Evolving time surfaces and tracking mixing indicators for flow visualization

Farid Harhad
Louisiana State University and Agricultural and Mechanical College

EVOLVING TIME SURFACES AND TRACKING MIXING INDICATORS FOR FLOW VISUALIZATION

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Farid Harhad
B.S., Louisiana State University, 2004
August 2014

Acknowledgments

I would like to thank my advisor Dr. Bijaya Karki for his support, guidance, and infinite patience throughout my research. I will always be grateful to him for introducing me to the field of scientific visualization. I would also like to thank Dr. Acharya for broadening my horizons by introducing me to the NSF IGERT on Multi-scale Computational Fluid Dynamics program. I am very grateful to Dr. Acharya for exposing me to a gifted group of professors and students that I would have not otherwise met. I am very fortunate to have been part of this unique opportunity.

I would like thank Dr. Somnath Roy for introducing me to his research on stirred tank mixing and for providing me his simulation dataset. I am very grateful to Dr. Werner Benger for providing me his software called VISH which I have used in a portion of my research, for his interest in my work, and for his knowledge of vector field visualization. I would like to thank Dr. Nathan Brener for including me in his research efforts on vector field curvature and torsion. I would also like to thank Pranaya Pokharel for providing me a sample of his oil spill simulation dataset.

Additionally, I would like to thank Dr. Jian Zhang, Dr. Robert Kooima, and Dr. Carol Friedland for serving on my graduate committee and for their valuable comments, feedback, and suggestions.

Finally, I would like to thank my family and friends for all their love and support. I would like to thank my father for all the things he has taught me, including, but not limited to, math and my first BASIC program on a Casio calculator. I would like to thank my mother and my sister for always cheering me up. And last but not least, I would like to thank all my friends for making this whole experience more memorable and worthwhile.

Table of Contents

List of Tables

List of Figures

Abstract

The complexity of large scale computational fluid dynamic simulations (CFD) demands powerful tools to investigate the numerical results. To analyze and understand these voluminous results, we need to visualize the 3D flow field. We chose to use a visualization technique called Time Surfaces. A time surface is a set of surfaces swept by an initial seed surface for a given number of timesteps. We use a front tracking approach where the points of an in initial surface are advanced in a Lagrangian fashion. To maintain a smooth time surface, our method requires surface refinement operations that either split triangle edges, adjust narrow triangles, or delete small triangles. In the conventional approach of edge splitting, we compute the length of an edge, and split that edge if it has exceeded a certain threshold length. In our new approach, we examine the angle between the two vectors at a given edge. We split the edge if the vectors are diverging from one another. This vector angle criterion enables us to refine an edge before advancing the surface front. Refining a surface prior to advancing it has the effect of minimizing the amount of interpolation error. In addition, unlike the edge length criterion which yields a triangular mesh with even vertex distribution throughout the surface, the vector angle criterion yields a triangular mesh that has fewer vertices where the vector field is flat and more vertices where the vector field is curved.

Motivated by the evaluation and the analysis of flow field mixing quantities, this work explores two types of quantitative measurements. First, we look at Ottino's mixing indicators which measure the degree of mixing of a fluid by quantifying the rate at which a sample fluid blob stretches in a flow field over a period of time. Using the geometry of the time surfaces we generated, we are able to easily evaluate otherwise complicated mixing quantities. Second, we compute the curvature and torsion of the velocity field itself. Visualizing the distribution and intensity of the curvature and torsion scalar fields enables us to identify regions of strong and low mixing. To better observe these scalar fields, we designed a multi-scale colormap that emphasizes small, medium, and large values, simultaneously. We test our time surface method and analyze fluid flow mixing quantities on two CFD datasets: a stirred tank simulation and a BP oil spill simulation.

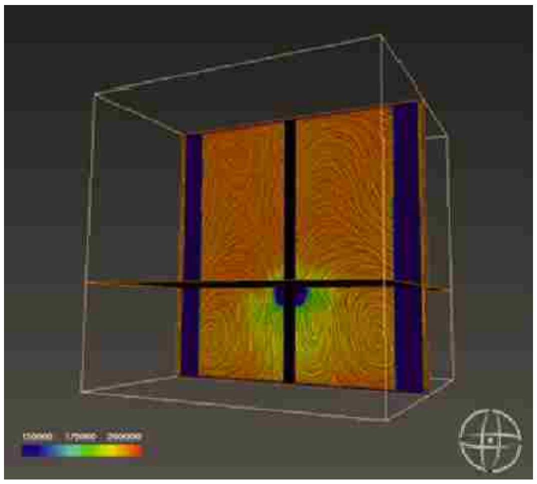
Chapter 1
Introduction

1.1 Motivation

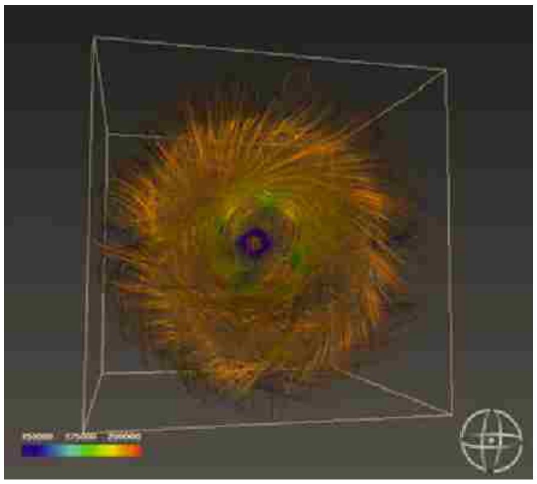
Computational fluid dynamics, or CFD, is a branch of fluid mechanics that studies systems involving fluid flow phenomena by means of computer-based simulation. CFD emerged in the 1960s as a third approach of studying fluid dynamics that complements theoretical and experimental fluid dynamics [1]. CFD spans a wide range of application areas, which include but are not limited to the following [35]: aerodynamics of aircrafts and vehicles, hydrodynamics of ships, combustion engines, gas turbines, turbo-machinery, environmental engineering such as pollutants dispersion, meteorology, biomedical engineering such as blood flow in veins and arteries, chemical engineering such as mixing of polymers, oceanography, and many more.

Computational fluid dynamics is a computationally intensive discipline that requires high computing power to solve complex differential equations. In order to analyze and interpret complex CFD simulation results, we employ scientific visualization methods. Scientific visualization is the field of computer science used to transform scientific datasets into display forms such as images. Such visual representation of data/information enables scientists and engineers to gain insight and understanding of the systems that are simulated. Examples of some visualization methods are shown in Figure 1.1. Scientific data can originate from numerical simulations as in CFD, or they can be acquired experimentally by measuring some physical quantity of interest with a variety of equipment such as a MRI machine or a 3D laser scanner. Scientific data is usually 3D or 4D, and is classified into one of three types: scalar, vector, and tensor. As in CFD, many scientific and engineering simulations-or experiments-yield large spatial and temporal results that can easily reach several terabytes in size. Without scientific visualization, it would be difficult to understand these raw results.

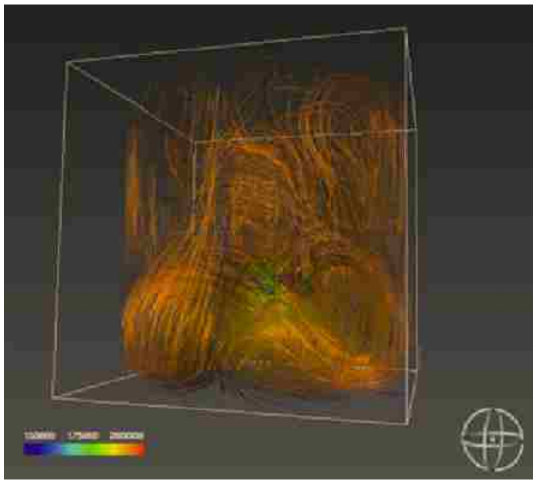
In the CFD datasets that we analyze for this dissertation, we are interested in visualizing the 3D flow field, i.e., the associated velocity field. One of our primary CFD datasets is a simulation



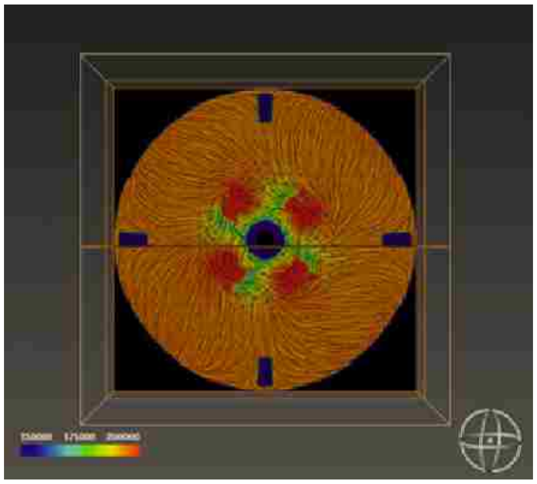
(a) LIC



(b) Streamlines



(c) Streamlines



(d) LIC

Figure 1.1: Visualization of the flow field in a stirred tank using three types of scientific visualization methods: 3D streamlines of the velocity field, 2D streamlines (LIC: Line Integral Convolution), and color-coding the streamlines (2D and 3D) according to the pressure field.

of flow in a stirred tank reactor (STR) using a large-eddy simulation (LES) technique with a body-fitted curvilinear mesh. Stirred tanks are used in the chemical industry in a wide variety of mixing operations. It was reported in [26] that an estimate of half of the $750 billion of US chemical industry is circulated through STRs, and that nearly $1-20 billion per year is potentially lost due to inadequately designed stirred tanks. To design better STRs, it is important to have a detailed understanding of the associated flow behavior. So, 3D visualization of the flow patterns is crucial to identify salient features of the flow. We chose a visualization technique called times surfaces. A time surface is a set of surfaces swept by an initial seed surface throughout a given time period.

The benefit of this visualization technique is that it will allow us to understand and track the mixing and folding of material surfaces in the stirred tank simulation. Furthermore, we want to use the sequence of surfaces we generated to compute some geometric mixing indicators, introduced by J. M. Ottino [23], such as how much and how fast does the material surfaces stretch ($\eta(t)$ and $D\ln(\eta)/Dt$). This approach of using time surfaces to analyze mixing quality is novel in the field of visualization. Examining such mixing indicators can allow us to quantitatively compare the performance of different stirred tank designs.

Current research in time surfaces and dynamic surfaces can be broadly categorized into front-tracking approaches and front-capturing approaches. Front-tracking methods track a set of particles on an initial surface, and each particle is advected in a Lagrangian fashion. Often times, these methods require surface refinement operations that increase, adjust the configuration, or delete the triangles on the surface to maintain smoothness. The challenge in such methods is handling topological changes. On the other hand, front-capturing methods-such as level set-handle topological changes well because the surface is defined implicitly, then it is recomputed at every new timestep in an Eulerian fashion by solving the level set equation again. These methods require constant re-initialization of the implicit surface because in under-resolved regions of the grid, front-capturing methods suffer from mass/volume loss. Other approaches tend to combine front-capturing and front-tracking methods, such as the particle level set method.

1.2 Research Objectives

In this dissertation, we propose a front-tracking solution to generate time surfaces for the purpose of computing and visualizing Ottino's mixing indicators and analyzing mixing quality of several CFD datasets. In this dissertation, we do the following:

- Generate an initial seed surface, then track its evolution along the pathlines this surface would sweep as seed surface is mixing in the fluid. We will explore different types of seed surfaces.

- At each timestep, the surface is examined and refined wherever necessary. Our method examines the edges of the triangles on the surface and performs one of three basic re-meshing operations: edge split, edge flip, and edge collapse. We use a new approach of inserting vertices based on a new refinement criterion we call vector angle. We examine the angle between the two vectors at a given edge, and we split the edge if the vectors are diverging from one another.

- Compute approximations of Ottino's mixing indicators, by taking advantage of the geometric information at our disposal in the surfaces that we generated, and map them onto the surfaces.
- Compute curvature and torsion of the vector field and use a multiscale color map approach [19] to render regions of high, medium, and low mixing.
- Test and analyze our methods by running our implementation on various CFD datasets such as the stirred tank simulation, a BP oil spill simulation, and other synthetic flow fields.

This document contains four major chapters excluding this chapter and the conclusion. In chapter 2, we introduce the main concepts used throughout this work, notably the definitions and interpretations of Ottino's mixing indicators. We also introduce the body of previous work in the area of generating dynamic surfaces. Then in chapter 3, we delve in the methodology of generating time surfaces focusing on aspects such as dataset models, numerical schemes, and computation of mixing indicators. Chapter 4 focuses on all aspects and details of surface refinement. In this chapter, we also introduce the new vector angle criterion. In chapter 6, we present time surfaces that we generated for two real-world CFD dataset: the stirred tank dataset, and the oil spill simulation dataset. We also compute and rendering the mixing indicators for these datasets. We conclude with a summary of main contributions and benefits of this thesis.

Chapter 2
Concepts and Related Work

In this chapter, we describe the fundamental concepts used throughout this thesis. We begin by describing the visualization pipeline which is the sequence of steps we follow to transform simulation results, or a dataset, into a visual representation that allows us to understand and interpret the results. Then, we explain what a time surface is and give all the definitions of the flow mixing quantities that we considered in this research. Finally, we go over the literature on surface refinement and dynamic surfaces.

2.1 Visualization Pipeline

The visualization pipeline refers to the stages necessary to visualize some dataset. These stages are:

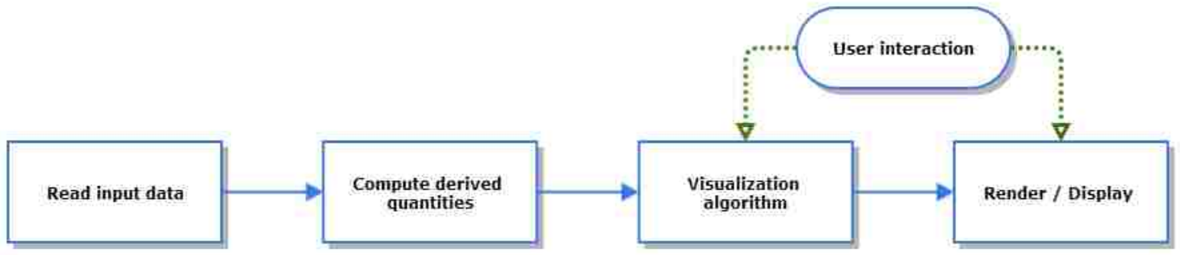


Figure 2.1: Visualization pipeline.

- **Read the input data:** this step involves converting some input dataset to a format the visualization environment recognizes. After the dataset is readable, it is stored in the dataset structures needed by the visualization algorithm.

- **Computation of derived quantities**: some visualization algorithms require further information about the data before rendering it. For example, if a velocity is provided and we would like to examine its magnitude, we would have to compute the magnitude of the vector field prior rendering. Another example would be if we were interested in analyzing some scalar field with a colormap, typically, we would need to find the maximum values of that scalar field before hand.

- **Visualization Algorithms**: given certain parameters such as derived quantities, the grid, the vector field, and so on, the visualization algorithm will take these parameters and transform them into a geometric representation that the user finds useful in understanding and analyzing the data. Visualization algorithms can be classified according to the type of data we supply them; i.e., scalar, vector, and tensor visualization methods.

- **Rendering/Display**: the last stage of the pipeline is to render the geometric representations using a graphics API such as OpenGL.

- **User interaction**: the user can examine the visualization by rotating, zooming, and modifying the visualization parameters.

Any visualization method, including the time surfaces method we introduce in this thesis, is subject to the same visualization pipeline stages.

2.2 Integral Line Mathematics

In flow visualization, it is customary to look at the following types of integral lines: pathlines, streamlines, streaklines, and time lines.

- A **pathline** is the trajectory of a massless particle. In unsteady flow, pathlines of a number of particles can cross repeatedly [31].

- A **streamline** is a line drawn in the flow field such that it is tangent to the velocity field. Streamlines can never cross except at critical points [31].

- A **streakline** is the locus of a series of particles which have been released sequentially from a fixed location.

- A **timeline** is the locus of a set of particles swept by a seed curve over time.

In steady flow, pathlines, streamlines, and streaklines are all equivalent.

For pathlines and streamlines, we obtain them by solving the following equation:

$$\frac{d}{ds}p(s) = \vec{V}(p(s)), \quad p(s_0) = p_0 \tag{2.1}$$

where

$\vec{V}$ is the vector field at point $p(s)$,

s is an integration parameter that is equivalent to time in the case of pathlines, and

p_0 is the initial position of the integral line.

Similarly to timelines, a ***time surface*** is the locus of a set of surfaces swept by a seed surface over time (Figure 2.2).

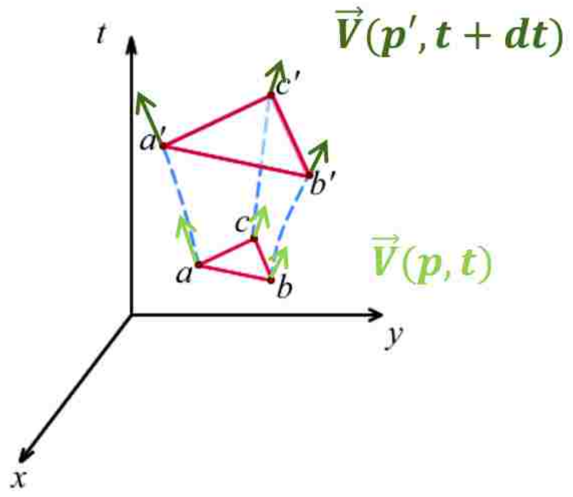


Figure 2.2: Time Surface

2.3 Quantitative Mixing Criteria

Ottino defines mixing of fluids as the process of stretching and folding of fluid filaments that spread throughout space, accompanied by breakup if the fluids are sufficiently different [[23], p 14]. This leads to the creation of interfacial area: stretching of lines in 2D and stretching of surfaces (volumes) in 3D [32]. Mixing depends on the rheology of the fluids. In the case where two fluids stretch and fold, but never breakup due to the fluids' similarities and negligible interface tension

[[23], p 2], Ottino refers to such mixing as topological mixing. If fluid filaments fragment or even coalesce, the mixing is non-topological [[23], p 19].

For a given fluid filament element, Ottino defines the length stretch as [[23], p 31]:

$$\textbf{length stretch } \lambda(t) = \frac{|dX|}{|dX_0|} \tag{2.2}$$

where $|dX_0|$ is the length of the fluid element at time $t = 0$, and $|dX|$ is the length of the fluid element at time t. See Figure 2.3.

Ottino also defines the area stretch and volume stretch of a fluid element as [[23], p 31]:

$$\textbf{area stretch } \eta(t) = \frac{|dS|}{|dS_0|} \tag{2.3}$$

$$\textbf{volume stretch } \upsilon(t) = \frac{|dV|}{|dV_0|} \tag{2.4}$$

where

$|dS_o|$, and $|dS|$ are the areas of a planar fluid element at time 0, and time t respectively,

$|dV_o|$, and $|dV|$ are the volumes of a fluid element at time 0, and time t respectively.

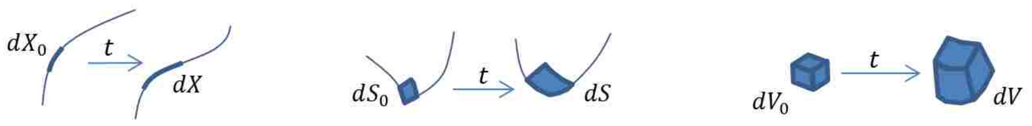


Figure 2.3: Fluid element deformation.

Furthermore, to better understand how the rate at which stretching (mixing) occurs, Ottino introduces the following quantities [[23], p 35]:

$$\textbf{rate of length stretching}: \ \frac{D\ln(\lambda)}{Dt} \tag{2.5}$$
$$\textbf{rate of area stretching}: \ \frac{D\ln(\eta)}{Dt} \tag{2.6}$$
$$\textbf{rate of volume stretching}: \ \frac{D\ln(\upsilon)}{Dt} \tag{2.7}$$

where $\frac{D()}{Dt} = \frac{\partial}{\partial t} + \vec{V}.\nabla()$ is the material derivative [[23], p 21].

Analysis of these rates will allow us to understand when the maximum amount of stretching occurred and how fast or how slow it took place.

Another quantity that is interesting to observe is the curvature of the surface. For example, mean curvature κ_H, which indicates the degree of flatness of a surface [13], can be used to indicate regions of a surface where fluid elements are experiencing strong straining. There are several ways to compute mean curvature. One way is to compute it from the principal curvatures κ_1 and κ_2, which represent the maximum and minimum curvature values of all the possible curves on S passing at a point $P \in S$ (see [10]):

$$\kappa_H = \frac{1}{2}(\kappa_1 + \kappa_2) \tag{2.8}$$

Another definition is that the mean curvature is the divergence of the normal $\vec{N}$ at point P ([13], p 12):

$$\kappa_H = \vec{\nabla}.\vec{N} \tag{2.9}$$

Other types of quantitative mixing criteria deal directly with the curvature and torsion of the vector field. Let $\vec{L}$ be the tangent curve (or streamline) of a vector field $\vec{V}$. A streamline is unique to a vector $\vec{V}$. The curvature κ and τ of a vector field are defined as [37]:

$$\kappa = \frac{|\ddot{\vec{L}} \times \dot{\vec{L}}|}{|\dot{\vec{L}}|^3} \tag{2.10}$$

$$\tau = \frac{det[\dot{\vec{L}}, \ddot{\vec{L}}, \dddot{\vec{L}}]}{|\ddot{\vec{L}} \times \dot{\vec{L}}|^2} \tag{2.11}$$

A high value of curvature indicates more straining of the fluid elements associated with it. The torsion is a measure of twisting strains on the fluid elements which also promotes the break up of clumps and enhances mixing. Torsion can be positive or negative, but the degree of mixing in the stirred tank depends primarily on the magnitude of the torsion rather than its sign.

2.4 Related Work

There are several aspects we deal with when generating time surfaces. We start by advecting the particles located on a seed surface using pathlines. Next, we re-mesh the new triangular surface using split, flip, and delete operations. This new surface is advected again then refined where necessary. These two steps are repeated until the given time period has elapsed. Our work draws from several approaches inspired from stream surfaces and streak surfaces generation. With respect to surface refinement, there are several works related to triangular mesh optimization. Finally, there are alternate methods such as the level-set methods and particle-based methods:

2.4.1 Front-Tracking Methods

One of the earliest works related to stream surface generation and that inspired our early prototype was done by Hultquist. Hultquist introduced an algorithm that constructs stream surfaces by generating triangular tiles of adjacent streamlines or stream ribbons. In Hultquist's algorithm, tiling is done in a greedy fashion. When forming the next triangle, the shortest leading edge is selected out of the two possible trailing triangles and appended to the ribbon. Each ribbon forming the stream surface is advanced until it is of equivalent length to its neighboring ribbon along the curve they share [17]. Particles are added to the trail of the stream surface by splitting wide ribbons, and particles are removed from the stream surface by merging two narrow (and adjacent) ribbons into one. Note that Hultquist's algorithm was developed for steady flows. Also, advancing the front of the stream surface requires examining all the trailing ribbons.

Along the same lines, Schafhitzel et al. [28] adopted the Hultquist criteria to define when particles are removed or added, but they derived a point-based algorithm that is designed for GPU implementation. In addition to rendering a stream surface, they applied line integral convolution to show the flow field patterns along the surface.

Rather than refine a stream surface when the surface becomes highly distorted, von Funck et al [36] introduced a new representation of smoke in a flow as a semitransparent surface by adjusting opacity of triangles that get highly distorted and making them fade. Throughout the evolution of the smoke surface, they do not change the mesh, but rather use the optical model of smoke as smoke tends to fade in high divergent areas [36]. However, the authors reported that this method does not work well if the seeding structure is a volume structure instead of a line structure.

In their work, Burger et al. [8] implemented interactive streak surfaces on the GPU that are continuously emitted from a line-shape seeding structure. Burger et al. presented two methods: a patch-based method and a triangular mesh-based method. They came up with novel ways to store the particles of the surface on the GPU. Their patch-based approach suffers from redundant calculations of particle advection and lacked connectivity information. In their mesh-based method they maintained connectivity information and this allowed them to enforce better mesh refinement or coursing criteria. Among these criteria were local curvature of along a timeline, particle splitting based on fitting a cubic polynomial, and streak line refinement [8]. The authors have reported that their mesh-based implementation can lead to distorted triangles in highly diverging flow regions [8].

