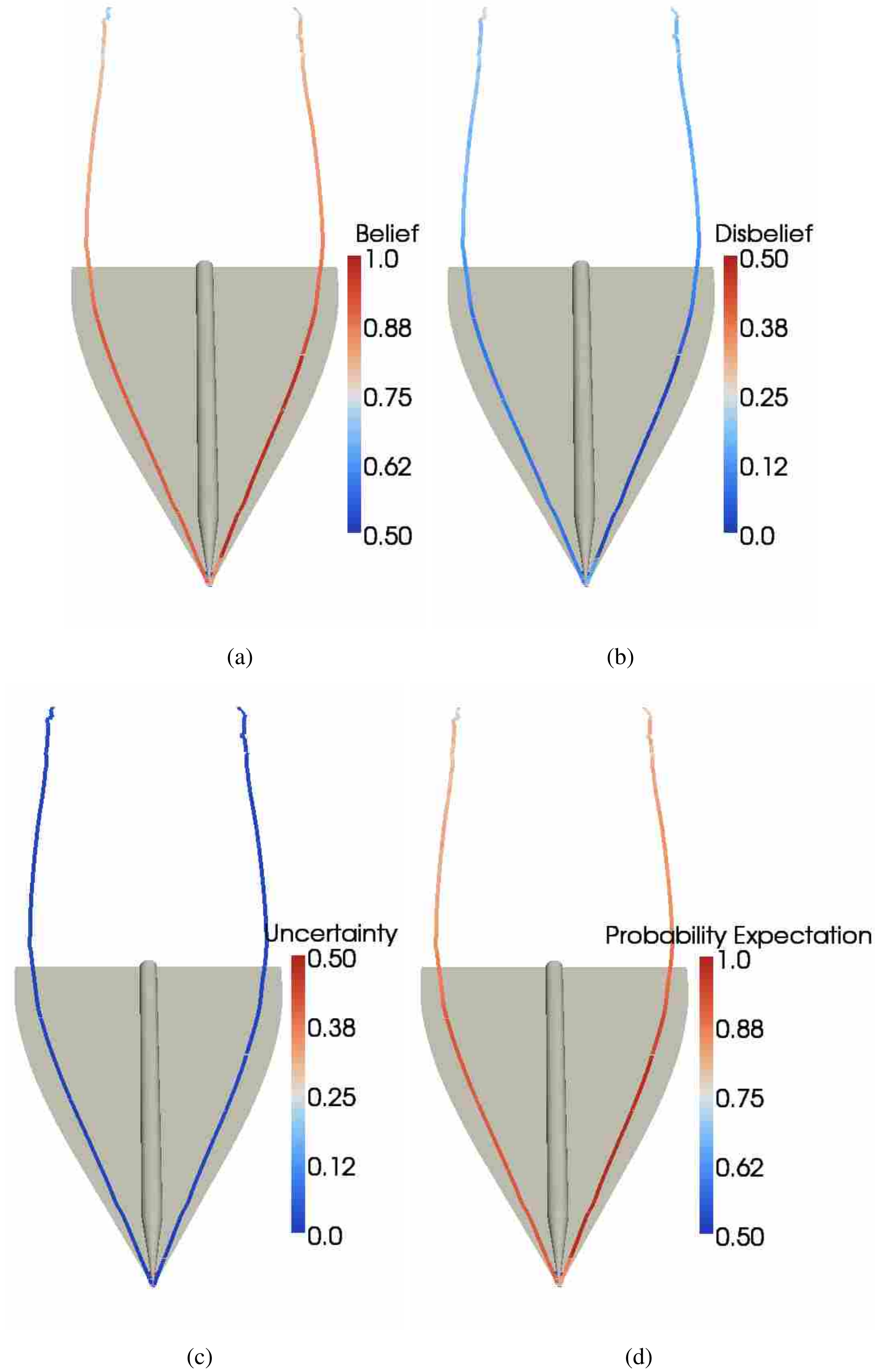
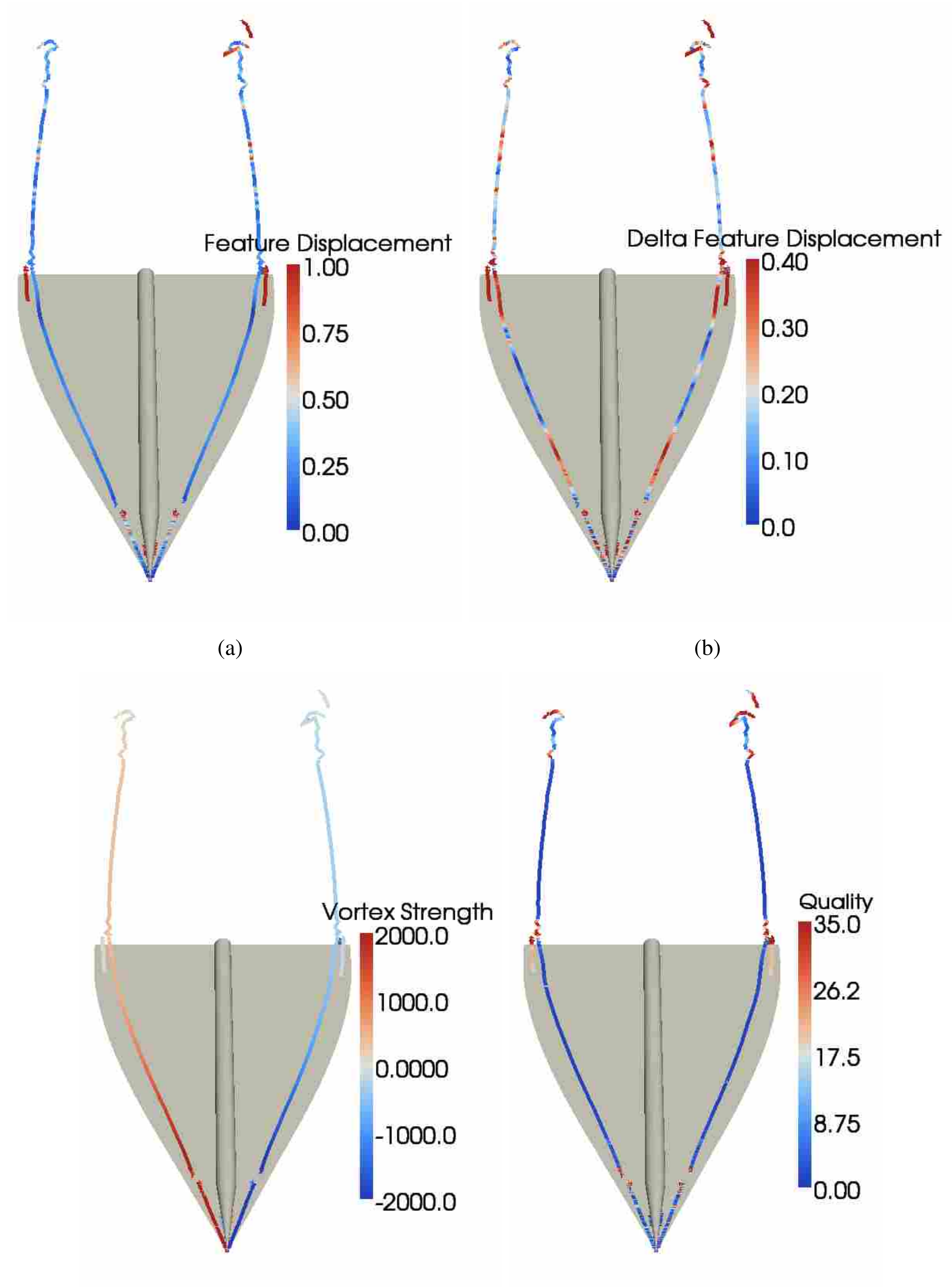