Similarly to [8], McLoughlin et al. [22] focused on a quad-based approach but maintained connectivity information and were able to generate stream and time surfaces more easily than using a triangular mesh. The advantages of this method are that edges of the quad mesh are aligned with flow lines and require less processing time. As a bonus, generating time lines and rendering them was easy with this representation. The splitting and merging criteria were similar to Hultquist's method. When streamlines curved, McLoughlin et al. adjusted the integration step size to better reflect curving surfaces. The only seeding structures they supported were curves.

Another work that we modeled our implementation after was by Krishnan et al. Just like Krishnan et al., our method decouples surface advection from surface refinement [21]. The vertices of a seed surface or particle are advected using an adaptive fifth order Runge-Kutta solver [21]. For the surface refinement, they used three types of mesh operations: edge split, edge flip, and edge collapse [21]. While they used a highly accurate integration scheme, their edge splitting operation introduces numerical errors because they insert new vertices midway of a large edge. To minimize the error, they worked with high resolution meshes.

In most of the stream surface work, only the advancing front has to be updated. Our work is similar to generating streak surfaces because when computing a new time surface, the entire previous time surface has to be examined. In most of the works we have encountered ([17], [22], [28], and [21]), insertion of a new vertex is done using linear interpolation when splitting an edge, thus introducing numerical errors in the generated time surface. Finally, all the seeding structures used in these works were curves, except for [21], where for their time surfaces they used a rectangular seeding structure. In our work, we use closed seeding surfaces such as a surface of a sphere or a tetrahedron.

2.4.2 Triangular Mesh Optimization

In their work, Hoppe et al. attempted to derive a well-conditioned mesh that has fewer vertices than the original mesh by minimizing an energy function that models tight geometric fit and compact representation [16]. The new mesh is also topologically equivalent to the original mesh. The energy function in question is given by (see [16]):

$$E(K,V) = E_{dist}(K,V) + E_{rep}(K) + E_{spring}(K,V) \tag{2.12}$$

where

E_{dist} is the sum of the square distances from the points in mesh M to mesh M_0,

E_{rep} penalizes meshes with a large number of vertices and is proportional to the number of vertices,

E_{spring} is a spring term that guarantees the existing of a minimum.

To solve this minimization problem, they performed an inner minimization of V over a fixed K, then an outer minimization over K. In the outer minimization problem, the mesh is alerted by either an *edge flip*, and *edge split*, or an *edge collapse*. Then, the inner minimization problem takes care of improving the vertices that have been repositioned. The mesh alteration is accepted if the energy function decreases after the move. For an edge split, the energy function is evaluated and the initial point is chosen to be the midpoint of the splatted edge. For an edge flip, the neighborhoods of the vertices of the newly flipped edge have to yield a smaller energy than the un-flipped edge did. And finally, for an edge collapse, three initial vertices are considered: the two vertices of the collapsed edge and its midpoint. The vertex that yields the smallest energy is used.

Another approach to mesh optimization is anisotropic mesh adaption. Jiao et al. computed the Riemannian metric tensor by approximating it with a quadric-based surface analysis introduced by Heckbert et Garland [18]. Jiao et al. optimized anisotropically a mesh using vertex redistribution and edge flipping operations. They also used edge splitting and edge contraction. It should be mentioned that they considered surfaces that are static or that are dynamic and do not change topology while being evolved.

Taubin introduced a low-pass filter smoothing method for a surface with faceted features in [33]. Taubin used a modified Gaussian filter that does not cause surface shrinkage like previous Gaussian techniques. Both the new Gaussian and old Gaussian methods require vertex displacement repeatedly for a number of times. For each vertex, an average weighted displacement vector is computed using the neighboring vertices of the vertex being corrected. Instead of using one scaling factor to adjust the displacement vector, Taubin used a positive factor and a negative factor. These factors are referred to as a pass-band and stop-band [33]. In each iteration, these factors are used alternatively until convergence.

Another work used to remove noise from a rough surface was introduced by Desbrun et al. through diffusion and curvature flow [9]. Taubin showed that solving the diffusion equation is equivalent to Gaussian filtering and can reduce shrinkage [9], so the first method they tried was to solve the diffusion equation using implicit integration. The diffusion requires tuning and it depends heavily on the mesh to ensure a stable solver and non-shrinkage. The second method they tried was a combination of their first method with volume preserving. They do so by scaling all the vertices by a factor of $\beta = (V^0/V^n)^{1/3}$ after the n^{th} integration step. Although this method is better, they observed poor approximations where the surface is flat. In their final method, they used curvature flow which moves the vertices along the normal $\vec{n}$ with a speed equal to the mean curvature $\overline{\kappa}$, taken to be the average of the principle curvatures; i.e., $(\kappa_1 + \kappa_2)/2$. The mean curvature flow equation is:

$$\frac{\partial x_i}{\partial t} = -\overline{\kappa}_i \vec{n}_i \tag{2.13}$$

Of all the methods they tried, this last one yielded the best results.

2.4.3 Front-Capturing Methods

One of the early methods to capture a dynamic free boundary surface is the Volume of Fluid (VOF) method, introduced by Hirt and Nichols [15]. In this method, each cell is assigned a percentage F indicating how much of the fluid is enclosed in the interface we wish to capture. So, 1 indicates a filled cell, 0 indicates an empty cell, and any value in between indicates the interface intersects a cell. F is a step function [15]. F has to satisfy:

$$\frac{\partial F}{\partial t} + \vec{V}.\nabla F = 0 \tag{2.14}$$

Since the interface is approximated as horizontal or vertical straight line segments crossing the cells, a better representation of the interface is obtained by computing how these line segments slope such that the volume is better preserved at the cells containing the interface. This method preserves the volume of the fluid, but yields a faceted surface.

In 1988, Osher and Sethian introduced the Level Set method. In this method, a surface is implicitly represented by the zero isocontour of some level set function $\phi(\vec{x})$. Outside the interface $\phi(\vec{x}) > 0$ and inside the interface $\phi(\vec{x}) < 0$. Then the interface is advected using the famous level set equation [13]:

$$\frac{\partial \phi}{\partial t} + \vec{V}.\nabla\phi = 0 \tag{2.15}$$

The initial $\phi(\vec{x})$ is chosen to be a signed distance function, so that $|\phi(\vec{x})| = 1$. This ensures level set is smooth and well suited for high order accurate numerical methods [11]. Unlike front tracking methods, the level set method has the advantage of not requiring topological repair of the surface; however, the level set has a tendency to lose mass / volume in under-resolved regions of the flow [13]. Another problem is that the level set function can cease to be a signed distance function quickly and requires re-initialization by either recomputing it for a new timestep [13] or using fast marching methods [30].

Particle level set introduced by Enright et al. [11] combines the level set method with Lagrangian particles to address the mass / volume loss of the level set method. They randomly introduce marker particles inside and outside the interface using the following equation (refer to [11]):

$$\vec{x}_{new} = \vec{x}_p + (\phi_{in/out} - \phi_p)\vec{N}_p \tag{2.16}$$

and advect these particles in a Lagrangian fashion. Enright et al. used these particles to correct the advected interface when it starts losing volume. Using the marker particles, they restored the level set function to a signed distance function by forcing the obtained interface to be bound between the inside and outside marker particles.

Chapter 3
Methodology

To generate a time surface, we need two inputs: an initial seed surface, and a vector field. Given a time-dependent CFD dataset, the time-dependent vector field is stored at the domain's grid points. We describe how we store the time-dependent vector field in section 3.1. The surface is evolved by computing the pathlines of the vertices of the surface at the current time. Every time we evolve the surface, we obtain a new surface that has deformed according to the vector field. We refine the surface to maintain a good geometric representation.

In this chapter, we describe the surface data structure used to store and manipulate time surfaces. In the next chapter, we explore in greater detail the surface refinement process. We will also discuss the numerical methods used to compute the pathlines (section 3.2), as well as how we interpolate the velocity field (section 3.3) from a given dataset when we compute the trajectory that a particle has traveled.

3.1 Data Models

We worked with a wide variety of datasets. Datasets are classified according to the type of grid on which a numerical simulation was performed. For a detailed description of types of datasets, please refer to [29]. A grid describes the domain in which the numerical simulation has been computed. In this work, common types of grids we use are curvilinear and rectilinear grids. Furthermore, because CFD computations tend to be executed in parallel, the computational domain is subdivided into blocks. The blocks are processed by several CPUs and the results are locally stored in the given blocks.

We tested our time surface algorithm with two CFD simulation datasets, and we used two visualization systems to generate the time surfaces. Each of these software systems use their own data model to store and manipulate the data:

- The first dataset is a simulation of mixing in a stirred tank reactor. For this dataset, we used a software called VISH [3]. VISH uses a data model called fiber bundle [2].

- The second dataset is a simulation of an oil spill in the gulf of Mexico. For this dataset, we used our own software called *Dessine-Moi*. *Dessine-Moi* uses a hybrid implementation of the CFD General Notation System, or CGNS.

VISH allowed us to interpolate the vector field in multiblock curvilinear grids, whereas *Dessine-Moi* allowed us more easy development and testing of the surface refinement. Initially, we developed *Dessine-Moi* as an alternative program to VISH that enabled us to easily interface our time surface refinement code with the Microsoft Windows console debugger (CDB). This allowed us to improve our surface refinement code. Gradually, we added more functionality to our software such as analytical grids, analytical vector fields, new seed surfaces, pathlines, vector orthoslices, and so forth. The result is a simpler and more user-friendly software alternative. In chapter 5, we describe in more detail the design and implementation of *Dessine-Moi*.

3.1.1 Fiber Bundle

Fiber bundle [2] is a data model used to store and manipulate complex scientific datasets. Fiber bundle is a hierarchical model consisting of seven levels. These levels, which constitute a bundle, are Slice, Grid, Skeleton, Representation, Field, Fragment, and Compound. The Field represents arrays of primitive data types, such as int or double, and the collection of Fields describes the entire Grid. The Grid objects for different time slices are bundled together and are represented as a Bundle.

As an example of our implementation, each Field contains values of a property such as coordinates, connectivity information, and velocity (Figure 3.1). The collection of these Fields is a Grid object, and the collection of Grid objects for all time slices is the Bundle of the entire dataset.

In the stirred tank dataset, the input vector field is fragmented and these fragments are the 2088 blocks of the Grid (Figure 3.1). The input dataset for each time slice consists of coordinate location, pressure and fluid velocity for each grid point in the entire 2088 blocks. These properties are stored as Fields in the Grid object for each time slice, and these Grid objects are then combined into a Bundle.

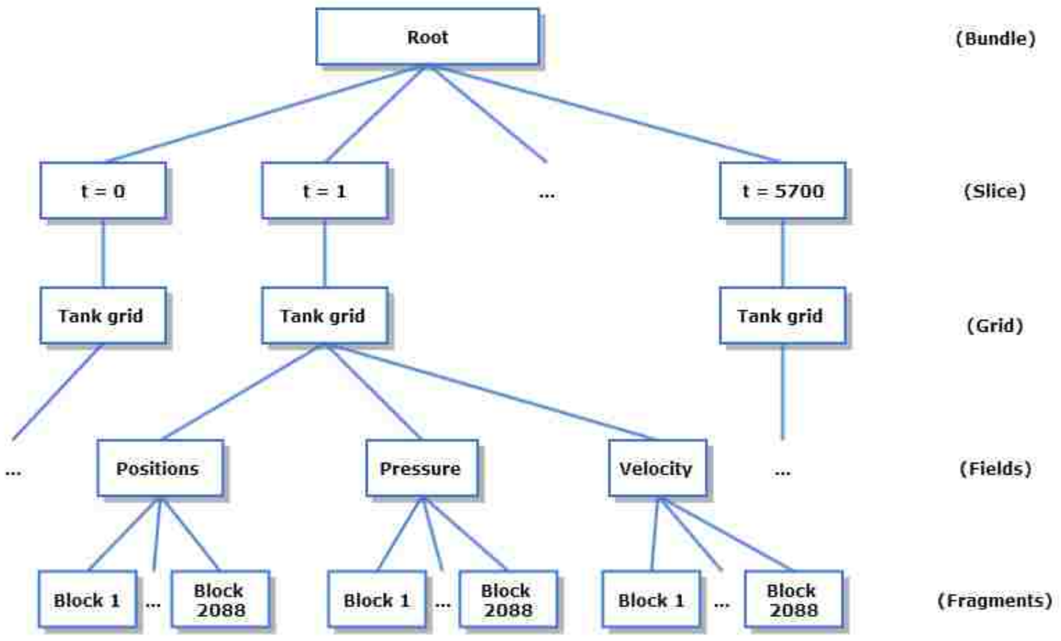


Figure 3.1: Fiber bundle data model

3.1.2 Hybrid CFD General Notation System

To store the previously discussed CFD datasets and to process them in *Dessine-Moi*, we developed a data model that is inspired by the CFD General Notation System (CGNS) [25]. CGNS is a standard which describes conventions for storage and retrieval of CFD data. CGNS was originally developed in 1994 by Boeing and NASA in an effort to standardize CFD input/output [27]. CGNS is an American Institute of Aeronautics and Astronautics (AIAA) recommended practice. CGNS is used to model a wide variety of fluid flow simulations, and the format is extensively documented online[1]. Considering how extensive this data model is, we decided to develop a simplified version that we could integrate more easily into the *Dessine-Moi* software. This allowed us to unify the way data are exchanged among the different modules in our software.

Just like fiber bundle, the CGNS data model is also hierarchical. Our hybrid model closely matches the organization of the CGNS objects[2], but stores less meta data. A hybrid-GCGNS dataset is a collection of blocks known as *Zones*, and the list of timesteps called *TimeSeries*. If the

[1] http://cgns.sourceforge.net/

[2] CGNS objects are commonly referred to as nodes in the CGNS online documentation.

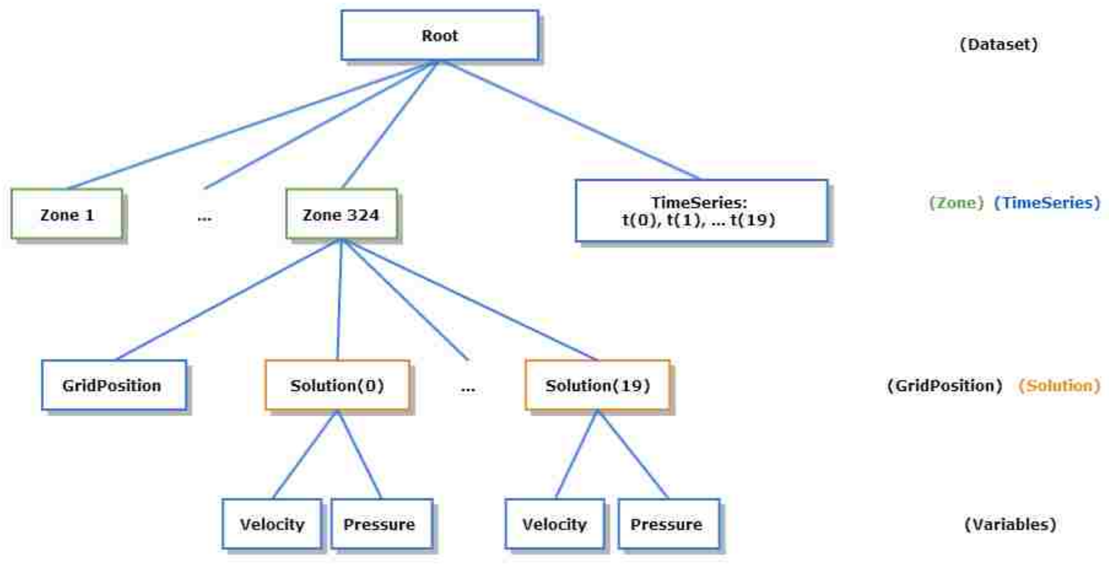


Figure 3.2: Hybrid-CGNS dataset of the oil spill simulation.

data are not multi-block, then the data are stored in the first Zone. A Zone consists of a description of the grid stored in *GridPositions*, and a collection of solution variables called *Solutions*. If the grid of a time-dependent dataset is static, we store a single GridPosition object per Zone. However, if the grid is dynamic, we store *n* GridPosition objects, where *n* is the number of time steps. Also, the number of Solution objects matches the number of time steps. Every Solution object stores one or more variables. At the moment, we only support scalar and vector variables.

In Figure 3.2, our oil spill dataset is modeled in our hybrid-CGNS format. The dataset consists of 324 blocks (Zones), and 20 time steps which are stored in the TimeSeries object. Each Zone consists of a GridPosition object that describes the grid coordinates of that particular block of the domain and the 20 Solution objects of that Zone. Within each Solution object, we have the local velocity and pressure data.

We were able to easily map the stirred tank simulation dataset to this hybrid-CGNS data model. We would have a similar diagram as in the Figure 3.2. We would have 2088 zones, spanning over more than 5000 time steps. Each Zone has only one GridPosition object. In addition, a zone will have over 5000 Solution objects that store the pressure and velocity variables at the local block.

3.2 Numerical Integration

At each timestep, a surface is made of a set of vertices, or points. Each point p is advanced to the next timestep according to integral line equation 2.1. Since the motion of a point p on a time surface is akin to its motion on its corresponding pathline, we can re-write equation 2.1 as:

$$p(t) = p(t_0) + \int_{t_0}^{t} \vec{v}(p,\tau)d\tau \tag{3.1}$$

Assuming we have an initial set of seed points and a vector field, we can compute pathlines by solving the equation above. In *Dessine-Moi*, we implemented three types of numerical methods: Euler, 2^{nd} order Runge-Kutta, and 4^{th} order Runge-Kutta.

Let Δt be the time increment between two consecutive time steps. Let P_n and P_{n+1} be the positions of point p at time t and $t+\Delta t$, respectively. Then, let $\vec{f}(P_n,t_n)$ be the vector field at point p and at time instance t. We now define the three following numerical integration methods [7]:

First-order accurate Euler's method:

$$P_{n+1} = P_n + \Delta t.\vec{f}(P_n,\ t_n) \tag{3.2}$$

Second-order accurate Runge-Kutta's method:

$$k_1 = \Delta t.\vec{f}(P_n,t_n) \tag{3.3}$$

$$P_{n+1} = P_n + \Delta t.\vec{f}\left(P_n + \frac{k_1}{2},\ t_n + \frac{\Delta t}{2}\right) \tag{3.4}$$

Fourth-order accurate Runge-Kutta's method:

$$k_1 = \Delta t.\vec{f}(P_n,\ t_n) \tag{3.5}$$

$$k_2 = \Delta t.\vec{f}\left(P_n + \frac{k_1}{2},\ t_n + \frac{\Delta t}{2}\right) \tag{3.6}$$

$$k_3 = \Delta t.\vec{f}\left(P_n + \frac{k_2}{2},\ t_n + \frac{\Delta t}{2}\right) \tag{3.7}$$

$$k_4 = \Delta t.\vec{f}(P_n + k_3,\ t_n + \Delta t) \tag{3.8}$$

$$P_{n+1} = P_n + \frac{(k_1 + 2(k_2 + k_3) + k_4)}{6} \tag{3.9}$$

3.3 Vector Field Interpolation

Often when solving pathline equation 3.1, the vector field at point p is not directly available from the dataset. Therefore, we cannot proceed with the numerical integration until we interpolate the vector field in space, at point p. Given the unit grid cell (Figure 3.3) defined by the grid positions $p_0,\ p_1,\ p_2,\ p_3,\ p_4,\ p_5,\ p_6$, and p_7 surrounding point p, let $\vec{v}_0,\ \vec{v}_1,\ \vec{v}_2,\ \vec{v}_3,\ \vec{v}_4,\ \vec{v}_5,\ \vec{v}_6$, and $\vec{v}_7$, the interpolated vector $\vec{v}_p$ is given by the following tri-linear interpolation formula [29]:

$$\begin{aligned}\vec{v}_p = \vec{v}_0(1-x)(1-y)(1-z)\ +\ \vec{v}_1(1-x)y(1-z)\ +\\ \vec{v}_2\, xy(1-z)\ +\ \vec{v}_3\, x(1-y)(1-z)\ +\\ \vec{v}_4(1-x)yz\ +\ \vec{v}_5(1-x)(1-y)z\ +\\ \vec{v}_6\, x(1-y)z\ +\ \vec{v}_7\, xyz\end{aligned} \tag{3.10}$$

where $(x,\ y,\ z)$ are the normalized coordinates of point p relative to point p_0.

Before we can apply formula 3.10, we must first locate the grid cell which encloses point p. And since our datasets are multi-blocked, we need to search for the block containing point p to locate its corresponding grid cell. Searching for a cell in a grid (single-block or multi-block) depends on the grid type.

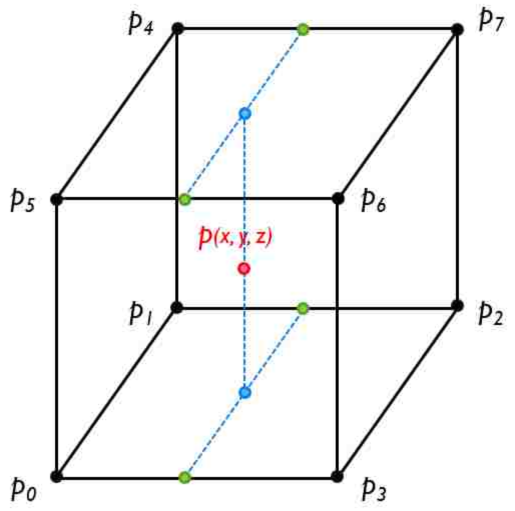
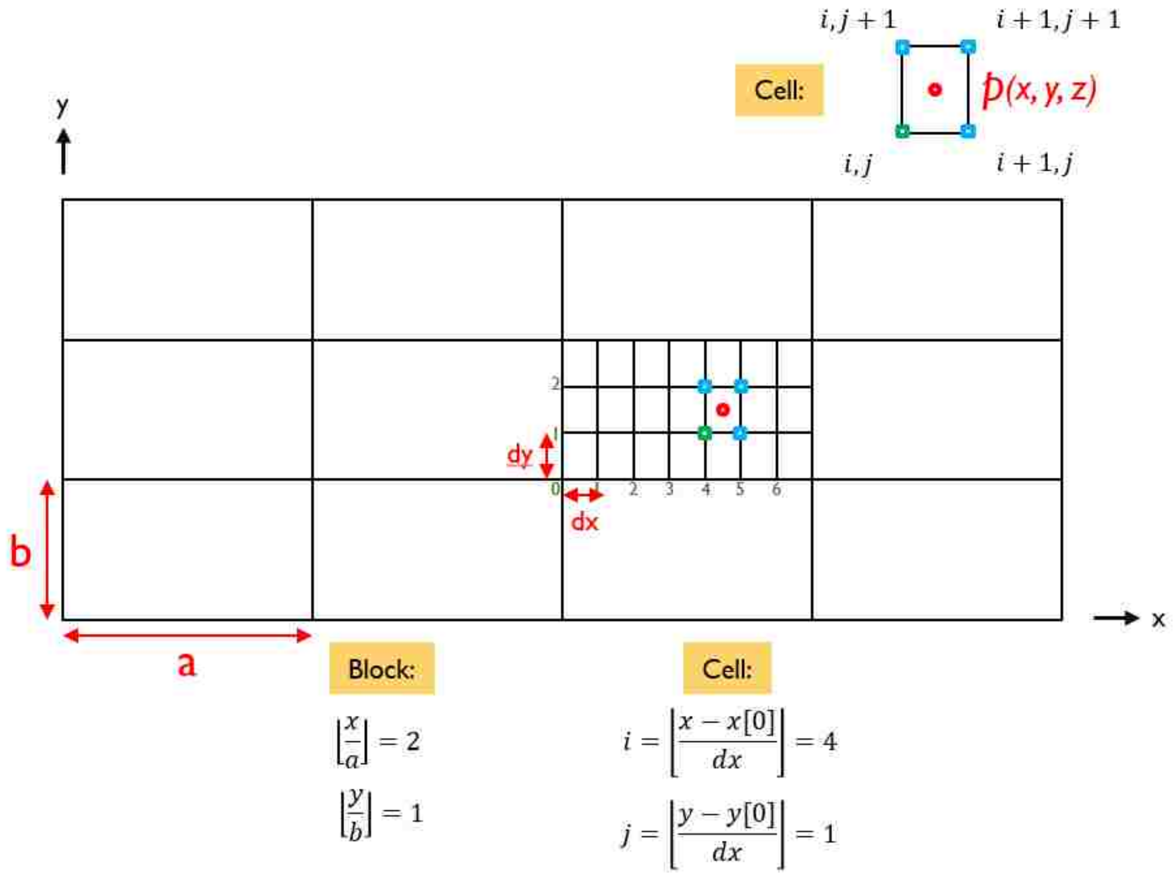


Figure 3.3: Grid cell surrounding point *p*

When a grid is curvilinear, the blocks and the cells have to be mapped to a search data structure such as an Octree, a KD-tree, or a binary sphere tree [29]. Iterating through the entire cells of a grid to find the cell of interest would be impractical. VISH supports locating blocks by building a KD-tree of the stirred tank dataset [4]. Each entry in the KD-tree consists of the center of a block and its maximal diagonal length [4]. Once a list of candidate curvilinear cells close to p are found, the next step is to find the one cell that surrounds point p. VISH uses two methods: one based on the local coordinates (Taylor approximation), and the other based on world coordinates (Direct interpolation). Refer to reference [4].