Figure B.6: Probability expectation and belief tuple values for primary cores extracted by SH from 68% converged simulation.



(a) (b)

(c) (d)

Figure B.7: Values for primary cores extracted by RP from 68% converged simulation.

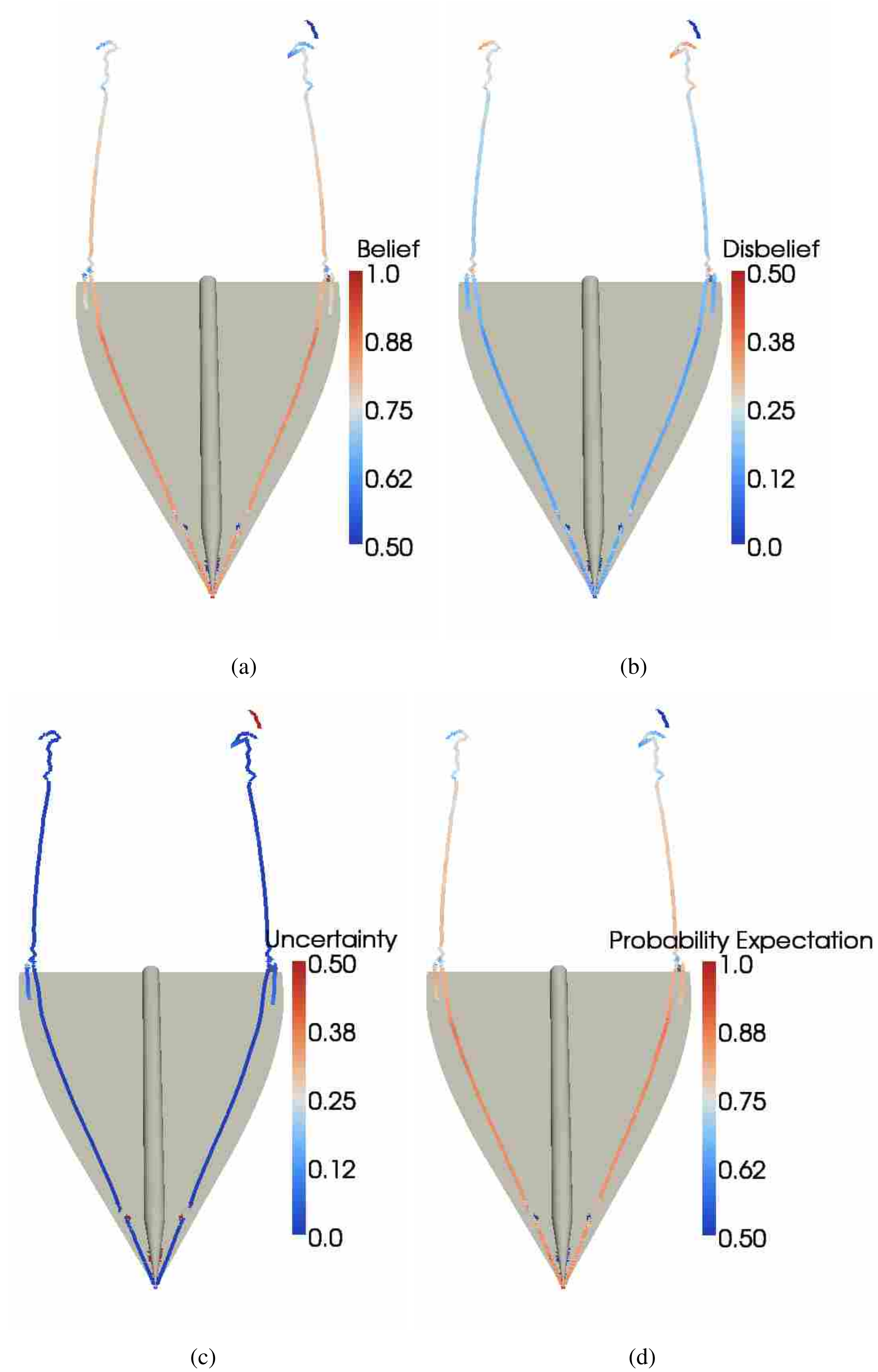


Figure B.8: Probability expectation and belief tuple values for primary cores extracted by RP from 68% converged simulation.

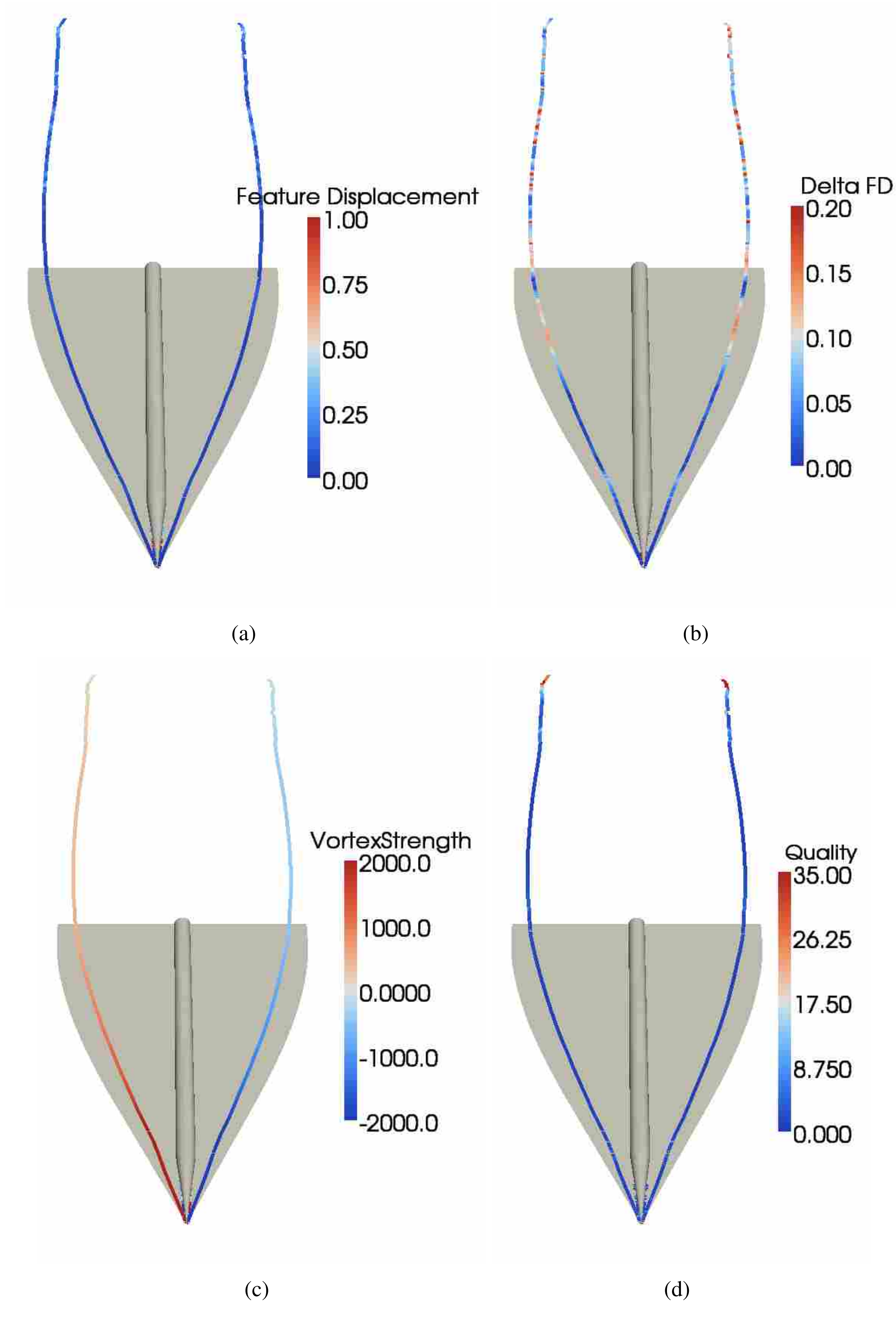


Figure B.9: Values for primary cores extracted by SH from converged simulation.