When a grid is rectilinear (and structured), the block and the cell location can easily be computed. As illustrated in Figure 3.4 with this simple 2D example, locating a block is a matter of dividing the distance of point p from the lower corner of the grid by the dimensions of the block $(a,\ b)$. Then once we locate the block, we divide the distance of point p from the lower corner of the block by the grid resolution $(dx,\ dy)$. Since block and grid position indexing start at 0, we use the floor function to determine the correct indices. This is the method we use in *Dessine-Moi* to search for grid cell surrounding a point p.

When we use the second-order accurate Runge-Kutta, or the 4th-order accurate Runge Kutta methods, we do not have the vector field at time steps $t + \Delta t/2$ in the dataset. Since the vector



$i,j+1$
$i+1,j+1$
Cell:
$p(x,y,z)$
i,j
$i+1,j$
y
x
dy
dx
b
a
Block:
$\left\lfloor \frac{x}{a} \right\rfloor = 2$
$\left\lfloor \frac{y}{b} \right\rfloor = 1$
Cell:
$i = \left\lfloor \frac{x - x[0]}{dx} \right\rfloor = 4$
$j = \left\lfloor \frac{y - y[0]}{dx} \right\rfloor = 1$

Figure 3.4: Block and cell search in a rectilinear grid.

field is not provided for such a time step, we have to interpolate the vector field in time. Another scenario that requires vector field interpolation would be when the user defines a time increment Δt_{user} that is smaller than the dataset's time increment Δt. For such cases, let t_n and t_{n+1} be the lower and upper bounds of time instance t. Also, let $\vec{f}(P,t)$ be the vector field interpolator. The interpolated vector $\vec{v}_p$ at time t is given by:

$$\vec{v}_p = \frac{\vec{f}(P,t_n)(t_{n+1}-t) - \vec{f}(P,t_{n+1})(t-t_n)}{t_{n+1}-t_n} \tag{3.11}$$

3.4 Surface Data Structures and Storage

A surface is modeled as a triangular mesh. We represent a triangular mesh M as a set of the following items:

1. A set of vertices V

2. A set of triangular faces F

3. A set of edges E

Additionally, we need the following information to perform the edge operations which we describe in chapter 4:

4. A set of edge-to-faces $E2F$

5. A set of face-to-edges $F2E$

Figure 3.5 depicts the type of data we need to store to represent and maintain a triangular mesh in our implementation. Whenever a user defines an initial seed surface, we require that they provide a triangular mesh description that includes the vertices, the faces, and the edges. When defining a triangular element, the three vertices must be specified in counter-clockwise order. Similarly, when defining a face-to-edge element, the three edges must be specified in counter-clockwise order.

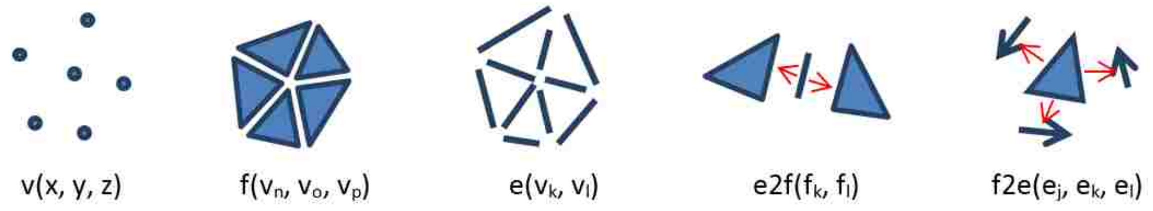


Figure 3.5: Triangular mesh data structure.

Our data are stored as an array of vertices, triangles, edge-to-faces, and face-to-edges. While there are advanced data structures to represent a triangular mesh such as the ”half-edge” data structure which uses linked lists data structures, the simplicity of our representation will allow us to parallelize surface re-meshing algorithms. Arrays are easier to map to parallel algorithms and the GPU.

After we complete generating a time surface, we store its geometry for later examination and analysis using the hybrid-CGNS model described in section 3.1. Figure 3.6 illustrates an example of a time surface with ten time steps.

The root of a time surface dataset consists of two objects: one Zone node and one TimeSeries node (Figure 3.6). Unlike the Zone objects in the oil spill simulation dataset (see Figure 3.2), the Zone object in a time surface dataset is *unstructured*. For a structured grid like the oil spill dataset, all we need are the grid coordinates which are stored in GridPosition objects. For a triangular mesh, we store the vertices of each surface in *n* GridPosistion objects, where *n* is the number of time steps. In addition, we store edges and triangles of the triangular mesh in Element objects. Note that each Element object is indexed by its type and by its corresponding time step; i.e., $(elementType, time)$. There are *n* Element objects of type *Triangle*, and *n* Element objects of type *Edge*.

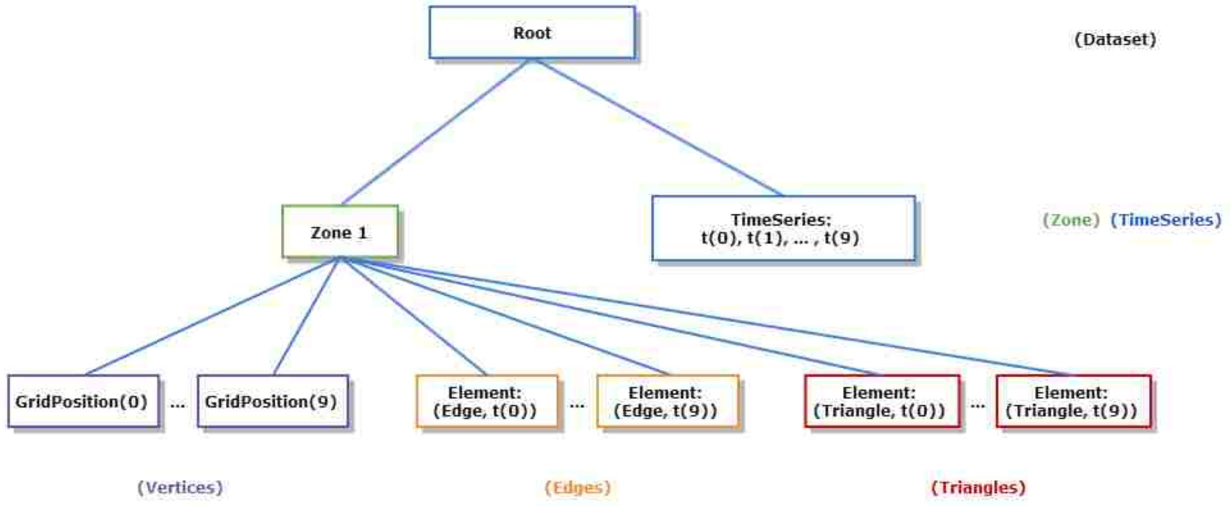


Figure 3.6: Hybrid-CGNS data model of a time surface.

3.5 Time Surface Generation

To generate a time surface, we do the following:

1. load a time-dependent dataset containing the flow field,
2. extract the velocity vector field from the dataset,
3. select an initial seed surface,

4. select the surface refinement criteria and refinement operations we want to use for the computation of the time surface,

5. compute the time surface,

6. render the time surface.

The main input parameters are the seed surface and the velocity field. For n timestep, the time surface algorithm performs two tasks (see Figure 2.2):

Algorithm 3.1: Time surface generation

```
For $i = 0$ to $n-1$, do:
    Task 1. Advect particles $p_{t_i}$ on the seed surface, yielding
        particles $p_{t_{i+1}}$ and new surface $S_{t_{i+1}}$
    Task 2. Refine surface $S_{t_{i+1}}$ according to user specified
        refinement criteria
```

In task 1, particles are advected by solving the pathline differential equation (eq. 3.1). Refer to section 3.2 for the numerical methods used.

In task 2, the triangular mesh is inspected for triangles that satisfy certain refinement criteria. These criteria are discussed in detail in section 4.3. Certain triangles will have to be updated accordingly. The edges of these triangles will be transformed in one of the following types of surface refinement (re-meshing) operations: edge split, edge flip, or edge collapse. Figure 3.7 illustrates these operations. For further details on surface refinement operations, refer to section 4.2.

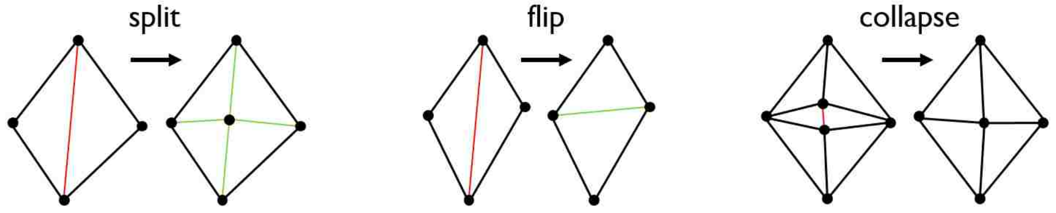


Figure 3.7: Surface refinement operations

3.6 Computation of Quantitative Mixing Criteria

We divide the quantitative mixing criteria into two categories: Ottino's mixing quantities, and vector field curvature and torsion.

3.6.1 Computation of Ottino's Mixing Quantities

Once we have a geometric description of a time surface, we can easily approximate the mixing criteria defined by Ottino [23]. To compute the area stretch $\eta(t)$ (see eq. 2.3), we need to compute the surface area of any surface in a given time surface. Similarly, to compute the volume stretch $\upsilon(t)$ (eq. see 2.4), we need to compute the volume of a closed surface.

We compute the area of a surface by simply iterating through *n* triangular elements *abc* of the surface, and summing their areas:

$$area\ of\ S(t) \ = \ \frac{1}{2}\sum_{i=1}^{n} |\vec{a_i b_i} \times \vec{a_i c_i}| \tag{3.12}$$

To compute the volume of a closed surface, we use Gauss' Divergence Theorem which enables us to evaluate a volume integral as a surface integral:

$$\iiint_V \vec{\nabla}\cdot\vec{F}\ dV = \oiint_S \vec{F}\cdot\vec{n}\ dS \tag{3.13}$$

where $\vec{F}$ is a continuous differentiable function.

We choose $\vec{F} = (x,\ 0,\ 0)$. Thus, $\vec{\nabla}\cdot\vec{F} = 1$ and $\vec{F}\cdot\vec{n} = x\ n_{x_{comp}}$. Hence, the volume of a closed surface can be evaluated as follows:

$$\begin{aligned} volume\ of\ S(t) &= \iiint_V 1\cdot dV = \oiint_S x\ n_x\ dS \\ &\approx \frac{1}{2}\sum_{i=1}^{n} x_c\ n_{x_i}\ |\vec{a_i b_i} \times \vec{a_i c_i}| \end{aligned} \tag{3.14}$$

where for the i^{th} triangular element *abc*:

$x_c = \dfrac{x_a + x_b + x_c}{3}$ is the *x*-component of the centroid of triangle *abc*, and

$\vec{n}_i$ is the normal of triangle abc.

Finally, to compute the rate of area stretch (eq. 2.6) and the rate of volume stretch (eq. 2.7), we use finite differencing [12] to compute the derivatives. At the lower boundary of area/volume stretch values, we use forward differencing. At the mid-values, we use central differencing. And at the upper boundary values, we use backward differencing. The mid values are second-order accurate, whereas the two boundary values are first-order accurate.

3.6.2 Vector Field Curvature and Torsion

To compute the vector field torsion τ (eq. 2.11) and curvature κ (eq. 2.10), we compute $\dot{\vec{L}}$, $\ddot{\vec{L}}$, and $\dddot{\vec{L}}$ as follows:

$$\dot{\vec{L}} = \vec{V} \tag{3.15}$$

$$\ddot{\vec{L}} = u\vec{V}_x + v\vec{V}_y + w\vec{V}_z \tag{3.16}$$

$$\begin{aligned}\dddot{\vec{L}} = &(uu_x + vu_y + wu_z)\vec{V}_x \\ &+ (uv_x + vv_y + wv_z)\vec{V}_y \\ &+ (uw_x + vw_y + ww_z)\vec{V}_z \\ &+ u^2\vec{V}_{xx} + v^2\vec{V}_{yy} + w^2\vec{V}_{zz} \\ &+ 2uv\vec{V}_{xy} + 2uw\vec{V}_{xz} + 2vw\vec{V}_{yz}\end{aligned} \tag{3.17}$$

where $\vec{V} = (u, v, w)$ is the vector field (velocity).

Computation of curvature and torsion consists of two parts: 1) computation of the curvature and torsion fields, and 2) rendering the resulting scalar fields as orthoslices using multi scale color mapping. In the first part, we read the stirred tank dataset, and do the following steps:

1. Define a 3D Cartesian grid of points in the stirred tank that will be used to calculate the curvature and torsion fields.

2. Since these Cartesian grid points are different from the curvilinear grid points where the vector field is given, we have to get the vector field at the Cartesian points by interpolation. We do this using the Direct Interpolation method which is described in [4].

3. Compute the first, second, and mixed derivatives of the vector field with respect to x, y, and z at the Cartesian grid points using finite difference formulas.

4. Use these derivatives to calculate the curvature and torsion at the Cartesian grid points. This gives us the curvature and torsion fields for the entire stirred tank.

In part two, we render the resulting scalar fields as follows:

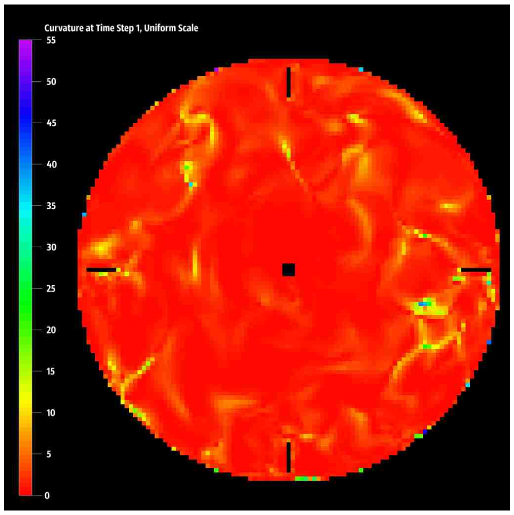
1. Construct a histogram of the scalar values, then design a multi scale color map. We show that the use of conventional uniform color coding for the whole range of these scalar values often gives a single predominant color, making it impossible to visualize variations in the scalar values. In contrast, multi scale color coding gives different colors for the small, medium, and large scalar values, thereby enabling one to visualize the entire range of these values.

2. Extract a plane (orthoslice) from the Cartesian grid on which the scalar fields are computed, and apply the multi scale color map to the orthoslice. This technique enables one to visualize the variations in the scalar values throughout the orthoslice.

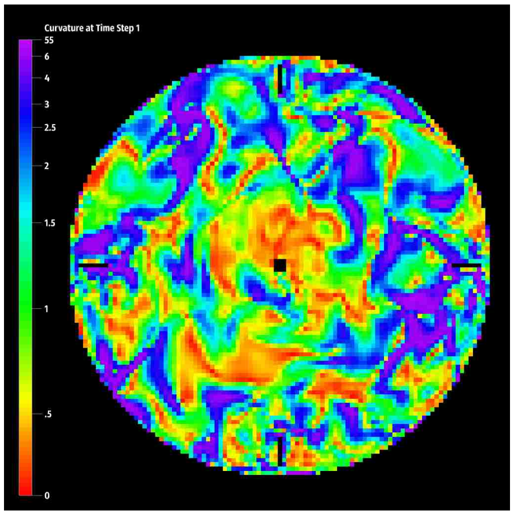
3.7 Multi Scale Color Mapping

Oftentimes, a colormap helps us visualize the range and distribution of a scalar field. However, if the range of the scalar values is highly irregular, as in the case of curvature and torsion, a conventional uniform colormap often gives a single predominant color for the most recurring values of the scalar field [6]. As a result, it becomes difficult to distinguish between the entire range of values. To address this problem, we use a multiscale color coding scheme. In designing a multi scale colormap, one has to specify (1) the number of divisions in the scale, (2) the range of the scalar values in each division, and (3) the color range for each division. We have to choose this set of parameters in such a way to visualize the entire range of curvature and torsion values.

Various techniques can be used to construct the multi scale. In the method presented here, the number of divisions in the multi scale and the range of scalar values in each division are determined manually so as to facilitate the visualization of the scalar field variations of interest. A cumulative histogram of the scalar values is used to determine the boundaries of the divisions so that each division contains the desired percentage of scalar values. The user can choose any number of divisions and specify any range of scalar values and any range of colors for each division in order to enhance the visualization of the scalar values of interest. There also exists other work in this area which involves automatic [14], semi-automatic [20] and manual generation of the functions to do this task [24].



(a) Uniform scale color coding for curvature



(b) Multi scale color coding for curvature

Figure 3.8: Comparison of uniform and multi scale color coding of an orthoslice of the curvature field.

An example of multi scale color coding is given in Figure 3.8, which compares the uniform and multi scale color coded curvature field in an orthoslice in the *xy* plane at timestep 1 in the CFD stirred tank simulations. A histogram of the curvature values, which range from 0 to more than 55, shows that about 41% of them are less than 1, 72% are less than 2, 86% are less than 3, 92%

are less than 4, 96% are less than 6, etc. The curvature values were capped at 55 so that values greater than 55 were treated as outliers and set to 55. If uniform color coding is used to display the curvature values from 0 to 55 on a color scale that goes from red to violet, most of the values are shown as shades of red and cannot be distinguished from each other, making it impossible to visualize the small, medium and large values, as shown in Figure 3.8(a).

Table 3.1: Curvature ranges and their cumulative frequencies

Division	Curvature Range	Percent Range
1	0 - 0.5	0 - 17.90
2	0.5 - 1	17.90 - 40.96
3	1 - 1.5	40.96 - 59.86
4	1.5 - 2	59.86 - 72.38
5	2 - 2.5	72.38 - 80.83
6	2.5 - 3	80.83 - 85.91
7	3 - 4	85.91 - 91.64
8	4 - 6	91.64 - 96.44
9	6 - 55	96.44 - 100

The curvature ranges and corresponding percent ranges used in this multi scale mapping were obtained manually from a cumulative histogram of the curvature values. The color scale that we are using goes from red to violet as its color index goes from 0 to 1. The color range for each division is specified by converting the percent range to decimals and using these as the starting and ending color indexes for that division (e.g., the starting and ending color indexes for division 2 are 0.1790 and 0.4096). Within each division, the color index varies linearly with the curvature value:

$$c = c_1 + \frac{(v - v_1).(c_2 - c_1)}{v_2 - v_1} \tag{3.18}$$

where v is the curvature value, c is the corresponding color index, v_1 and v_2 are the starting and ending curvature values of the division, and c_1 and c_2 are the starting and ending color indexes. Thus this multi scale color coding is piecewise linear.

When this multi scale scheme is used to display the curvature in the orthoslice, one can now easily distinguish between the small, medium and large curvature values, as shown in Figure 3.8(b).

The red and orange areas, yellow and green areas, and blue and violet areas correspond to small, medium, and large curvature, respectively. Thus the red and orange areas indicate a small degree of fluid mixing while the blue and violet areas indicate a large degree of mixing.

Chapter 4
Surface Refinement

In this chapter, we discuss how we refine a surface. In the previous chapter, we defined the array-based surface data structure that we are using to represent a surface. In the next section, we begin by describing elementary data structure operations used throughout the refinement process, such as orienting the edges in a face, and constructing the edge-to-face table. Next, we will explain the three surface refinement operations: edge split, edge flip, and edge collapse. Then, we will describe how we decide when to perform a certain refinement operation. Most surface refinement techniques use the edge length as the refinement criterion. In this thesis, we introduce a new refinement criteria based on the vector angle at an edge: two converging vectors means that the edge is shrinking, whereas two diverging vectors means the edge is stretching. Next, we describe surface refinement by showing how the flip, the split, and the collapse operations work together to produce a refined mesh. Finally, we discuss how refining a surface before or after it is advected can affect the surface.

4.1 Surface Data Structure Operations

As described in section 3.4, a triangular mesh M consists of a set of vertices V, triangles F, edges E, edge-to-faces $E2F$, and face-to-edges $F2E$. Our data structure can be described as an array-based ”half edge” data structure. In the following subsections, we describe important helper operations that allow us to query a triangular mesh M. Henceforth, we will refer to a triangular element as simply a face.

4.1.1 Generating the Edge-to-Faces Table

The Edge-to-Faces table allows us to determine the left and right faces adjacent to a given edge. We use this table in many edge refinement criteria and operations. For instance, every time we need to split or flip an edge, we consult this table. Typically, we generate the edge-to-faces table only at the beginning of the time surface algorithm when the seed surface has an empty edge-to-faces table. As we refine a surface, it is more efficient to update this table while we perform an edge split or an edge flip, than to reconstruct the entire table from scratch. The algorithm below describes the steps we follow to construct the edge-to-faces table:

Algorithm 4.1: Edge-to-faces generation

```
For each edge $e$ in $E$:
  • set $facesFound = 0$
  • While $facesFound < 2$, do for each face $f$ in $F$ the following:
      ◦ If two of the vertices of $f$ match the vertices of $e$:
          • increment $facesFound$
          • save the face: $E2F[e] = f$
```

4.1.2 Orienting Edges

In the design of our data structure, we impose a counter-clockwise orientation on an edge. This allows us to find previous and next edges (section 4.1.4). Also, oriented edges help us locate faces incident to a particular vertex (section 4.2.3). Note that the orientation of an edge is relative to the face we are examining. The following algorithm takes as input a face-to-edges $f2e$ and returns it, in the correct orientation, as shown below:

Algorithm 4.2: Orienting edges in a triangle

```
1. Construct the correct sequence of edges $orientedEdges$, using the
   vertices of face $f$
2. For each edge $e$ in $f2e$:
   • retrive the vertices of $e$ from the edges table
   • save the index $i$ of $orientedEdges$ that corresponds to the
     vertices of $e$
   • build the triplet $correctOrientedEdges$ as follows:
     $correctOrientedEdges[i] = e$
3. set $f2e = correctOrientedEdges$
4. return $f2e$
```

4.1.3 Generating the Face-to-Edges Table

The following algorithm searches for the edges that are adjacent to a particular face. This algorithm depends on the edge-to-faces table. The edges are re-oriented using the algorithm in subsection 4.1.2:

Algorithm 4.3: Face-to-edges generation

```
1. construct the edge-to-face table, if it is empty
2. For each edge $e$ in $E$:
    • to the first face adjacent to $e$, do:
        ∘ $f = edge2Faces[e].i$
        ∘ $index = faceIncrement[f]$
        ∘ $face2Edges[f][index] = e$
        ∘ incrment $faceIncrement[f]$
    • to the second face adjacent to $e$, do:
        ∘ $f = edge2Faces[e].j$
        ∘ $index = faceIncrement[f]$
        ∘ $face2Edges[f][index] = e$
        ∘ incrment $faceIncrement[f]$
3. For each $f2e$ in $F2E$: re-orient the edges in $f2e$
```

Just like the edge-to-face table, we only construct the face-to-edges table at the beginning of the time surface algorithm. Also, it is more efficient to maintain the $E2F$ table while performing an edge operation than to re-compute the table all over.