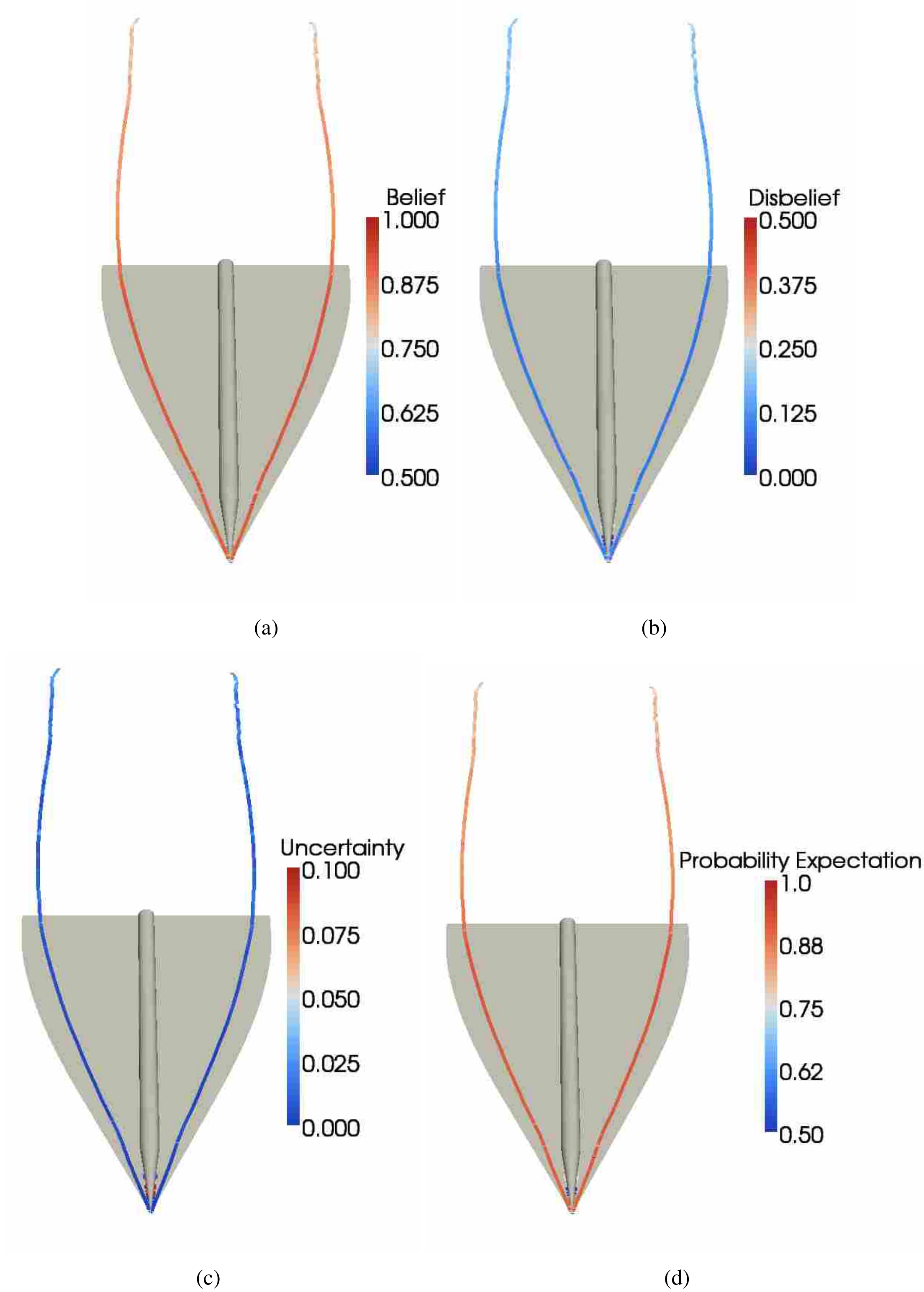


Figure B.10: Probability expectation and belief tuple values for primary cores extracted by SH from converged simulation.