4.1.4 Previous Edge and Next Edge

The previous edge of an edge e at index j in $e2f$ is:

$$previousEdge(e) = e2f[(j-1)\ mod\ 3] \tag{4.1}$$

and the next edge is:

$$nextEdge(e) = e2f[(j+1)\ mod\ 3] \tag{4.2}$$

To find index j, we loop through the three edges in $e2f$ until one of them matches e.

4.2 Surface Refinement Operations

There are three surface refinement operations: edge splitting, edge flipping, and edge collapse.

4.2.1 Edge Splitting

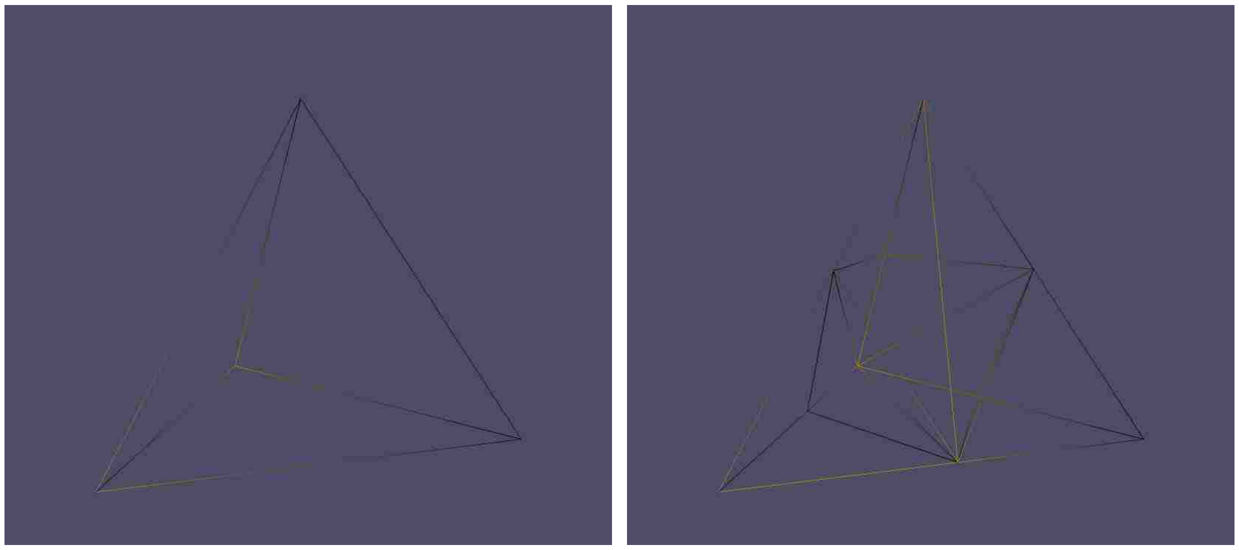
Edge splitting is an operation that divides an edge into two smaller ones, if the edge satisfies a certain refinement criterion. The algorithm to split edges in a triangular mesh that we developed is listed below:

Algorithm 4.4: Edge splitting

```
For each edge $e$ needing split:
  • insert the midpoint in edge $e$
  • For each face $f$ adjacent to $e$:
      ◦ add new edge $e_{new}$
      ◦ add new face $f_{new}$
      ◦ add new edge-to-face $e2f_{new} = (f, f_{new})$
      ◦ update the vertices of $f$ (i.e., reuse $f$ to store the new
        modifications of $f$)
      ◦ correct $e2f$ that is adjacent to $f_{new}$
  • split $e$ to $e_{left}$ and $e_{right}$ (reuse $e$ to store $e_{left}$)
  • add another new edge-to-face (i.e., $e2f = (f_{new_1}, f_{new_2})$)
```

Note that when an edge is split:

- Four triangles are obtained: two are new, and two are updated.
- One new vertex is created midway between the vertices of an edge needing to be split.
- Four new edges are obtained: three newly created, and one updated.


(a) Initial surface (b) 1^{st} split-only refinement

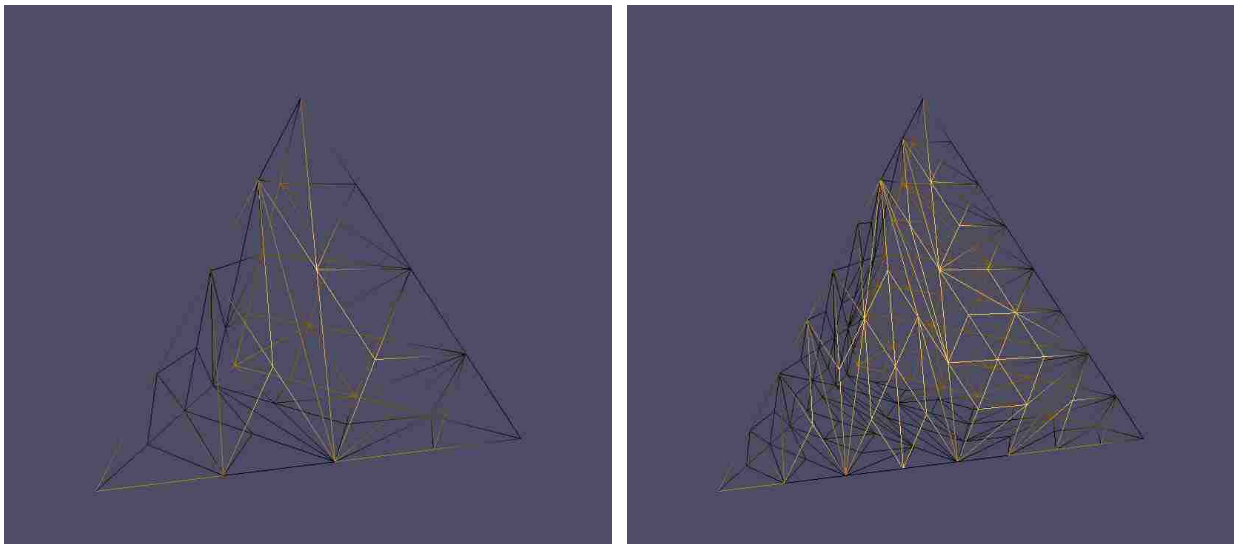

(c) 2^{nd} split-only refinement (d) 3^{rd} split-only refinement

Figure 4.1: Three successive split-only surface refinements on a tetrahedral initial surface.

In Figures 4.1, we successively refine a tetrahedral seed surface three times using only edge splitting. This Figure illustrates how the split operation works.

4.2.2 Edge Flipping

We notice from the previous section that continuously splitting an edge at each timestep can yield narrow triangles (see Figure 4.1). To remedy to this, and hence improve the quality of the mesh, we must flip edges in narrow triangles to produce well-conditioned triangles. we refer to the edges that need to be flipped as illegal edges. Unlike edge splitting; however, edge flipping can modify the topology of the surface yielding erroneous results as illustrated in Figure 4.2.

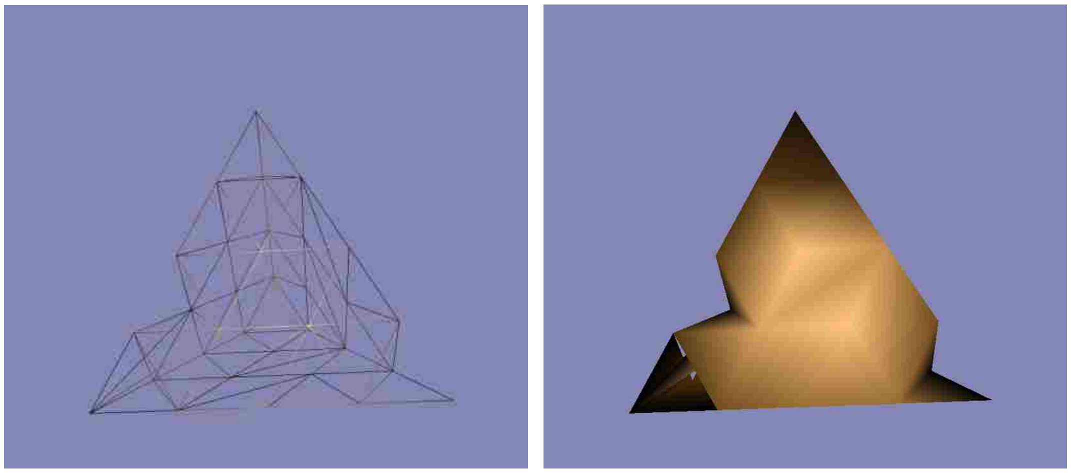


Figure 4.2: Improper edge flipping and corresponding surface.

The flip operation is divided into two parts: illegal edge detection and edge flipping operation. Note that the first part (edge flip detection) is part of the flip criterion search.

Algorithm 4.5: Edge flipping detection

```
For each edge $e$ in the mesh:
  • retrieve the adjacent triangles $(f_1, f_2)$ of $e$
  • compute flipped version of $e$
  • If $flipCriterion$ is true:
      ◦ flag $e$ for a flip operation
```

Once we determine which edges to flip, the actual flip operation can then be implemented as follows:

Algorithm 4.6: Edge flipping

```
For each flippable edge $e$ in the mesh:
  • retrieve the adjacent triangles $(f_1, f_2)$ of $e$
  • label the vertices of the triangles
  • do the flip operation: $e = (v_3, v_4)$
  • update $f_1$: $f_1 = (v_1, v_4, v_3)$
  • update $f_2$: $f_2 = (v_4, v_2, v_3)$
```

Figures 4.3 illustrates the difference after a mesh is refined using a split operation, followed by a flip operation.

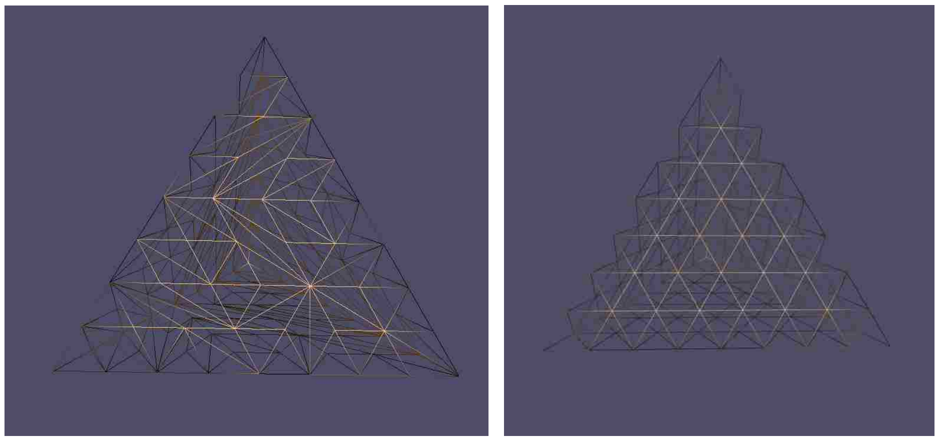


Figure 4.3: Improving a surface with edge flipping.

4.2.3 Edge Collapse

Edge collapse is an operation that removes small edges. When an edge becomes very small, its vertices can be merged into a single point, thus reducing the amount of particles that need to be advected.

Before we can collapse an edge, we search for edges that are suitable for collapse. This is described in the following algorithm:

Algorithm 4.7: Edge collapse detection

```
For each edge $e$ in the mesh:
  • If ($e$ has not been visited) and ($collapseCriterion$ is true), then:
        ○ flag $e$ for a collapse operation
        ○ mark all the edges in the neighborhood of $v_1$ and $v_2$ as
          visisted
```

Once we determine which edge to collapse, the collapse operation can then be performed as follows:

Algorithm 4.8: Edge collapse operation

```
1. For each collapsible edge $e$ in the mesh:
    • mark $v_1$ of $e(v_1,v_2)$ for deletion , and mark $v_2$ for collapse onto
      $v_1$
    • mark adjacent faces to $e(f_1,f_2)$ for deletion
    • mark the two edges emanating from $v_1$ and adjacent to $(f_1,f_2)$
      for deletion
2. For each vertex $v$ in the old mesh not marked for deletion:
    • copy $v$ in the new mesh
    • relable $v$
3. For each vertex $v$ in the old mesh marked for deletion:
    • relable $v_1$ in the new mesh as $v_2$
    • interpolate new $v_2$
4. For each edge $e$ in the old mesh not marked for deletion:
    • relable the vertices of $e$ in the new mesh
    • copy $e$ in the new mesh
    • relable $e$
5. For each edge $e$ in the old mesh marked for deletion:
    • relable the edges emanating $v_1$
6. For each face $f$ in the old mesh not marked for deletion:
    • relable the vertices of $f$ in the new mesh
    • copy $f$ in the new mesh
    • relable $f$
7. **For** each edge $f$ in the old mesh marked for deletion:
    • relable $f_1$ and $f_2$ with their neighboring faces
```

Because all mesh items are stored as arrays, we have to relable all vertices, edges, and faces into the newly constructed arrays.

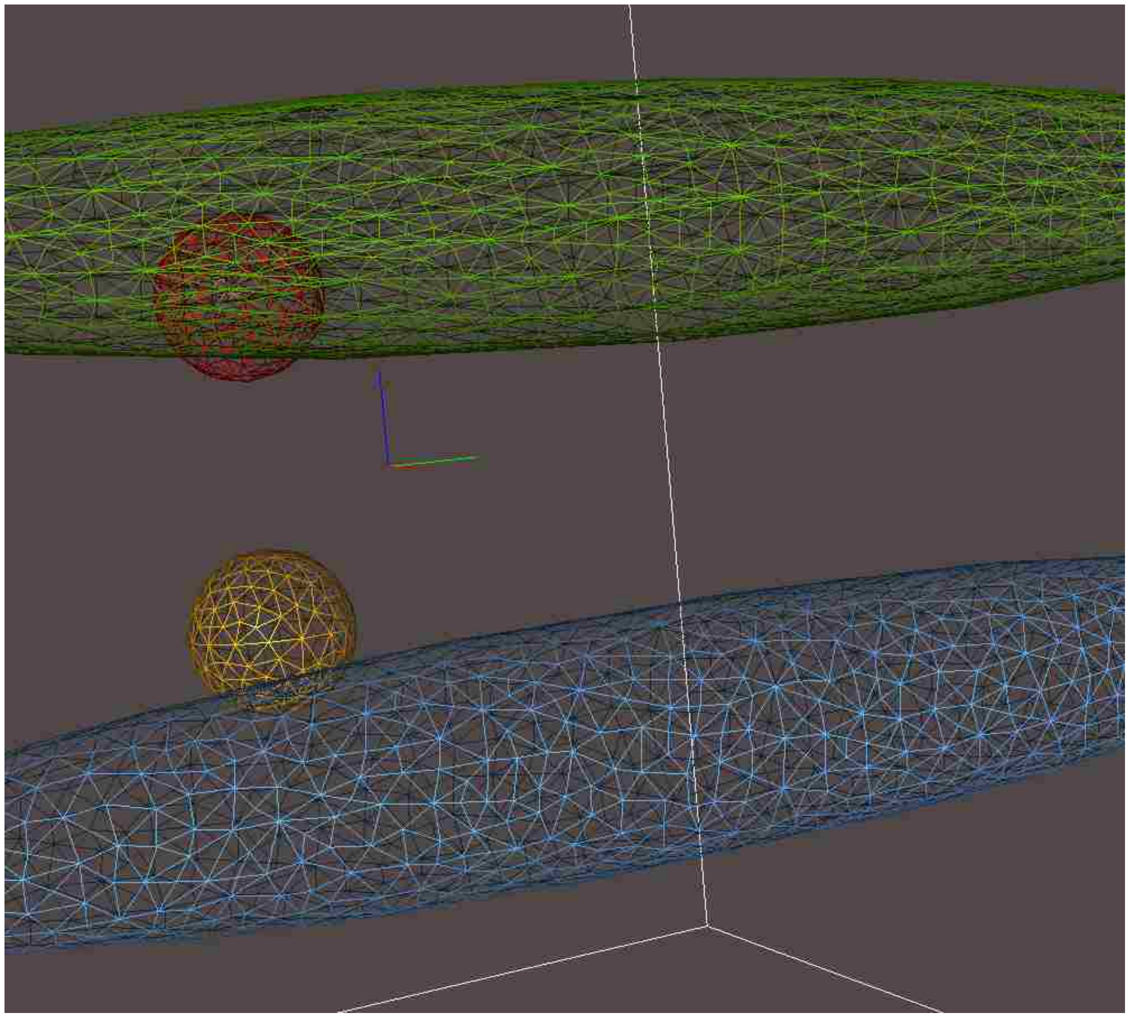


Figure 4.4: Comparison of surface refinement without (green) and with edge collapse (blue).

Figure 4.4 illustrates the benefit of using collapse from a visual point of view. The red sphere is the seed surface for the green time surface that uses only split and flip operations. The orange sphere is the seed surface for the blue time surface that uses all three surface refinement operations.

4.3 Surface Refinement Criteria

Surface refinement (or surface re-meshing) criteria are crucial to surface re-meshing operations; they dictate under what circumstances a given surface re-meshing operation can be performed. Regardless of the type of surface refinement operation, all surface refinement criteria are derived from the `SurfaceRefinementCriterion` abstract class. This base class provides a method called `IllegalEdgesCount()`, which returns the number of edges that satisfy a certain refinement criterion. In addition, every derived refinement criterion class must implement a method called `DetectEdges()`, which takes as input a triangular mesh M, and returns a boolean array of size $|E|$ that flags every edge $e \in E$ indicating whether or not e passes the specified refinement condition by the derived class. This approach allows us to easily integrate a wide variety of refinement criteria with our surface refinement algorithm. In the next subsections, we will describe some of these refinement criteria.

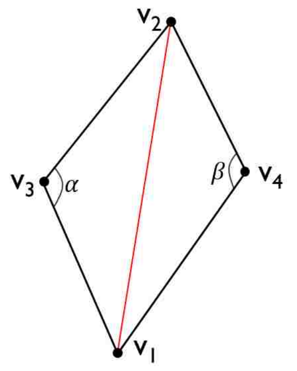
4.3.1 Edge Length

This edge length criterion apply to the edge split and the edge collapse. When an edge exceeds a certain maximum edge length threshold (l_{max}), we split that edge. The user chooses the value of the maximum length threshold. Typically, we choose l_{max} to be equal to the smallest edge in the seed surface. Similarly, when an edge shrinks beyond a certain minimum edge length threshold (l_{min}), we collapse that edge. The default value we use as minimum edge length is %15 the length of the smallest edge in the seed surface. Alternatively in [34], Tryggvason et al. used a minimum edge length threshold of $h/3$ where h is the grid resolution.

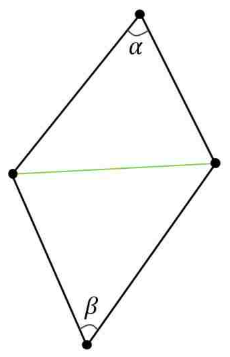
4.3.2 Hybrid-Delaunay Condition

The hybrid-Delaunay criterion is the flip condition that we use to decide to flip a particular edge. This flip condition is based on the Delaunay condition. The Delaunay condition (Figure 4.5) states that if the sum of the opposite angles α and β to a common edge e is greater than 180, we flip that edge.

Let $\vec{n}_1$ and $\vec{n}_2$ be the normals of triangles $v_1v_2v_3$ and $v_1v_4v_2$, respectively (Figure 4.5). The hybrid-Delaunay condition flips an edge when all of the following conditions are met:



(a) $\alpha + \beta > 180^o$



(b) $\alpha + \beta < 180^o$

Figure 4.5: Delaunay flipping condition.

1. **Loose Delaunay condition**: rather than computing the angles α and β, we compute their cosine values instead. This does not guarantee that an edge satisfying the Delaunay condition will be flipped. For example, if $\alpha = 150^o$ and $\beta = 60^o$, then our loose condition will not flip this edge. Rather, it will wait for the angles to grow larger. We designed this alternative approach to simplify the evaluation of the Delaunay condition. Thus, we have:

$$v_3\vec{v}_1 \cdot v_3\vec{v}_2 < 0 \quad AND \quad v_4\vec{v}_1 \cdot v_4\vec{v}_2 < 0 \tag{4.3}$$

2. **No edges with small triangles are flipped**: we define a threshold triangle area that we call *zeroArea*. We choose this threshold to be %1 of the area of the smallest triangular element. If one of the triangles $v_1v_2v_3$ or $v_1v_4v_2$ is smaller than this threshold, we do not flip the edge. Hence, we check the following:

$$(area\ of v_1v_2v_3\ >\ zeroArea) \quad AND \quad (area\ of v_1v_4v_2\ >\ zeroArea) \tag{4.4}$$

3. **Sharp edges are not flipped**: if the angle between the normals of triangles $v_1v_2v_3$ and $v_1v_4v_2$ is greater than 90^o, we do not flip the edge. In other words,

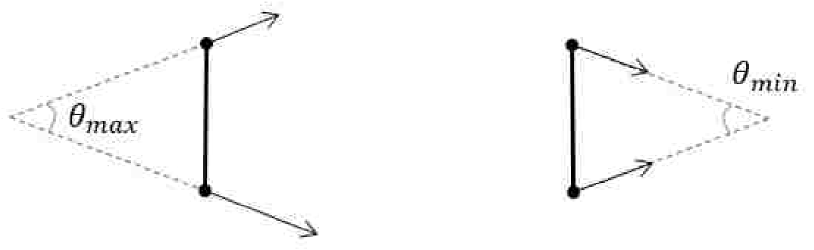
$$\vec{n}_1 \cdot \vec{n}_2 > 0 \tag{4.5}$$

4.3.3 Triangle Area

This criterion applies to the edge split. This criterion is similar to the edge length criterion. If the area of a triangle is larger than a threshold area, we split that edge.

4.3.4 Vector Angle

This criterion examines the angle between two vectors at a given edge. If the vectors are parallel, we do nothing. If the vectors are converging, we collapse the edge. If the vectors are diverging, we split the edge. To control how often the split is performed, we set a maximum angle threshold θ_{max} (Figure 4.6(a)). Likewise, for the collapse operation, we define a minimum angle threshold θ_{min} (Figure 4.6(b)).



(a) Divergence angle (b) Convergence angle

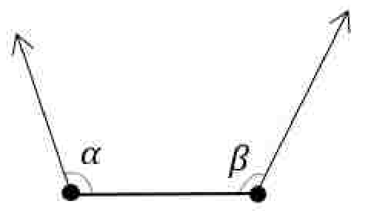
Figure 4.6: Vector-angle condition scenarios.

We *split* an edge if one of the two conditions is satisfied:

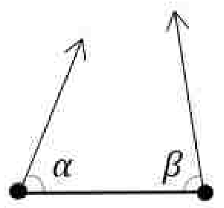
- $\alpha+\beta > \pi+\theta_{max}$: see Figure 4.7(a) and Figure 4.7(c), or
- $\alpha > \frac{\pi}{2}$ *and* $\beta = 0$: see Figure 4.7(e). In this case, one of the velocity vectors is zero.

We **collapse** an edge if the following condition is met: $\alpha+\beta < \pi-\theta_{min}$: see Figure 4.7(b), and Figure 4.7(d).

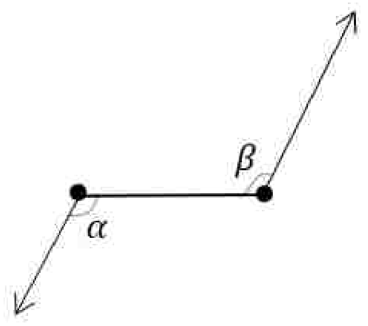
Note that there is one scenario where the vector-angle test is *inconclusive*. If one of the velocity vectors is zero and the other vector forms an acute angle with the edge (Figure 4.7(f)), we fall back on the edge length criterion.



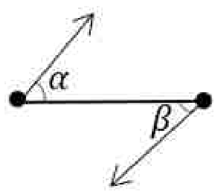
(a) $\alpha + \beta > \pi + \theta_{max} \Rightarrow stretch$



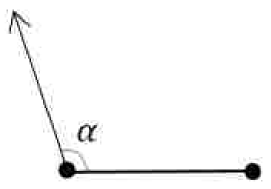
(b) $\alpha + \beta < \pi - \theta_{min} \Rightarrow shrink$



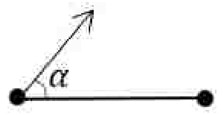
(c) $\alpha + \beta > \pi + \theta_{max} \Rightarrow stretch$



(d) $\alpha + \beta < \pi - \theta_{min} \Rightarrow shrink$



(e) $\alpha > \frac{\pi}{2} + \frac{\theta_{max}}{2}$ *and* $\beta = 0 \Rightarrow stretch$



(f) $\alpha < \frac{\pi}{2}$ *and* $\beta = 0 \Rightarrow unknown$

Figure 4.7: Vector-angle condition scenarios.

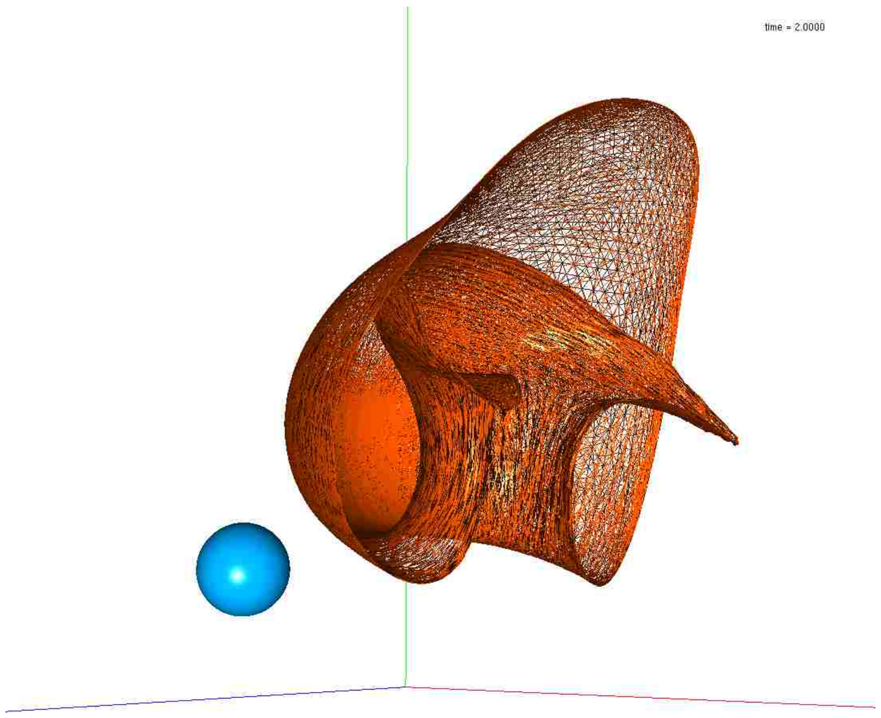
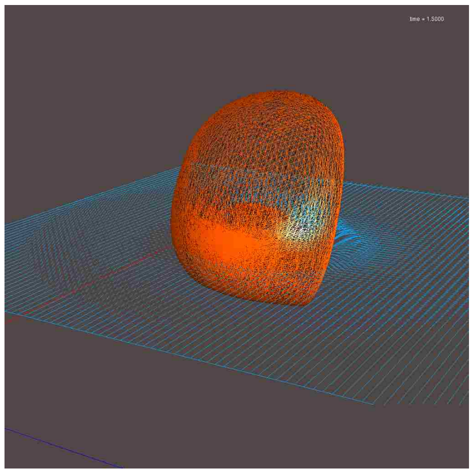


Figure 4.8: Time surface (orange) at t = 2 using a $\theta_{max} = 4$ and corresponding seed surface (blue sphere).

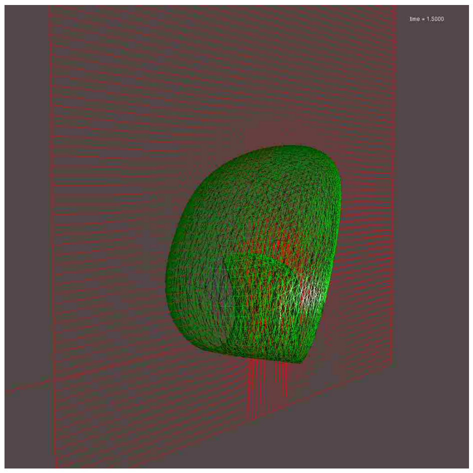
4.3.5 Vector Angle Criterion Example

As an example, we used the flow field of an oil spill simulation dataset (see section 6.2). We placed a sphere of radius equal to 0.05 and maximum edge length equal to 0.005 at location $(0.2, 0.15, 0.5)$ in the grid (Figure 4.8). We generated two time surfaces for the time interval $[0.1, 2]$ using a time increment of $\Delta t = .025$. The first time surface (green, Figure 4.9) was generated using and edge length threshold equal to 0.1. The second time surface (orange, Figure 4.9) was generated using a maximum threshold angle equal to 4.

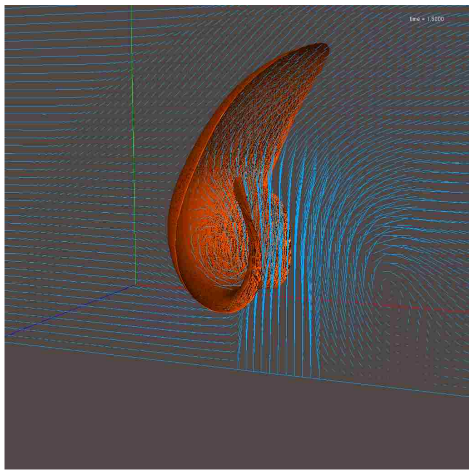
A visual inspection of the meshes in Figure 4.9 reveals that the vector angle criterion causes more refinement in curved regions of the vector field, and less refinement in relatively flatter regions of the vector field. This is noticeable first in Figures 4.9(a) and 4.9(c). The top of the surface is relatively sparse compared to the bottom part. In Figure 4.9(c), we see an orthoslice of the vector field and notice how the surface curves around the swirling velocity (vortex). In Figure 4.9(e), we observe a high degree of edge splitting at the bottom of the surface and relatively less



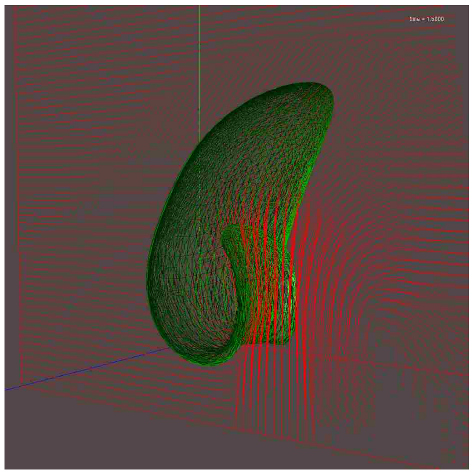
(a)



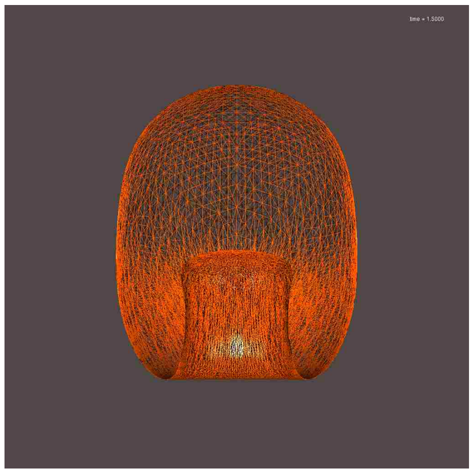
(b)



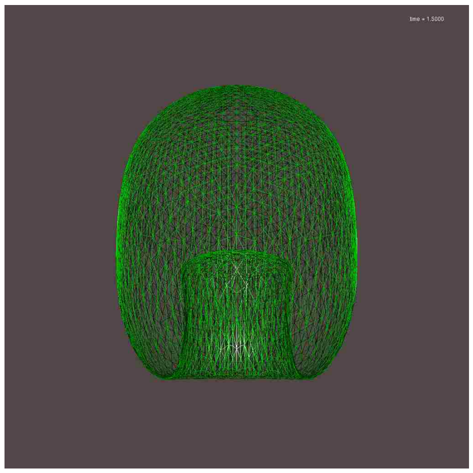
(c)



(d)



(e)



(f)

Figure 4.9: Comparison of a refined surface at t = 1.5 with the vector angle criterion (orange) and with the conventional edge length criterion (green).

edge splitting at the top half of the surface. In contrast, the edge-length-based refinement produces a relatively even mesh. Unlike the vector angle criterion, the edge length criterion splits a mesh homogeneously regardless of the surface curvature or the the curvature of the vector field.

To evaluate the accuracy of the surface, we computed the area of the time surface at t = 1.975 for various values of the θ_{max}. Then we compared those areas to the area of a control time surface derived from an initial seed surface with a high number of vertices and triangles (radius = 0.05, max edge threshold = 0.0003, 680477 vertices). We evolved this surface without refinement. The control time surface's area at t = 1.975 is 1.041118. We used this value to calculate the error. In Table 4.1, we show chosen θ_{max} values and corresponding calculated areas. We notice that as the number of vertices increases, due to smaller and smaller angle thresholds, the value of the area gets closer to the desired value. From the graph below (Figure 4.10), we see that after 47000 vertices ($\theta_{max} = 10°$), the area error does not change significantly.

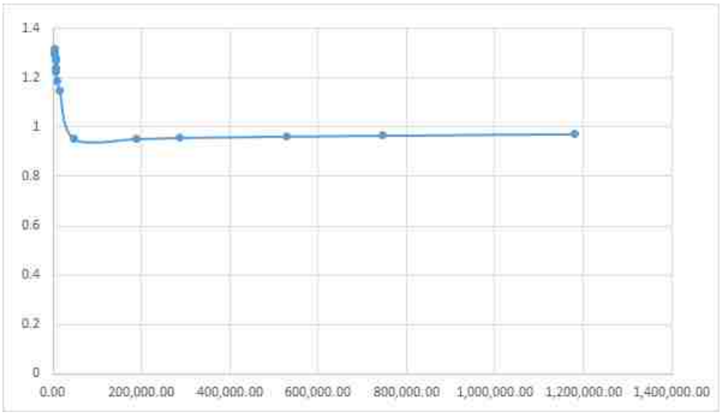


Figure 4.10: Surface area error vs. the number of vertices of various time surfaces at t = 1.975

4.4 Surface Refinement Algorithm

With the three basic surface refinement operations described in section 4.2, and the refinement criteria described in the previous section, we can refine any surface. All refinement operations take as input:

1. A triangular mesh M.

Table 4.1: Surface Area computation for various vector angle thresholds

Threshold θ_{max}	Vertices	Surface Area	Error
100°	5156	1.317839	26.58 %
90°	5196	1.303848	25.24 %
80°	5196	1.292669	24.16 %
70°	5482	1.28163	23.10 %
60°	5721	1.268887	21.88 %
50°	6135	1.240645	19.16 %
40°	6821	1.225894	17.75 %
30°	8438	1.187791	14.09 %
20°	14719	1.145927	10.07 %
10°	47843	0.95185	8.57 %
5°	189231	0.95185	8.57 %
4°	285555	0.956402	8.14 %
3°	528028	0.961877	7.61 %
2.5°	745243	0.965445	7.27 %
2°	1181010	0.972058	6.63 %

2. A list of modifiable edges. These lists result from whatever refinement criteria the user has decided to use.

The split and collapse require a reference to an edge interpolation scheme. This interpolator is also specified by the user.

The surface refinement algorithm requires as input a triangular mesh M, and returns a newly refined triangular mesh M_{new}. The surface refinement algorithm is described below:

Algorithm 4.9: Surface refinement algorithm

```
1. Set $M_{new} = M_{old}$ and return $M_{new}$ if there are no refinement
   criteria
2. Find splitable edges in $M_{old}$
3. While there exists splitable edges, do:
    • $M_{new} = Split(M_{old}, listOfModifiableEdges)$
    • find flippable edges in $M_{new}$
    • While there exists flippable edges, do:
        ◦ $M_{new} = Flip(M_{new}, listOfModifiableEdges)$
        ◦ find flippable edges in $M_{new}$
    • Find collapsible edges in $M_{new}$
    • **While** there exists collapsible edges, do:
        ◦ $M_{new} = Collapse(M_{new}, listOfModifiableEdges)$
        ◦ find collapsible edges in $M_{new}$
    • find splittable edges in $M_{new}$
    • set $M_{old} = M_{new}$
4. Return $M_{new}$
```

We demonstrate this algorithm with the example in Figure 4.11. In the first iteration, we split all 3 edges. Then, we flip only one flippable edge. No collapsible edges were found, so we obtain a new mesh that has 9 edges. In the second iteration, we split all 9 edges. Then, we flip 4 flippable edges. The resulting mesh has 34 edges.

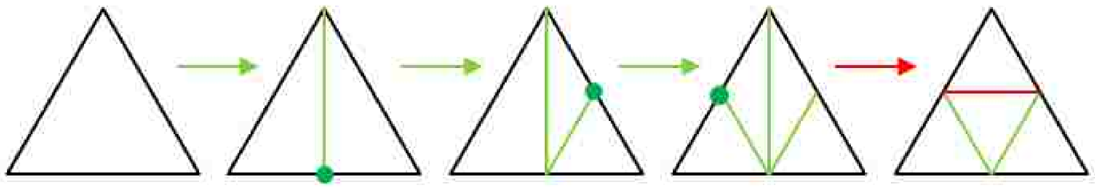
4.5 Post- vs. Pre- Refinement

In section 3.5, we presented the conventional way the time surface generation loop is implemented. We propose the following method to generate a time surface:

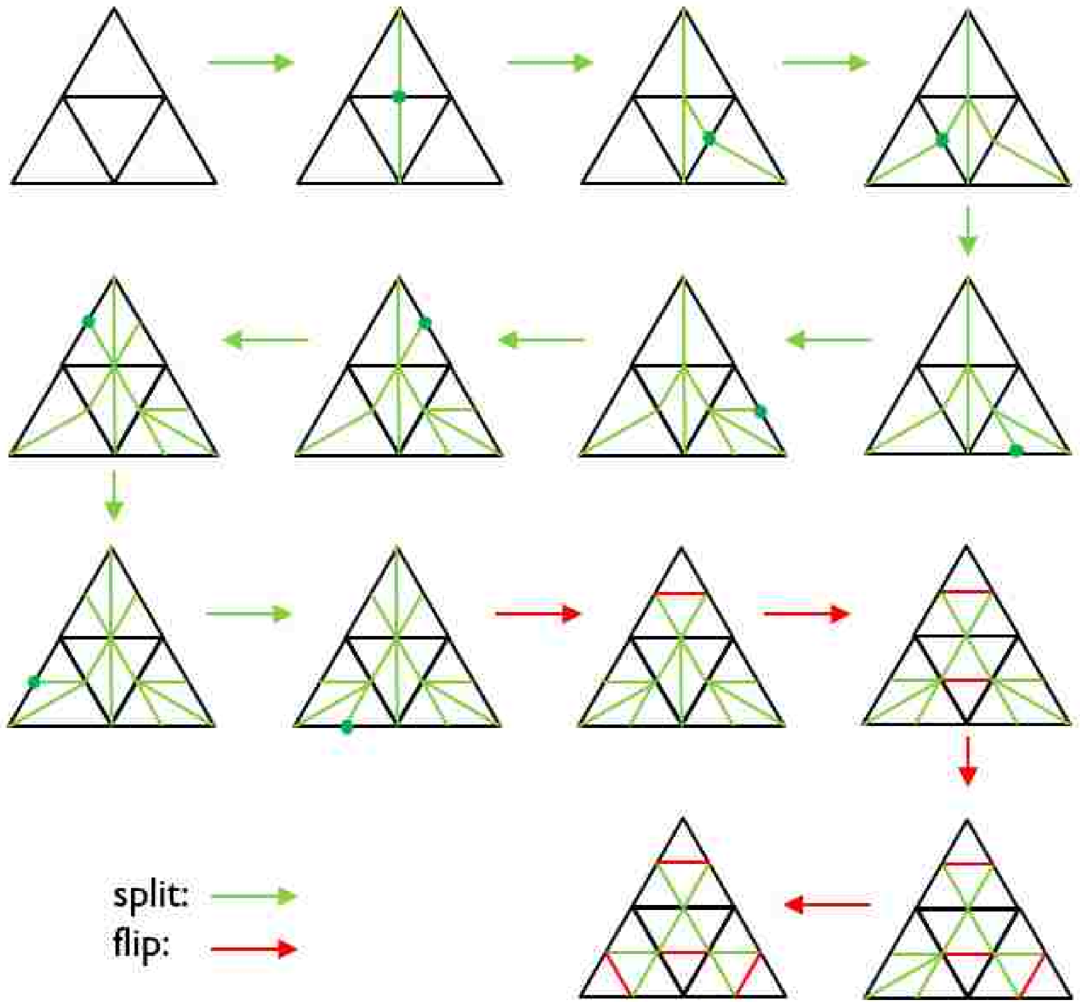
1. **Pre-refinement**: before the triangular mesh stretches, we refine it by adding, deleting, and updating certain edges of the surface. We will examine the local vector field at the edge to decide whether to perform a refinement operation or not (see vector angle criterion in subsection 4.3.4).

2. Compute the new locations of the points from the current time step to the next.

The first pre-refinement step will minimize error, because we split edges before they have the chance to elongate and introduce more roundup error. This is possible because when we examine the vector angle at an edge, we can predict if an edge needs splitting or collapse.



(a) 1^{st} refinement iteration



(b) 2^{nd} refinement iteration

Figure 4.11: Surface refinement algorithm example of a triangle.

4.5.1 Interpolation

In many mesh optimization works, a new vertex is typically inserted at the midpoint of an edge which has undergone a split or a collapse operation. Since the vector field (its magnitude and direction) influences where the points move, we can improve the midpoint interpolation by shifting the midpoint closer to the vertex with a higher velocity magnitude (Figure 4.12). Let $\vec{v}_1$ and $\vec{v}_2$ be the associated vectors at edge $e(p_1, p_2)$. Then, the new insertion point is:

$$p_3 = \frac{1}{(|\vec{v}_1| + |\vec{v}_2|)}(|\vec{v}_1|p_1 + |\vec{v}_2|p_2) \tag{4.6}$$

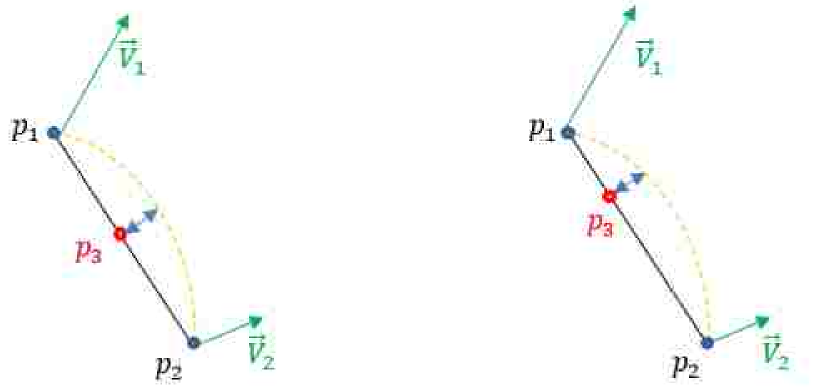


Figure 4.12: Midpoint interpolation vs. interpolation with vector magnitude.

4.5.2 Evaluation and Analysis

We tested both algorithms using an initial sphere and the following vector field:

$$\vec{V}((x, y, z), t) = \frac{1}{\sqrt{x^2 + y^2 + z^2}}(x, y, z) \tag{4.7}$$

In Figure 4.13, we plot a slice of the vector field. We use a coarse yellow unit sphere as initial seed surface (Figure 4.13). The magnitude of the vector field is 1 throughout the domain, so we know at each time step the exact radius of the sphere. Thus, we can evaluate the surface and volume analytically. It is obvious from Figure 4.14 that the green time surface (vector-angle based refinement) matches a sphere better than the purple time surface (edge-length based refinement).

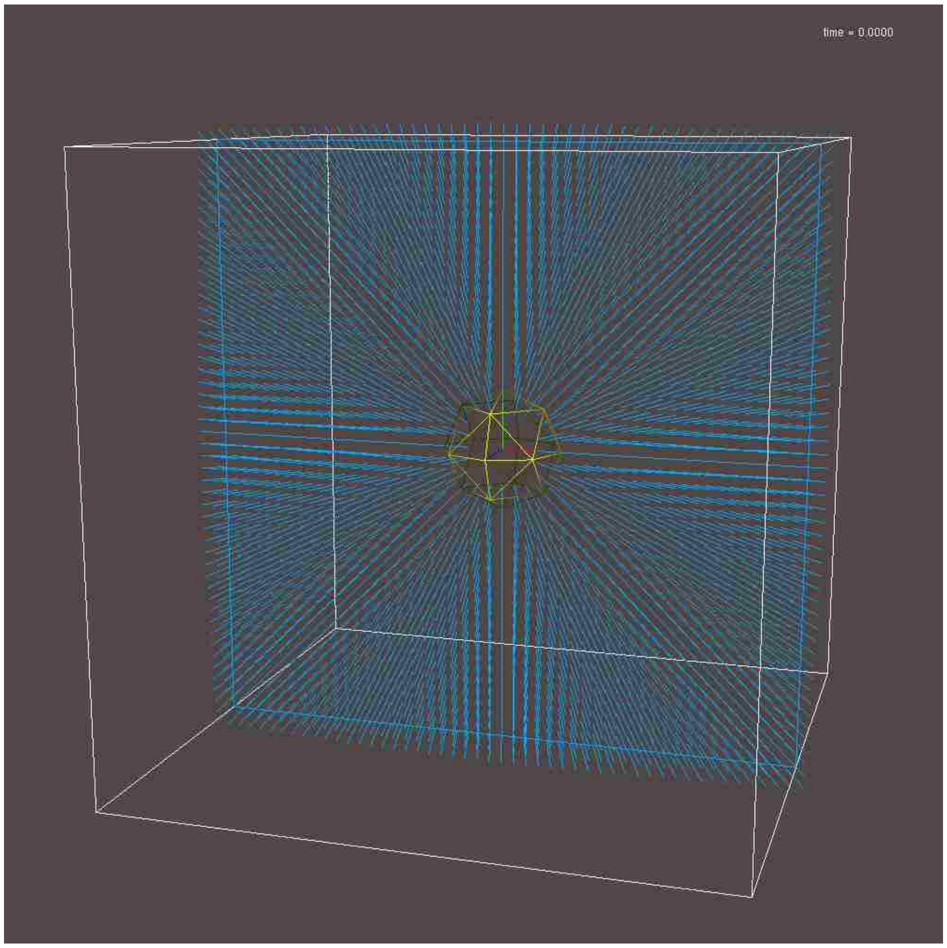


Figure 4.13: Orthoslice of the vector field described by equation 4.7 (blue) and coarse initial sphere (yellow)

We computed the areas and volumes for the green and purple time surfaces in Tables 4.2 and 4.3, receptively. Comparing the errors in these two tables, we notice that the error is smaller when we use vector-angle criterion and pre-refinement. Furthermore, we notice that the vector-angle criterion uses fewer vertices and triangles to model the surface. At timestep, the number of vertices and triangles in the purple time surface is 24 times higher than in the green time surface, yet the error is larger in the purple time surface.