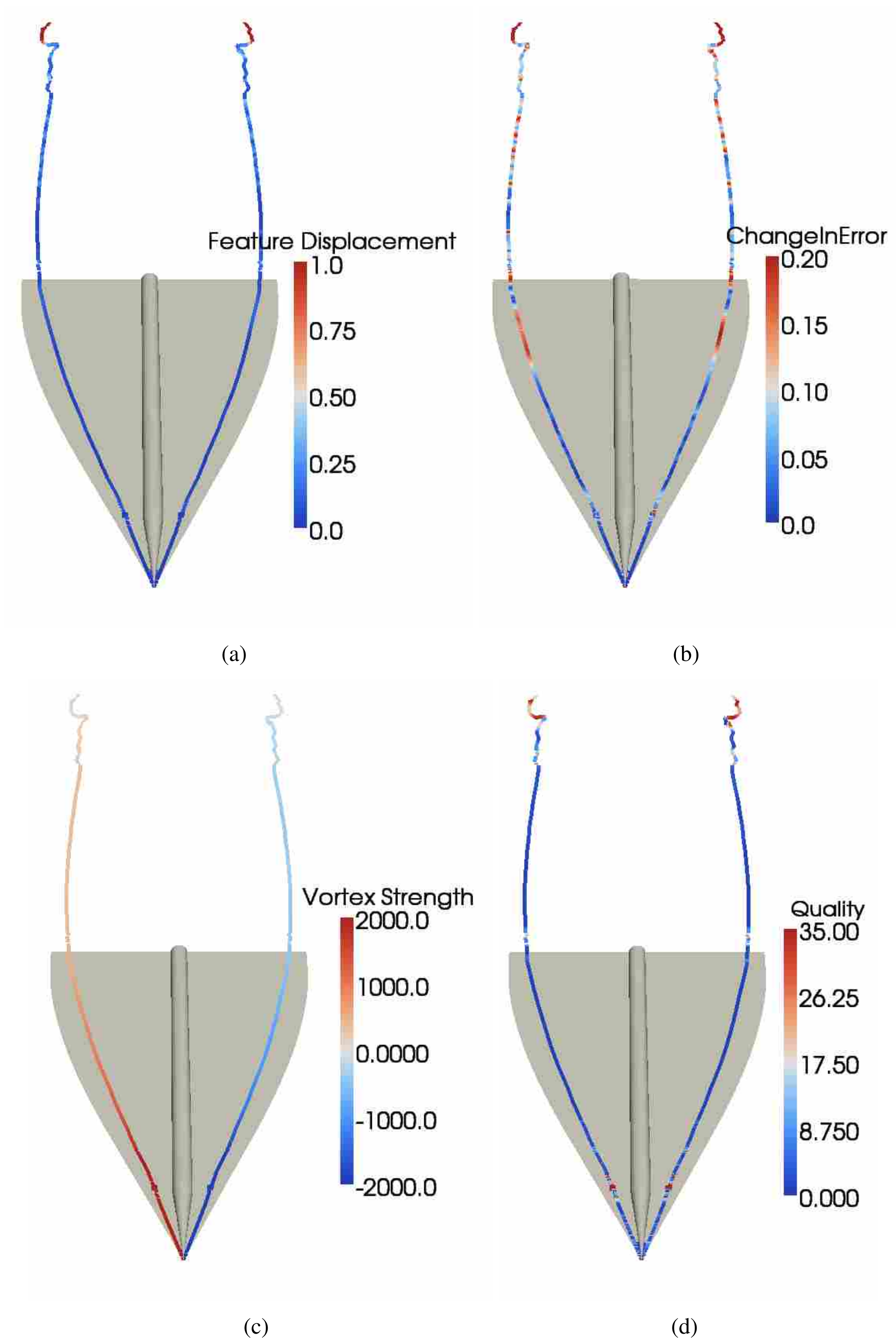
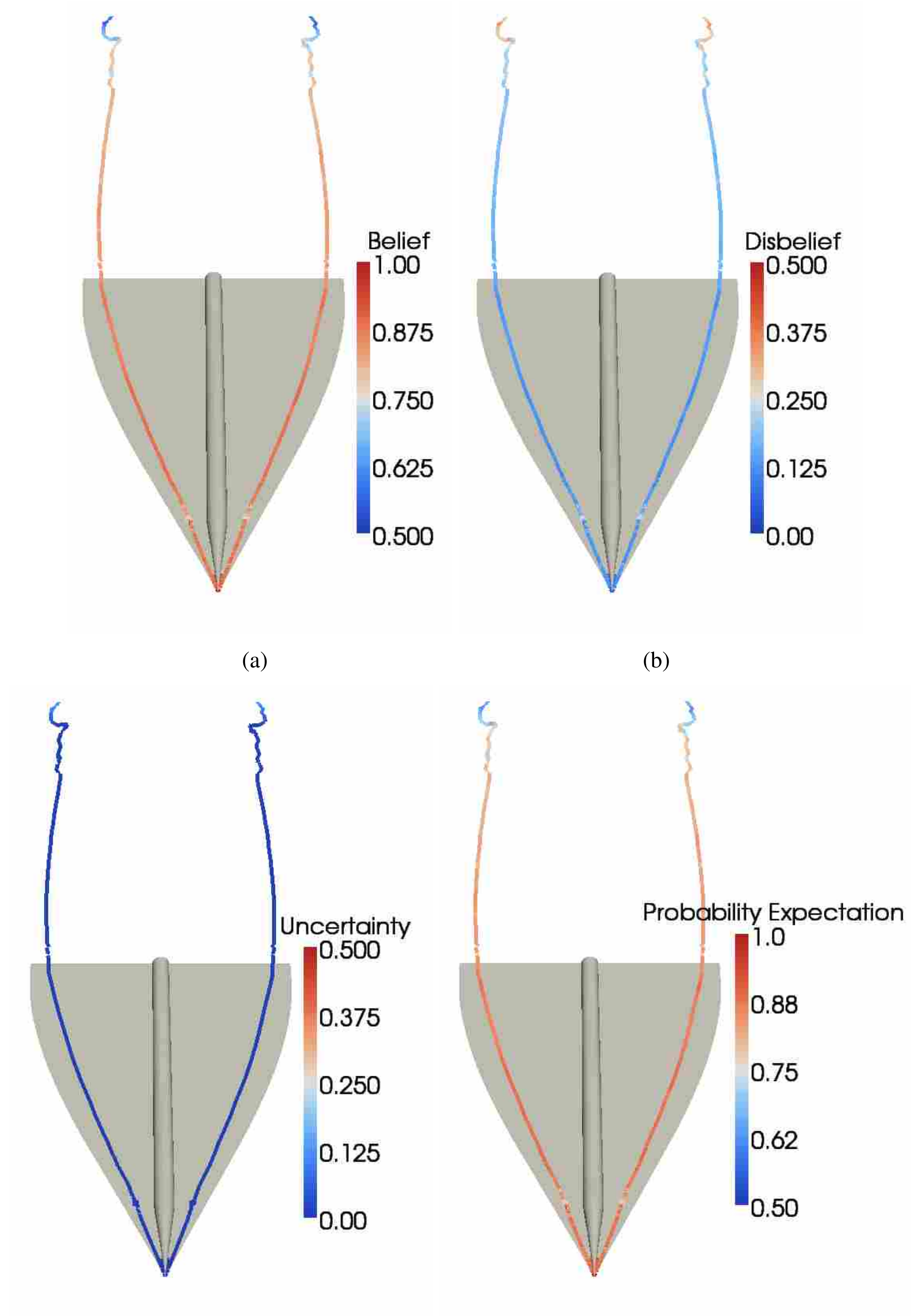


Figure B.11: Values for primary cores extracted by RP from converged simulation.



(a) (b)

(c) (d)

Figure B.12: Probability expectation and belief tuple values for primary cores extracted by RP from converged simulation.