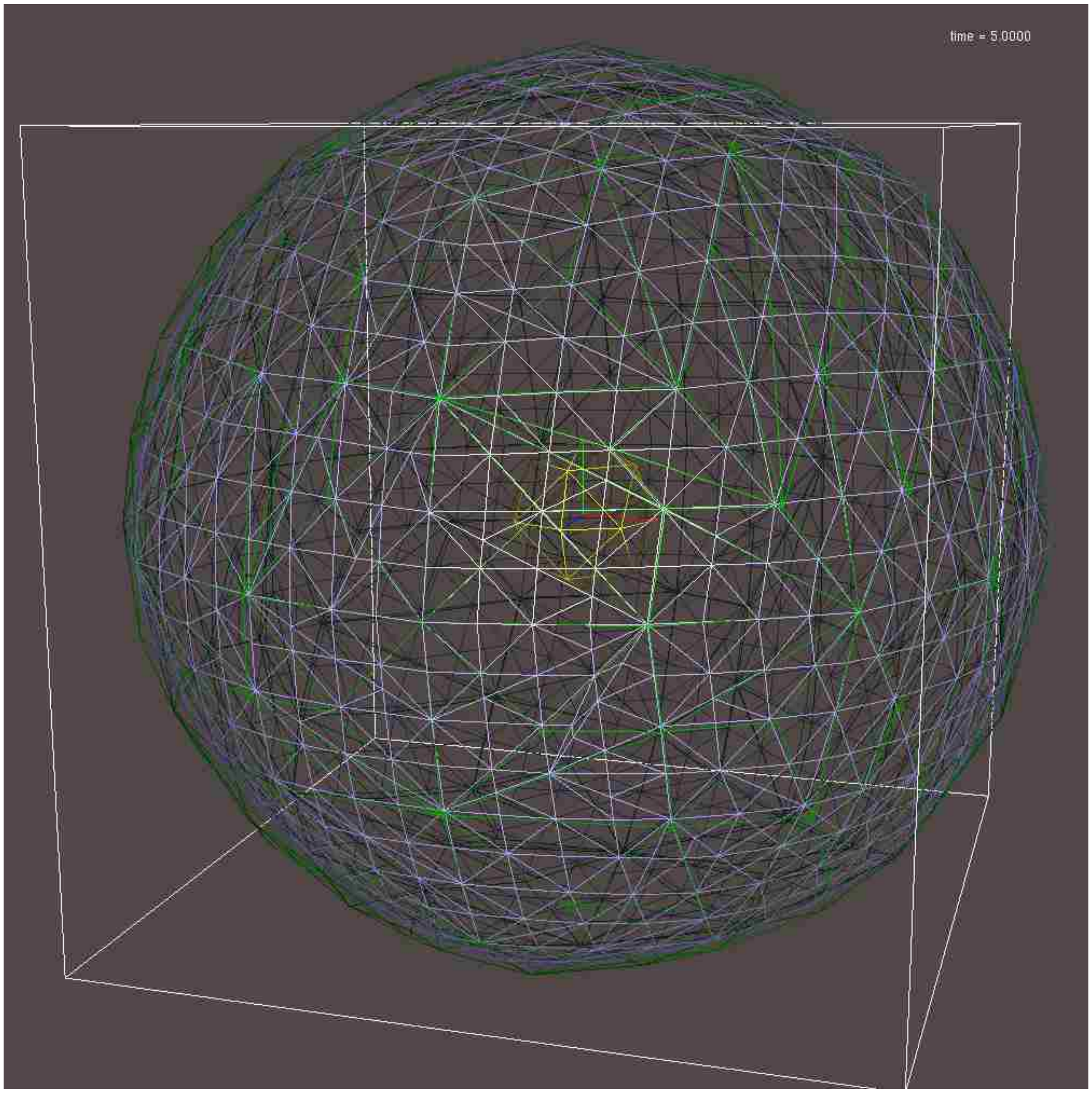


Figure 4.14: Edge-length based refinement (purple, max edge length = 1) vs. vector-angle based refinement (green, max angle = 30^o).

Table 4.2: Volumes and areas of a time surface generated using pre-refinement (vector angle = 30)

Timestep	Vertices	Edges	Faces	Area	Error	Volume	Error
0	207	615	410	10.42	17.10%	2.94	29.75%
1	207	615	410	44.94	10.60%	27.64	17.52%
2	207	615	410	104.25	7.83%	98.24	13.14%
3	207	615	410	188.35	6.32%	238.90	10.89%
4	207	615	410	297.25	5.38%	473.76	9.52%
5	207	615	410	430.95	4.74%	826.99	8.60%
6	207	615	410	589.45	4.27%	1322.75	7.94%
7	207	615	410	772.75	3.92%	1985.17	7.44%
8	207	615	410	980.85	3.64%	2838.42	7.05%
9	207	615	410	1213.75	3.41%	3906.65	6.74%

Table 4.3: Volumes and areas of a time surface generated using post-refinement (max length = 1)

Timestep	Vertices	Edges	Faces	Area	Error	Volume	Error
0	98	288	192	10.42	17.10%	2.94	29.75%
1	381	1137	758	41.67	17.10%	23.54	29.75%
2	671	2007	1338	97.75	13.57%	87.41	22.71%
3	1117	3345	2230	179.08	10.93%	220.22	17.85%
4	1393	4173	2782	285.41	9.15%	446.71	14.69%
5	1577	4725	3150	416.86	7.85%	792.06	12.46%
6	2451	7347	4898	573.46	6.87%	1281.30	10.82%
7	3123	9363	6242	755.13	6.11%	1939.41	9.57%
8	4412	13230	8820	961.95	5.49%	2791.87	8.57%
9	5032	15090	10060	1193.89	4.99%	3863.66	7.76%

Chapter 5
Software Design and Implementation

In the process of implementing the algorithms described throughout this thesis, we used two software. When we started, we used VISH because it supported curvilinear interpolation of the velocity field. In this software, we implemented the first time surfaces prototype. The second module we implemented was the curvature and torsion module which converted a subset of a curvilinear grid into a uniform Cartisian grid, then calculated the curvature and torsion fields on the uniform grid.

Due to the difficulty of debugging code in VISH and the lack of documentation, we developed an alternate software called *Dessine-Moi*. In this software, we implemented the latest version of the time surfaces module and the module that computes Ottino's mixing indicators. For further information about VISH, refer to [3]. In this chapter, we will describe our software *Dessine-Moi*. We will begin by listing the middleware we used. Then, we will describe the user interface (UI). For each UI component, we will describe how it was implemented.

5.1 Middleware

The first middleware we use is the Qt library. Qt is a powerful C++ cross-platform framework, similar to the .NET and JAVA frameworks. In addition to being used for developing GUI-driven applications, it has extensive support for non-GUI applications. These include SQL database access, networking capabilities, XML parsing, and thread management. Qt is available for all major platforms (Windows, Apple, Linux, and others) and support many compilers (such as gcc, and visual studio). Qt and its supporting tools are available under an open source (LGPL v2.1) version and a commercial version.

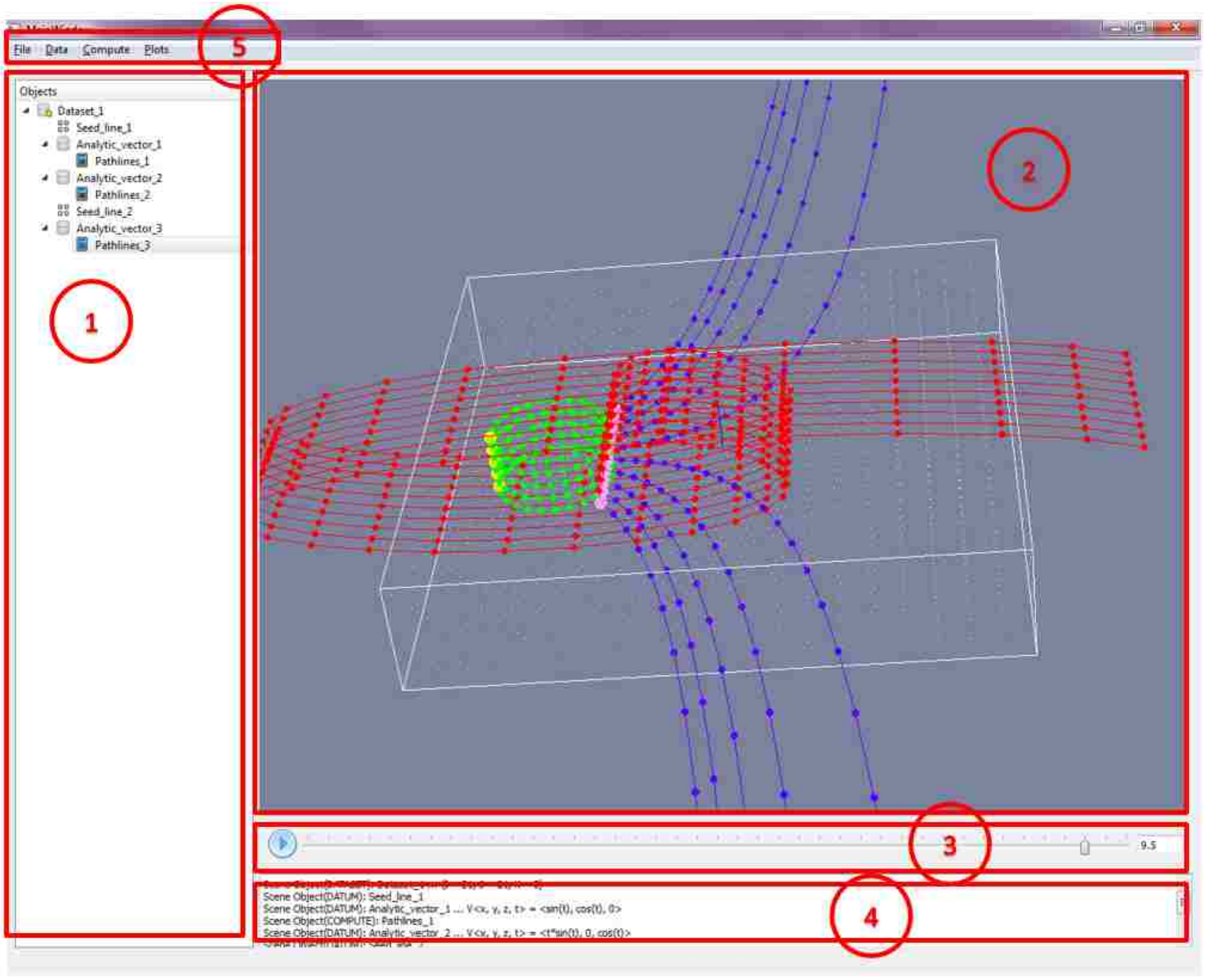
Additionally, we use the following libraries:

1. *HDF5*. HDF5 is a data model, library, and file format for storing and managing data in binary.

2. *OpenGL Extension Wrangler Library* (GLEW). GLEW is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions.

3. *freeglut* is an open source alternative to the OpenGL Utility Toolkit (GLUT) library. Freeglut allows the user to create and manage windows containing OpenGL contexts on a wide range of platforms and also read the mouse, keyboard and joystick functions.

5.2 User Interface



```
Scene Object(DATUM): Seed_line_1
Scene Object(DATUM): Analytic_vector_1 ... V<x, y, z, t> = <sin(t), cos(t), 0>
Scene Object(COMPUTE): Pathlines_1
Scene Object(DATUM): Analytic_vector_2 ... V<x, y, z, t> = <t*sin(t), 0, cos(t)>
```

Figure 5.1: Overview of *Dessine-Moi*.

The main parts of *Dessine-Moi* are:

1. *The scene object viewer*: illustrates the hierarchical relationship between the different modules in the scene. If the user wants to modify a module, they simply right click and select the appropriate option in the context menu.

2. *The viewer*: this is where the rendering of scene objects is done using OpenGL. Users can rotate, move, and zoom in the scene.

3. *The animation control slider*: this is where we explore a time-dependent dataset. If the dataset is time-independent, this control is disabled.

4. *The info window*: information about the created modules is displayed here.

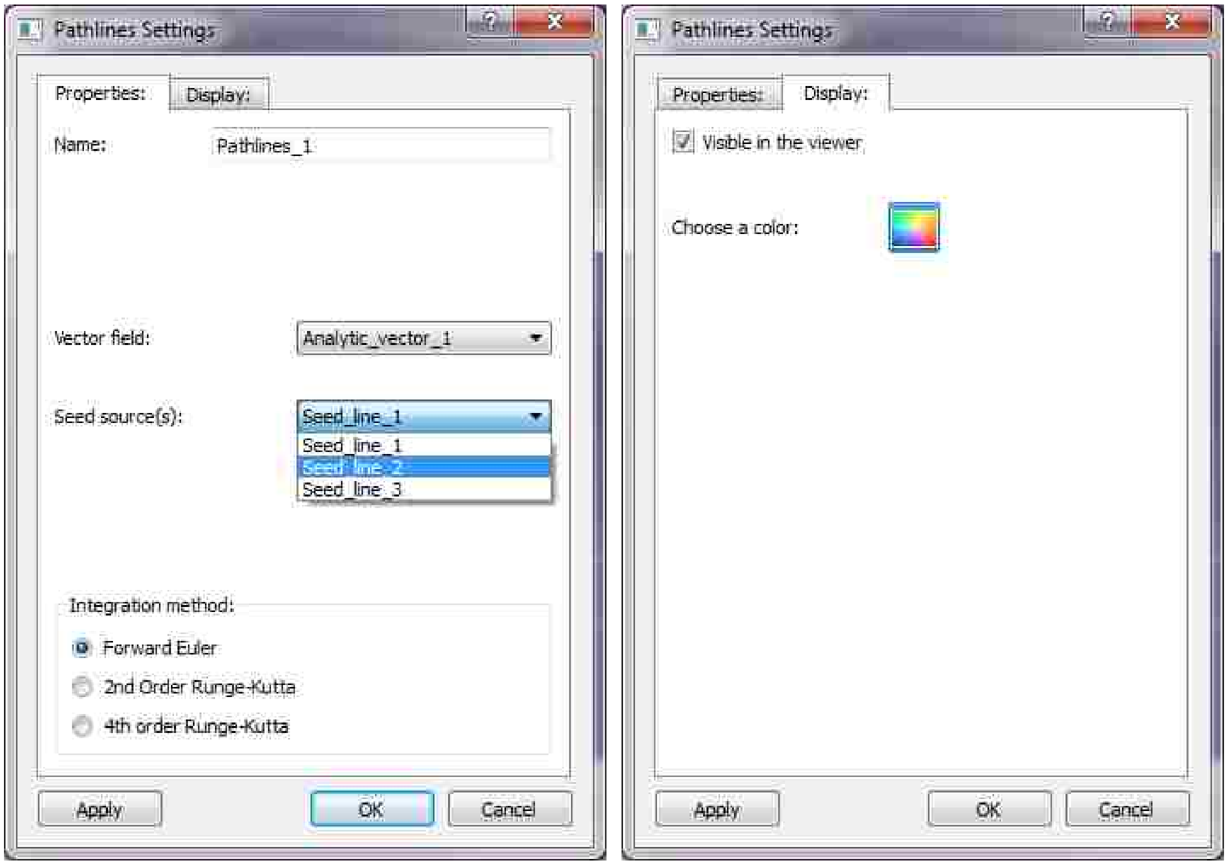
5. *The menu*: the different modules available in this application are invoked by choosing them from this menu.

These five parts are grouped together into one class called `MainEnvironment`. `MainEnvironment` stores pointers to these five objects. When a module is started, an instance of `MainEnvironment` is passed to the module so that the module may interact with the main application.

5.3 Modules in *Dessine-Moi*

Any given module in *Dessine-Moi* is derived from a base class called `SceneObject`. The `SceneObject` class has two important virtual methods: `Draw()` and `Compute()`. In *Dessine-Moi*, any module can be rendered. If a developer writes a module that computes a quantity, they can leave out the `Draw()` method. Likewise for the `Compute()` method, if a developer writes a module meant only to be used for rendering purposes, they do not have to implement the `Compute()` method. A good example of this is the `AxesSceneObj` class which is responsible for drawing the *xyz* axes. Additionally, any instance of a `SceneObject` has the following members: `name`, `visible`, and `currentTime`.

In addition to implementing a derived class from `SceneObject`, the user implements a Qt dialog window that serves as the UI that allows the user to interact with the derived `SceneObject` class. We use the Qt designer which is integrated in the Qt Creator IDE. This is very convenient as it allows us to drag and drop the UI elements we need directly onto the Qt form. This saves development time. As a convention, we always include two tabs in a module's dialog. The first one is the *Properties* tab, this is where the user introduces computational parameters to the module. The second tab is the *Display* tab. In this tab we control the visual appearance of the scene object. In Figure 5.2, we illustrate both of these tabs.



Pathlines Settings

Properties: | Display:

Name: Pathlines_1

Vector field: Analytic_vector_1

Seed source(s): Seed_line_1
- Seed_line_1
- Seed_line_2
- Seed_line_3

Integration method:
- [✓ Forward Euler]
- 2nd Order Runge-Kutta
- 4th order Runge-Kutta

Apply | OK | Cancel

Pathlines Settings

Properties: | Display:

[✓] Visible in the viewer

Choose a color:

Apply | OK | Cancel

(a) Properies tab

(b) Display tab

Figure 5.2: Pathlines Module UI.

The *Apply* button in Figure 5.2 refreshes the viewer and the *OK* triggers the `Compute()` method. Note that these conventions are not mandatory, but we try to follow them so that the UI remains consistent for the user. As a programming convention, we suffix a scene object class with "SceneObj", and we suffix a dialog class of module with "UI".

Finally, a corresponding menu item is added to one of the menu lists (Figure 5.1). Depending on the functionality of the module, we add it to the *Data*, *Compute*, or *Plot* menu lists. This is how the user is able to locate a module of interest and invoke it.

5.4 Scene Object Viewer

The scene object viewer allows the user to keep track of the scene objects that the user created and to modify them if necessary. This scene viewer UI is a `QTreeView` widget. Associated with the UI is a data source. We use Qt's model/view architecture to manipulate the content (data source) of the `QTreeView` widget. We wrote a `SceneModel` class which is derived from `QAbstractItemModel` class. This class dictates how to add and remove items from the `QTreeView` widget. It contains a pointer to the root node of type `SceneNode`. The `SceneNode` class stores a pointer to the scene object associated with it, a pointer to its parent scene node, and a pointer to its children scene nodes. In the future, we will write a function that will traverse the tree stored in the scene model and save the scene into a XML file (as a script). That way, when the user opens this XML file, *Dessine-Moi* will re-create the scene (by parsing the XML) that the user was working on in a previous session.

5.5 Viewer

This is where OpenGL commands are invoked to render a scene. We derived this widget from `QGLWidget`. `QGLWidget` provides three convenient virtual functions that we reimplemented:

- `initializeGL()`: here we set up the OpenGL rendering and the background color.
- `resizeGL()`: here we set up the OpenGL viewport and projection.
- `paintGL()`: here we render the scene objects. This method is called whenever the widget needs to be updated.

The viewer has a queue of scene objects. When the user clicks on a module's *OK* button for the first time, the module adds its scene object to the end of the queue. When `paintGL()` is triggered, this method loops through all its scene objects and invokes their `Draw()` methods. If the queue holds a dataset, a spherical seed surface, and the axes, then `paintGL()` will call each of these scene object `Draw()` methods. If a scene object's `visible` flag is switched off, then `paintGL()` will not render that particular scene object.

5.6 Animation Control Slider

This control is responsible for updating all scene objects with the current time. When a time-dependent dataset is created, this control stores all the time instances available in the dataset. When a new scene object is added with its unique time series, the time steps are merged. We use a hash table (C++ map) to easily add, delete, and merge time steps. The animation control slider, or time slider is connected to the OpenGL viewer via Qt's signal and slot mechanism. The animation control sends a signal to the OpenGL viewer when the user moves the slider. The signal is the index at which the slider is at. The OpenGL viewer has a slot that catches the signal. We wrote a method that converts the slider's index into the actual time step. In this method, we loop through all the scene objects and update their time with the current time step.

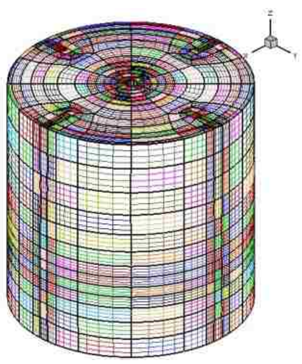
Chapter 6
Applications

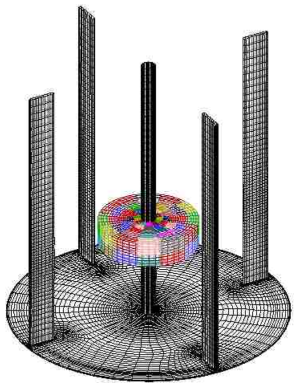
In this thesis, we generated and rendered time surfaces, computed fluid mixing indicators, and visualized these indicators. We applied and tested our techniques on several datasets to gain insight about the fluid flow phenomena being simulated.

6.1 Stirred Tank Simulation

Stirred tank reactors (STR) are mixing devices commonly used in chemical and processing industries. Improvements in the design of stirred tanks can translate into several billion dollars of annual profit. Therefore, to improve stirred tanks, detailed understanding of flow and mixing behavior inside such tanks is crucial. A dataset of such a flow field was obtained from a large eddy simulation (LES) of a STR (Figure 6.1(b)). The immersed boundary method (IBM) was used to model rotating impeller blade (Figure 6.1(c)). The simulation was performed on 200 processors (64 bit 2.33 GHz Xeon quadcore) where each timestep is calculated in approximately 36 seconds [5].

(a) Stirred tank


(b) Grid


(c) Baffels geomerty

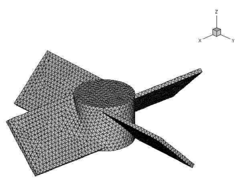

(d) Impeller gerometry

Figure 6.1: Stirred tank reactor geometry.

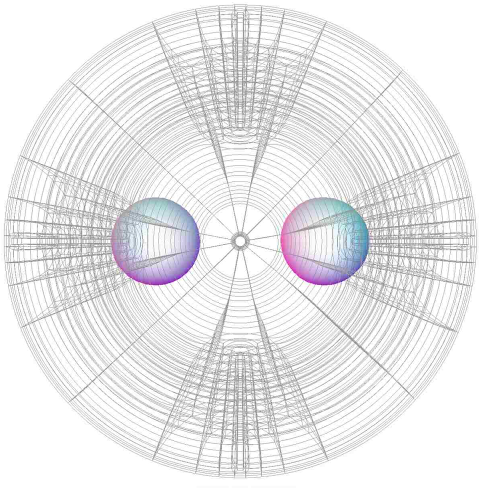
The stirred tank reactor modeled is cylindrical in shape and has a hemispherical base (Figure 6.1(a)). The tank has four baffles that break the flow and create more turbulence (Figure 6.1(c)). The impeller consists of rectangular blades at a 45° pitch mounted on a shaft passing through the tank's central z-axis (Figure 6.1(d)).

The grid is distributed over 2088 curvilinear blocks 6.1(b)) and comprises 3.1 million cells in total [5]. The velocity and pressure fields are given at the center of each cell. The simulation is computed over a total of 5700 timesteps.

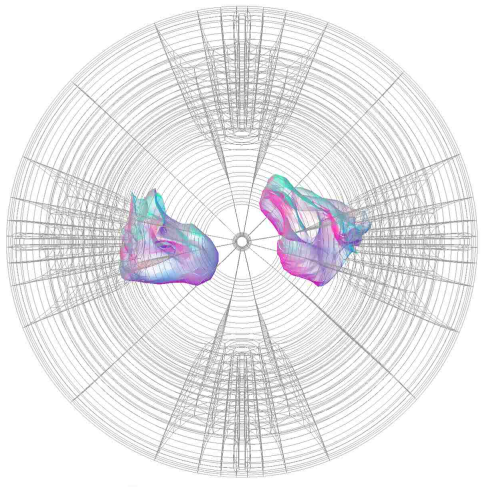
6.1.1 Evolving Time Surfaces in the Stirred Tank

Our initial surface refinement algorithm (see [5]) was implemented in VISH. While VISH supported interpolation of a vector field in a curvilinear grid, the interpolator was slow. Thus, for the time surfaces module, VISH supported only the Euler integration method to minimize the calls to the interpolator. For surface refinement, VISH supported a rudimentary triangular mesh data structure consisting only of an array of vertices and an array of triangles. Since we did not have any triangle adjacency information, we refined a triangle by simply splitting all its edges in half. The original triangle was replaced by four smaller triangles. In [5], we implemented the edge length and the triangle area refinement criteria. Depending on the refinement criterion, the adjacent triangles may or may have not been refined. If the adjacent triangles have been refined, we inserted redundant vertices because of the lack of neighborhood information.

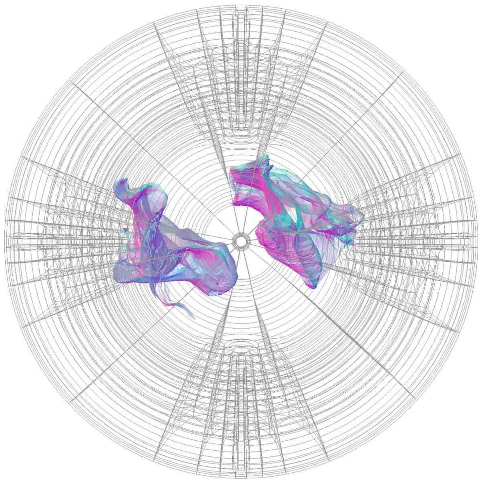
In our initial time surfaces prototype [5], we used the stirred tank dataset described in the previous section. We were interested in visualizing how injected matter in the stirred tank flow field would disperse and mix in various regions of the stirred tank. We emulated injection of two blobs of matter by placing two spherical seed points at the top of the stirred tank (Figure 6.2(a)). Due to memory limitations in VISH, we generated two time surfaces of the corresponding two initial spheres for only the first 150 timesteps of the stirred tank dataset. A few of these timesteps (0, 50, 100, 125, and 150) are shown in figure 6.2. As time elapses, each sphere begins to expand and sink towards the impeller of the stirred tank (see Figure 6.1(c)). At t_{150}, the time surfaces are about to mix.



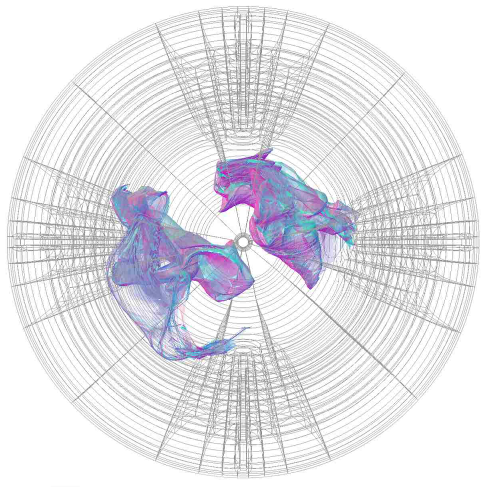
(a) t_0



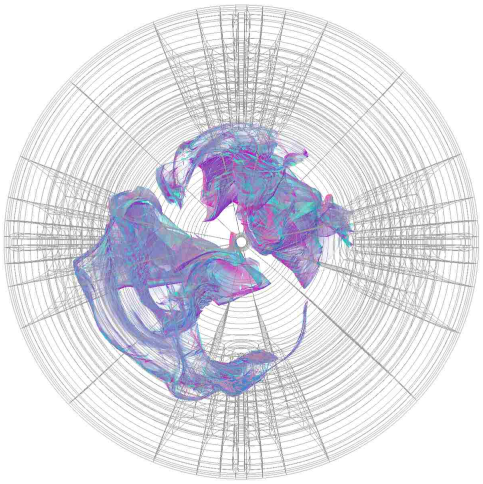
(b) t_{50}



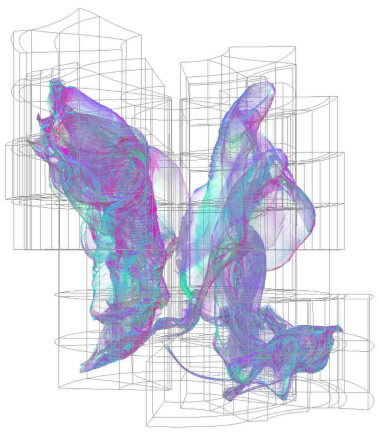
(c) t_{100}



(d) t_{125}



(e) t_{150}



(f) t_{150} (side view)

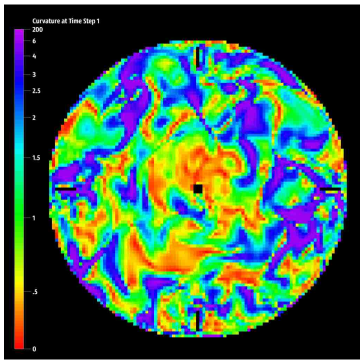
Figure 6.2: Sample top view snapshots of two time surfaces evolving in a stirred tank.

6.1.2 Curvature and Torsion Fields of the Flow Field

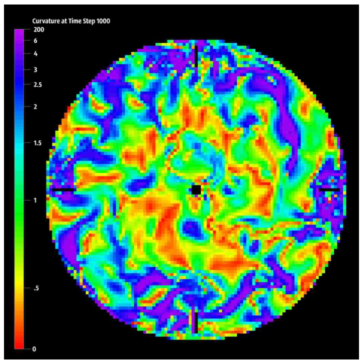
We used the multi scale color coding technique described in section 3.7 to color code the curvature and torsion fields in a specified orthoslice of the stirred tank's uniform Cartesian grid. The orthoslice we used is parallel to the *xy* plane and is located around the middle of the stirred tank. The goal is to visualize the variations of the curvature to see the regions of high and low straining of the fluid elements, and to visualize the variations of the torsion to see where fluid elements get twisted the most. We rendered 12 multi scale color coded orthoslices, including 6 for the curvature field and 6 for the torsion field at timesteps 1, 1000, 2000, 3000, 4000, and 5000, in order to observe the scalar fields at different timesteps (Figure 6.3).

Figures 6.3(a) through 6.3(f) show the multi scale color coded curvature images in the orthoslice at time steps 1, 1000, 2000, 3000, 4000 and 5000. The multi scale used in these 6 images is similar to the one described in section 3.7 except that the upper end point of the curvature range in the last division is 200 instead of 55 (i.e., the curvature values are capped at 200 instead of 55) and the percent ranges for the 9 divisions are slightly different at each of the 6 time steps. Thus, Figure 6.3(a) is the same as Figure 3.8(b) except that the curvature values in Figure 3.8(a). are capped at 200 instead of 55 in order to be consistent with Figures 6.3(b) through 6.3(f). In all of these images, the multi scale color coding enables one to clearly visualize the small, medium and large curvature values as the red and orange, yellow and green, and blue and violet areas, respectively. It can be observed in these time series images that in general the regions of high and low curvature do not change spatial locations significantly with time. This type of visualization and analysis is not possible with uniform color coding.

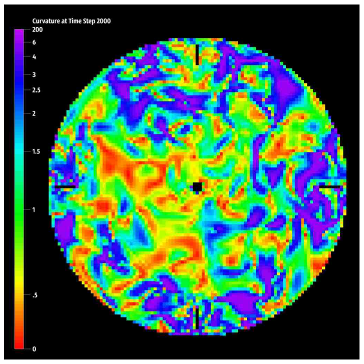
The greater the values of curvature, the greater the mixing of the fluids in that region. There are many regions (blue and violet) of high curvature indicating high mixing. These regions are typically located in the near-baffle regions close to the stirred tank wall. Similarly, there are areas (red and orange) of low curvature which indicate low mixing and are typically located in the interior of the tank. These types of observations are not possible in the uniformly color coded images as they are visually incapable of providing a definitive analysis of the curvature field.



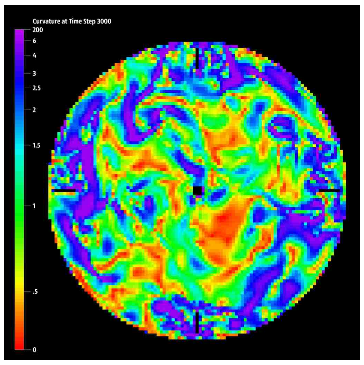
(a) Curvature at t_1



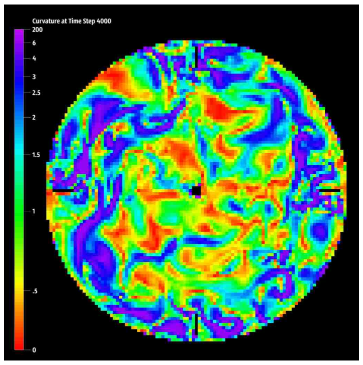
(b) Curvature at t_{1000}



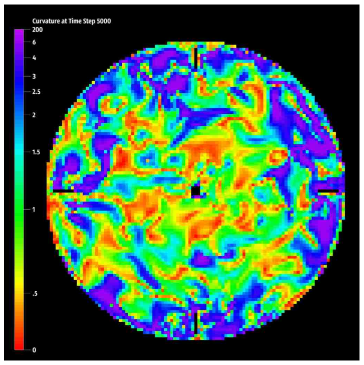
(c) Curvature at t_{2000}



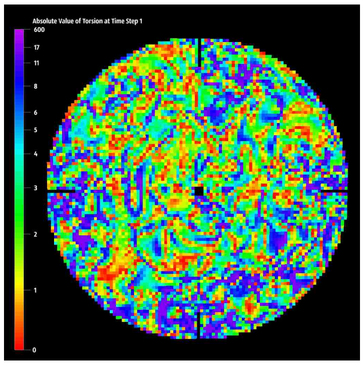
(d) Curvature at t_{3000}



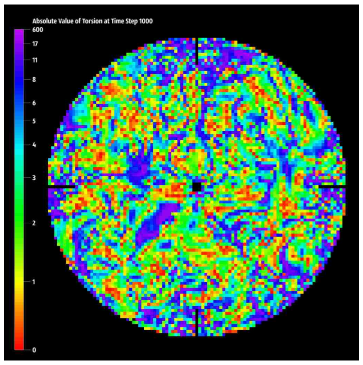
(e) Curvature at t_{4000}



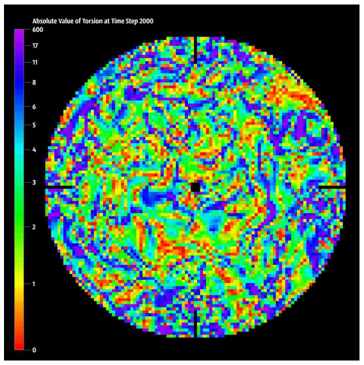
(f) Curvature at t_{5000}



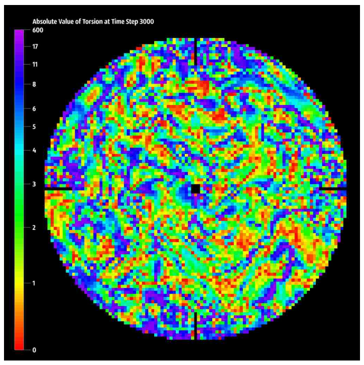
(g) Torsion at t_1



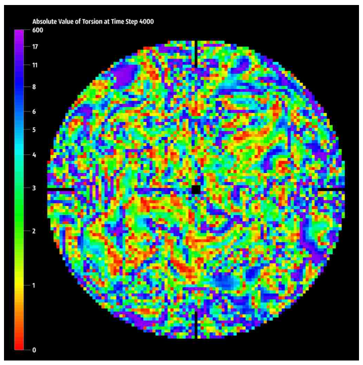
(h) Torsion at t_{1000}



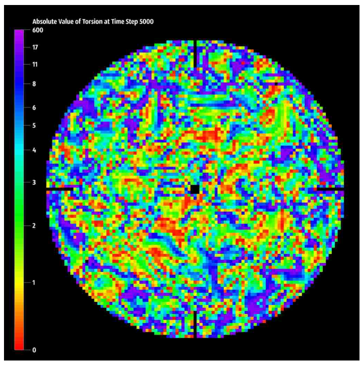
(i) Torsion at t_{2000}



(j) Torsion at t_{3000}



(k) Torsion at t_{4000}



(l) Torsion at t_{5000}

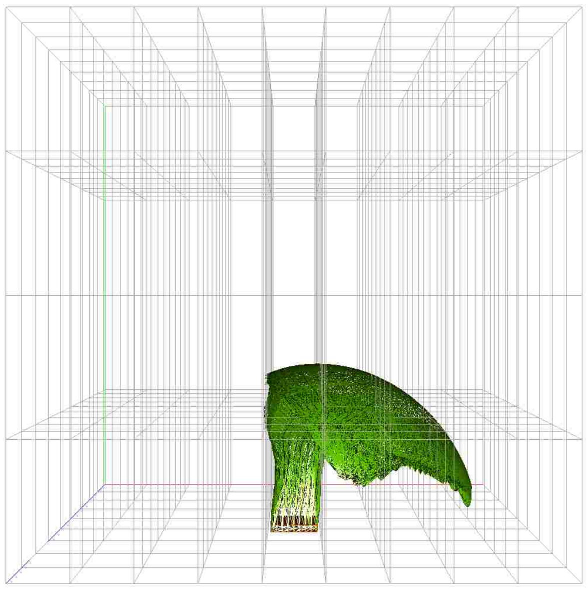
Figure 6.3: Sample top view snapshots of two time surfaces evolving in a stirred tank.

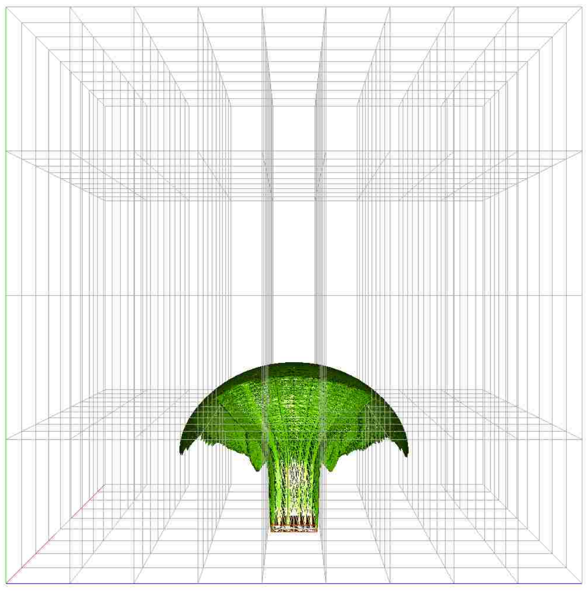
Figures 6.3(g) through 6.3(l) show the multi scale color coded torsion images in the orthoslice at time steps 1, 1000, 2000, 3000, 4000 and 5000. As mentioned in section 2.3, the torsion values can be positive or negative, but we are visualizing only the absolute value of the torsion, neglecting whether the twist is clockwise or counterclockwise since the degree of fluid mixing depends primarily on the magnitude of the torsion rather than its direction. The torsion distributions reflect the angular twist of the fluid elements, and regions of high torsion are associated with fluid elements that are significantly distorted in the angular plane and hence likely to be well mixed. As in the case of curvature, in general the outer regions of the stirred tank near the baffles produce areas of high torsion and hence high mixing, while low and medium torsion areas, which indicate low and medium mixing, are primarily located in the interior of the tank.

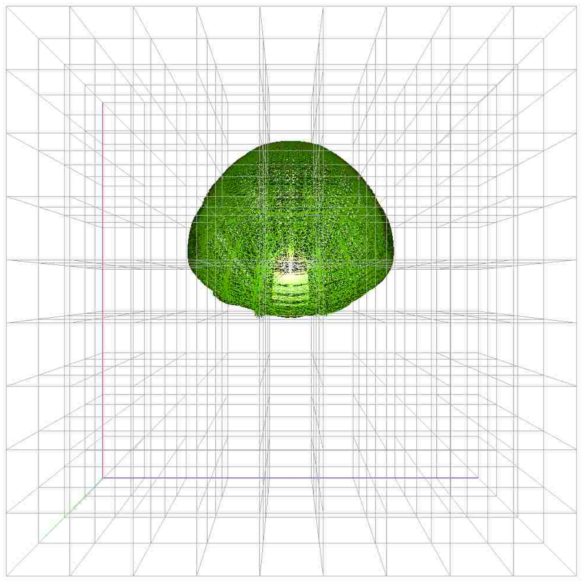
6.2 Oil Spill Simulation

This simulation attempts to model the oil spill plume that gushed out from a broken wellhead in the sea floor of the Gulf of Mexico in April 2010. To better understand the impact of this disaster, the LSU CFD team simulated the fate of the millions of barrels of petroleum lost and how it dispersed in the Gulf Coast under realistic sea and atmospheric wind conditions. We tested our algorithm on a preliminary implementation of this simulation. Using time surfaces, we want to visualize the surface of the plume, watch it evolve, and compute its volume at each timestep of the simulation.

The sample dataset we used contains 20 timesteps. The grid is rectilinear and is subdivided into 324 blocks. The size of the computational domain is the unit cube which corresponds in the physical world to a region of $4.8m \times 4.8m \times 4.8m$ surrounding the well head. The velocity of the oil is $0.53m/s$. The ocean current (cross flow velocity) is $0.08m/s$. All the blocks are of dimension $8 \times 15 \times 8$. Each timestep contains the following variables: $\vec{v}(u,v,w)$, *mut*, pressure P, *rho*, and *smn*. We converted the ASCII data to the HDF5 binary format. This allowed us to minimize storage size and improve I/O. In Figure 6.4, we show the grid geometry and the shape of the oil plume at timestep 10 (t = 1). We used the vector angle criterion and set the split angle to $2°$, $l_{max} = 0.02$, and $\Delta t = 0.025$.


(a) side view


(b) rear view


(c) top view

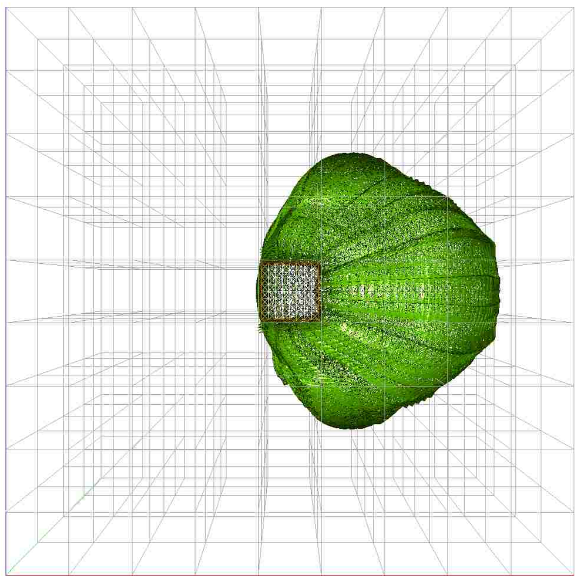
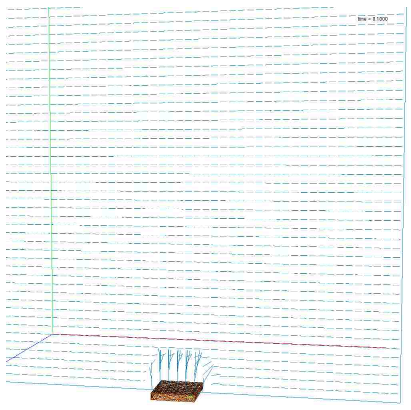
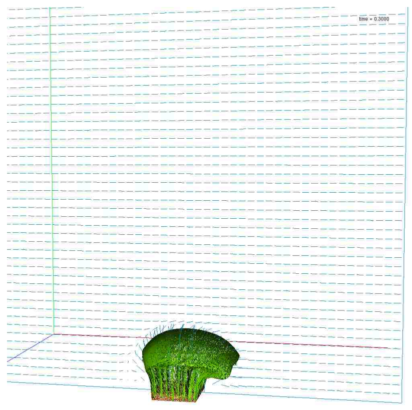
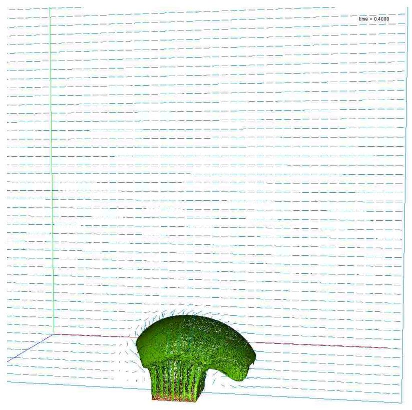
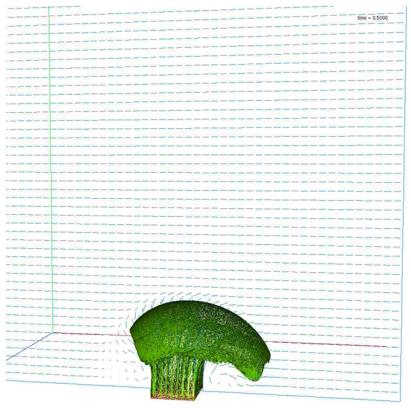

(d) bottom view

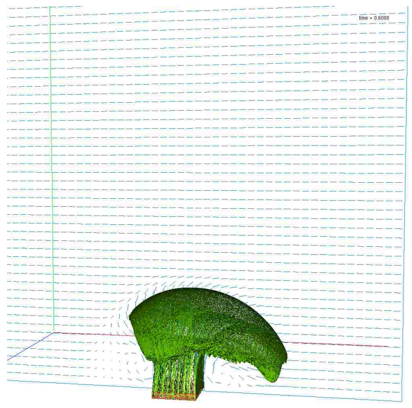
Figure 6.4: Simulation grid and corresponding oil plume at $t = 1$

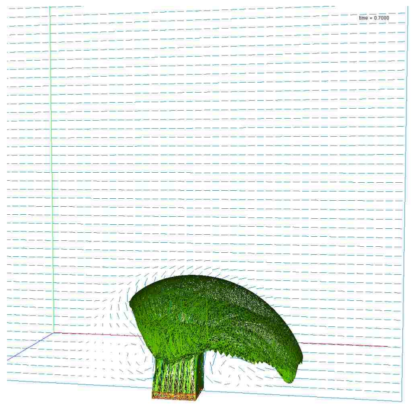

(a) $t_0 = 0.1$

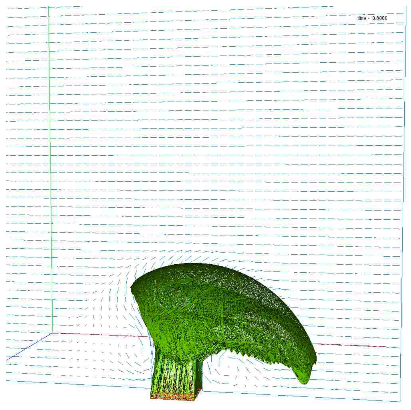

(b) $t_2 = 0.3$

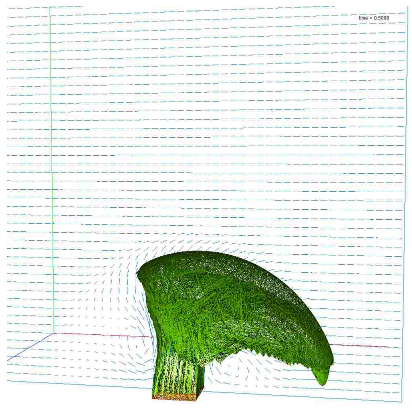

(c) $t_3 = 0.4$


(d) $t_4 = 0.5$


(e) $t_5 = 0.6$


(f) $t_6 = 0.7$


(g) $t_7 = 0.8$


(h) $t_8 = 0.9$

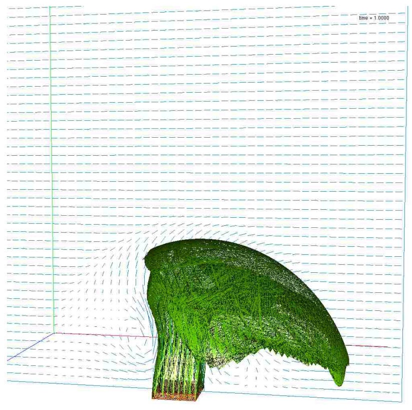

(i) $t_9 = 1.0$

Figure 6.5: Oil spill time surface during the first 10 timesteps

As an example, we generated a time surface for the first 10 timesteps of the dataset, as shown in Figure 6.5. We set the vector angle criterion to split if the the angle exceeds $\theta_{max} = 3°$ and $l_{max} = 0.02$. For the integration, we used a time increment $\Delta t = 0.025$. In the simulation provided, the oil gushes out of a square located at the bottom center of the computational domain. As initial seed surface, we used a $0.1 \times 0.1 \times 0.015$ cuboid mesh with maximum edge length $l_{max} = 0.015$. This cuboid surface is big enough to cover the oil leak (shown in orange in Figure 6.5(a)). We protruded the bottom of the cuboid outside the grid, where the velocity is zero, so that the bottom surfaces of the time surface remain fixed. At $t = 0.4$, the surface begins to shape up like a mushroom (Figure 6.5(c)). As time progresses, the top of the surface shifts in the direction of the ocean current (Figure 6.5(d) and onward).

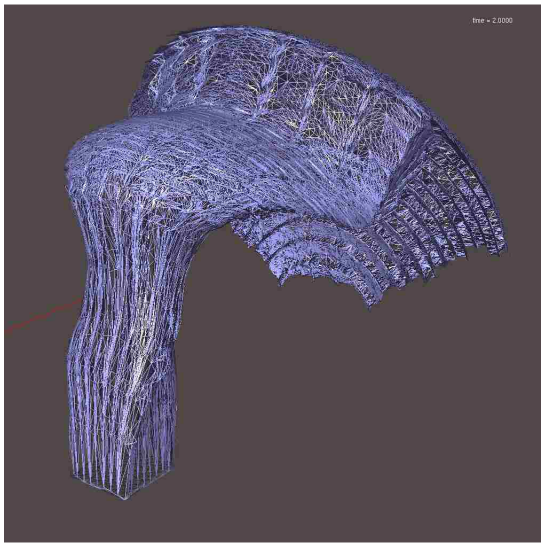
6.2.1 Surface Refinement Criteria Comparison

In this section we compare the vector angle criterion with the edge length criterion. We generate two time surfaces for the first 20 timesteps of the dataset. We used a time increment $\Delta t = 0.05$ for the integration which is equal to half the time increment of the dataset $(\Delta t_{Dataset} = 0.01)$. For the edge length method, we used post refinement and we set the maximum edge length to 0.02. For the vector angle, we used pre-refinement and set the the maximum angle to $2°$.

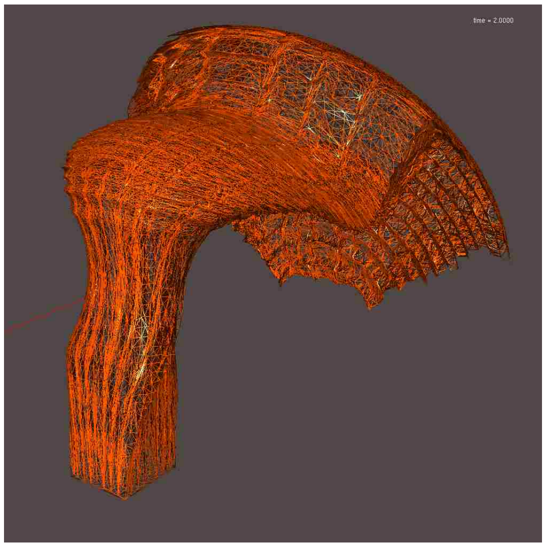
From Figure 6.6, we notice that the surface is less refined at the bottom of the well where the vector field is less curved, and the surface is more refined at the top of the plume where the vector field is more turbulent. For the particular vector angle criterion we used, we notice that the purple time surface generated has more vertices and triangles than (Table 6.1) the orange time surface (Table 6.2). In both cases, the number of vertices and triangles is a function of the refinement threshold be it the edge length criterion or the vector angle criterion. For maximum angle equal to $3°$, the number of vertices and triangles would be less. For $\theta_{max} = 3°$ and at $t = 1.95$, the number of vertices is 531735 and the number of triangles is 1063466.

Another observation worth noting is that the vector angle criterion is more time consuming to evaluate than the edge length criterion. This is because we need to interpolate the two vectors at any given edge. This is further confirmed in column two of Table 6.1. In Table 6.2, we notice that refinement time fluctuates between 40% to 90% depending on whether the surface needs intense refinement or not.

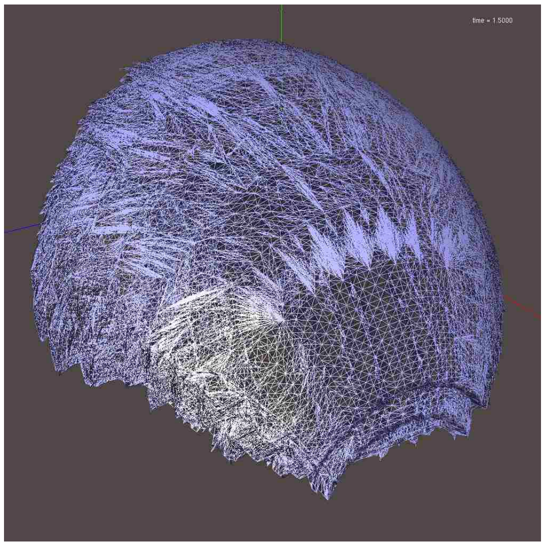
We notice that refining with the vector angle criterion is more time consuming (refer to table 6.3). In the future, we will make the vector-angle criterion multi-threaded, or possibly map this algorithm on the GPU. Our array-based data structure makes this algorithm a good candidate for a GPU implementation, as well.



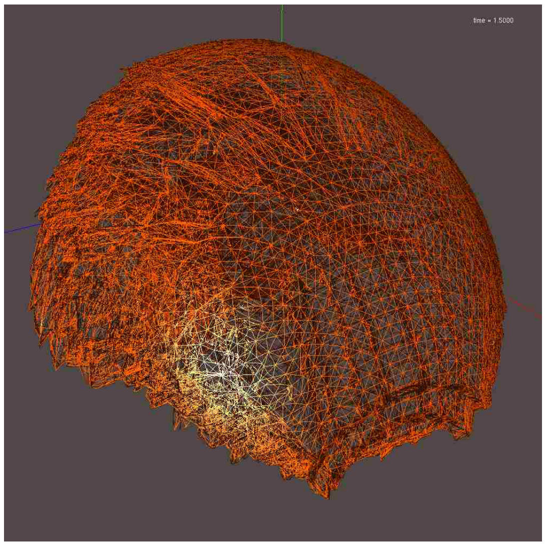
(a) Vector angle criterion (rear view, t = 2)



(b) Edge length criterion (rear view, t = 2)



(c) Vector angle criterion (front view, t = 1.5)



(d) Edge length criterion (front view, t = 1.5)

Figure 6.6: Mesh comparison reveals that the vector angle criterion splits edges less in areas where the vector field has low curvature.

Table 6.1: Vector angle refinement statistics for the purple time surface in Figure 6.6

Timestep	Refinement time	Integration time	Vertices	Edges	Faces
0.1	97.06%	2.94%	490	1464	976
0.15	96.75%	3.25%	2004	6006	4004
0.2	97.15%	2.85%	9219	27651	18434
0.25	97.38%	2.62%	16711	50127	33418
0.3	98.01%	1.99%	33959	101871	67914
0.35	97.78%	2.22%	52598	157788	105192
...	...	...	...	...	...
1.75	97.32%	2.68%	866903	2600703	1733802
1.8	97.73%	2.27%	921666	2764992	1843328
1.85	97.34%	2.66%	970792	2912370	1941580
1.9	97.79%	2.21%	1026342	3079020	2052680
1.95	97.35%	2.65%	1077065	3231189	2154126

Table 6.2: Edge length refinement statistics for the orange time surface in Figure 6.6

Timestep	Refinement time	Integration time	Vertices	Edges	Faces
0.1	94.59 %	5.41 %	870	2604	1736
0.15	59.80 %	40.20 %	2846	8532	5688
0.2	89.38 %	10.63 %	5040	15114	10076
0.25	48.57 %	51.43 %	9576	28722	19148
0.3	87.91 %	12.09 %	14178	42528	28352
0.35	44.51 %	55.49 %	21347	64035	42690
...	...	...	...	...	...
1.75	40.57 %	59.43 %	964637	2893905	1929270
1.8	85.59 %	14.41 %	1027253	3081753	2054502
1.85	40.82 %	59.18 %	1090991	3272967	2181978
1.9	85.57 %	14.43 %	1151878	3455628	2303752
1.95	41.54 %	58.46 %	1221535	3664599	2443066

Table 6.3: Time surface generation timing examples

Time surface generation parameters:	Time:
20 timestep-dataset, $\Delta t_{integration} = 0.05$, edge-length = 0.02	16.94 min
20 timestep-dataset, $\Delta t_{integration} = 0.05$, angle = 2°, edge-length = 0.02	1.16 hours
10 timestep-dataset, $\Delta t_{integration} = 0.05$, edge-length = 0.03	26.67 sec
10 timestep-dataset, $\Delta t_{integration} = 0.05$, angle = 4°, edge-length = 0.03	4.80 min

6.2.2 Ottino's Mixing Quantities

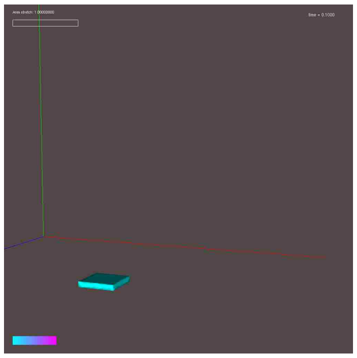
Once we generated time surfaces, we were able to easily approximate Ottino's mixing quantities. For surface stretching, we considered the following surface area measurements:

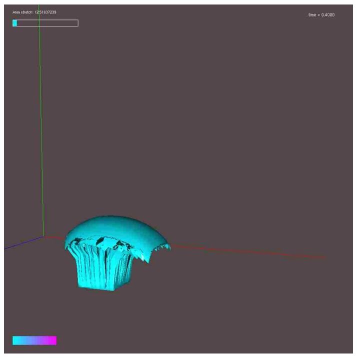
1. Surface area,
2. Surface area stretch ($\eta(t)$ see eq. 2.3),
3. Natural log of surface area stretch,
4. Rate of surface area stretch, and
5. Rate of the natural log of surface area stretch (see eq. 2.6).

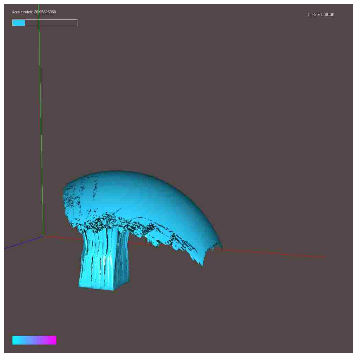
The above measurements (1), (3), and (4) are intermediate information that provide useful information about the deformation of the surface. Likewise, we have the following surface volume measurements:

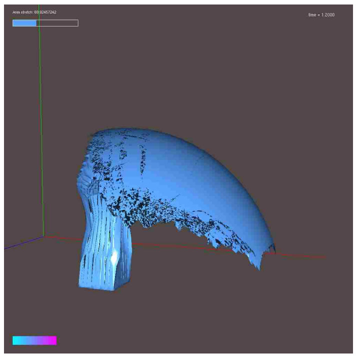
1. Surface volume,
2. Volume stretch ($\upsilon(t)$ see eq. 2.4),
3. Natural log of the volume stretch,
4. Rate of the volume stretch, and
5. Rate of the natural logarithm of volume stretch (see eq. 2.7).

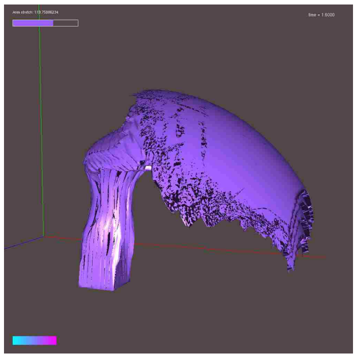
We decided to plot these time-dependent scalar values directly onto the time surface itself (Figure 6.7). The user chooses a colormap, and we assign a color value from the colormap that corresponds to a particular timestep the user is examining. We used the purple time surface that we generated in the previous section ($\theta_{max} = 2^\circ$ and $\Delta t = 0.05$. See Figure 6.6). After $t = 1.35$, the plume reaches the end of the computational domain. Since the velocity outside the domain is zero, the plume begins to flatten when it reaches plane $yz = 1$. This becomes even more apparent at

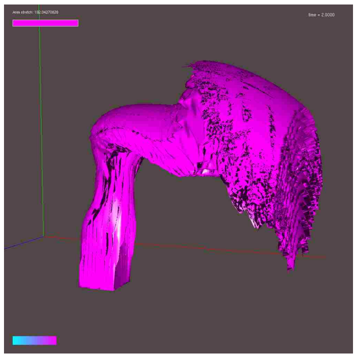

(a) $\eta(0)$=1

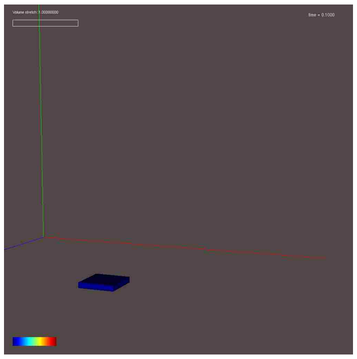

(b) $\eta(0.4) = 12.52$

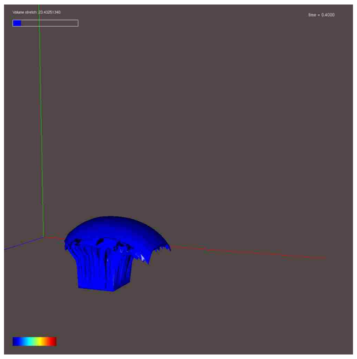

(c) $\eta(0.8) = 36.96$

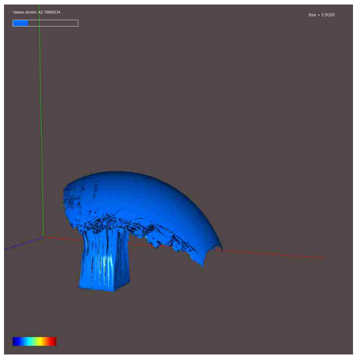

(d) $\eta(1.2) = 69.92$

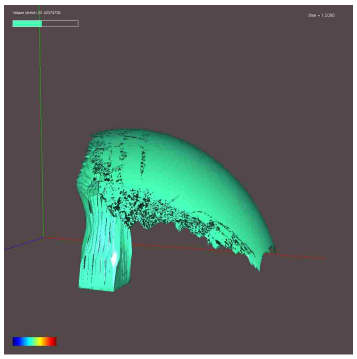

(e) $\eta(1.6) = 119.76$

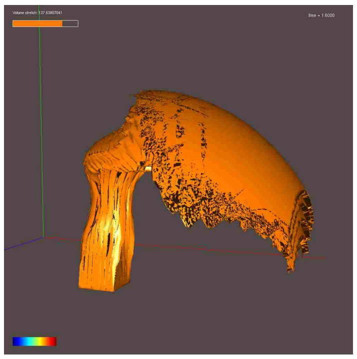

(f) $\eta(2) = 192.04$


(g) $\upsilon(0) = 1$


(h) $\upsilon(0.4) = 23.43$


(i) $\upsilon(0.8) = 42.80$


(j) $\upsilon(1.2) = 81.42$


(k) $\upsilon(1.6) = 137.84$

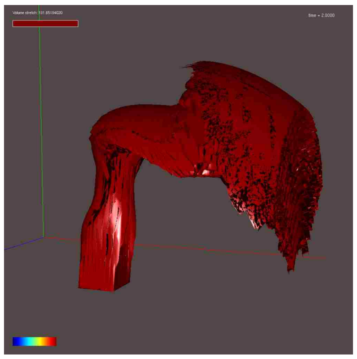
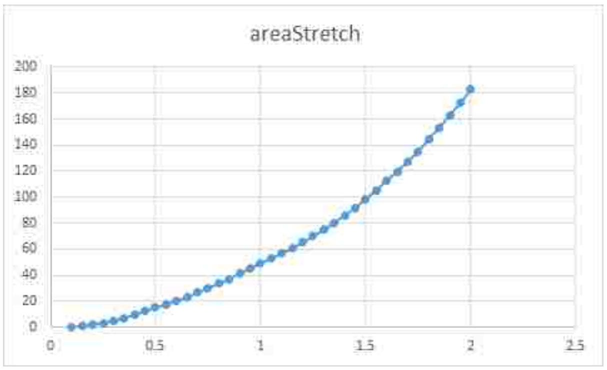

(l) $\upsilon(2)$=181.65

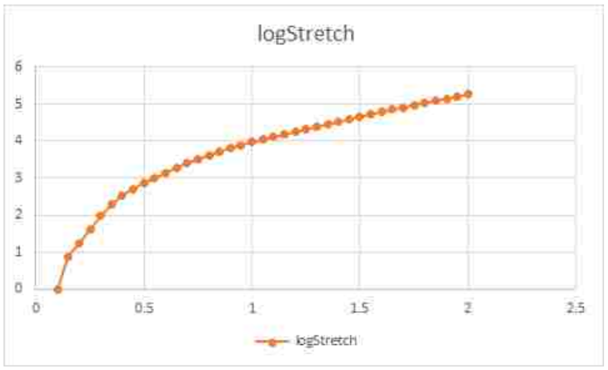
Figure 6.7: Area stretch and volume stretch mapped onto the oil plume.

$t = 2$. At $t = 0$ (Figure 6.7(a)), the surface stretch is 1. At $t = 1.2$ (Figure 6.7(d)), the surface area is 69.92 times bigger than at $t = 0$. At $t = 2$ (Figure 6.7(f)), the surface stretch of the oil plume is $\eta(2) = 192.04$. If we knew the velocity outside the computational domain or if our computational domain were bigger, then the value of $\eta(2)$ would be higher than it is. The same observations are also valid for the volume stretch of the oil plume. At $t = 2$ (Figure 6.7(l)), the volume of the plume stretched 181.65 times.

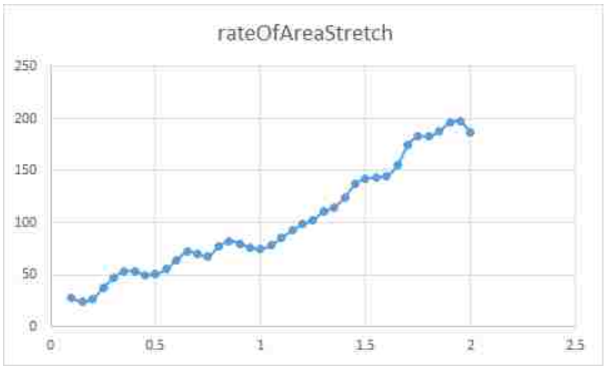
In Figures 6.8 and 6.9, we plotted conventional 2D graphs of Ottino's mixing indicators. From these graphs, we notice that looking at the logarithm quantities are not very illuminating since our surfaces do not grow exponentially. The logarithmic indicators are more informative in cases where a surface is growing at an exponential rate.



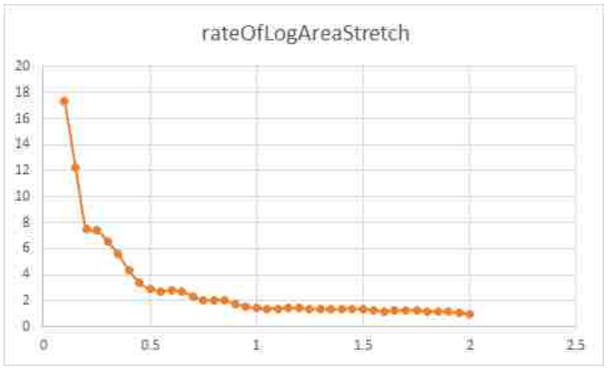
(a) area streatch (eq. 2.3)



(b) *ln*(area stretch)

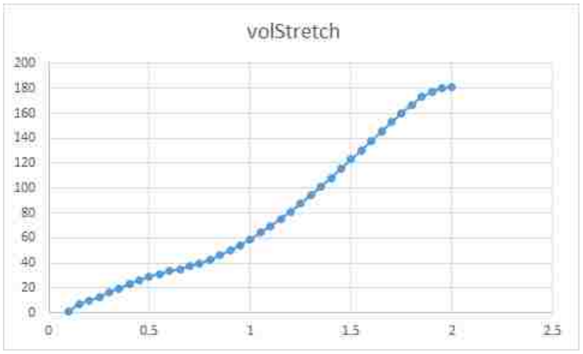


(c) rate of area stretch

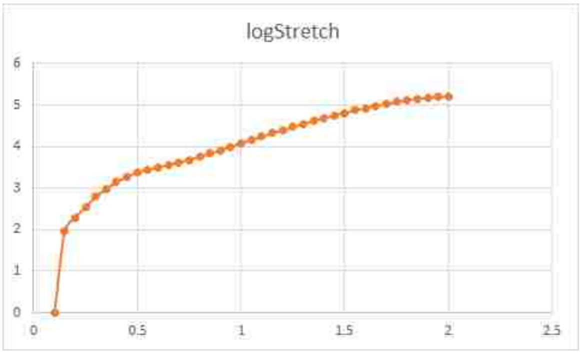


(d) rate of *ln*(area stretch) (eq. 2.6)

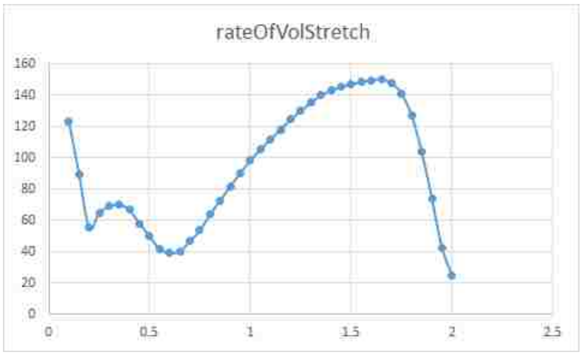
Figure 6.8: Ottino's surface area measurements.



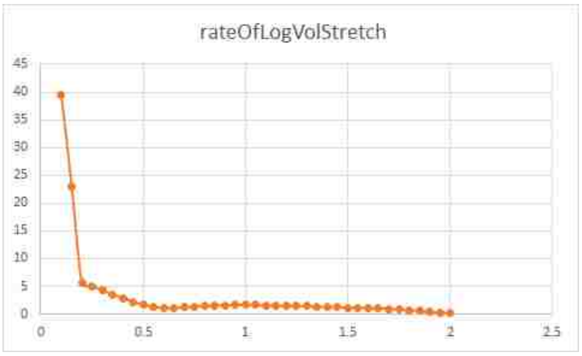
(a) volume streatch (eq. 2.4)



(b) *ln*(vol. stretch)



(c) rate of vol. stretch



(d) rate of *ln*(vol. stretch (eq. 2.7))

Figure 6.9: Ottino's surface volume measurements.

From Figure 6.8(a), we are able to determine how fast the surface area of the plume grows. Using Excel's curve regression functionality, we were able to determine that as time progresses, the area stretch grows as a 4^{th} order polynomial (Figure 6.8(a)):

$$\eta(t) = 10.9t^4 - 31.64t^3 + 61.199t^2 + 9.8122t - 2.0205 \tag{6.1}$$

Figure 6.8(b) is obtained by taking the natural logarithm of the values in Figure 6.8(a). If the area stretch were exponential, this graph would be more useful. From Figure 6.8(d), we see that the rate of area stretch drops as time advances. However, examining directly the rate of area stretch (Figure 6.8(c)) indicates the nuances of the rate of growth.

From Figure 6.9(a), we are able to determine how fast the volume of the plume grows. From Figure 6.9(d), we see that the rate of volume stretch drops as time advances. However, examining directly the rate of area stretch (Figure 6.9(c)) indicates a more nuanced rate of volume growth.

Examining Figure 6.9(c), the rate of volume stretch drops until $t = 0.6$. After $t = 0.6$, the volume grows rapidly. Looking at Figure 6.7, this coincides with the growth of the top surface, or the mushroom cap of the plume. Advanced by the ocean current, the mushroom cap continues to grow and the rate of the volume stretch spikes until $t = 1.5$. After $t = 1.5$, the rate of growth slows and drops as the mushroom cap hits the end of the computational domain (plane $yz = 1$).

6.2.3 Volume of the Oil Plume

Another benefit of computing the Ottino mixing indicators using time surfaces is that we can compute the volume of the oil plume. For a time surface generated using $\theta_{max} = 4^\circ$ and $\Delta t = 0.025$, the volume of the plume at $t = 2.0$ is 0.0385365. The volume of the plume was computed using equation 3.14. Since the volume of the computation domain is 1 and the volume of the physical domain is $(4.8m)^3 = 110.592m^3$, it follows that the volume of the plume at $t = 2.0$ is $0.0385365 \times 110.592 = 4.26m^3$. Since $1m^3 = 1000L = 264.172gal$, then the volume of the plume is $4.26m^3 = 4260L = 1125gal$.

Chapter 7
Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented a powerful Lagrangian-based surface generation method that is capable of letting the user choose a wide variety of surface refinement criteria, such as the novel vector angle criterion. The time surface algorithm is divided into two independent parts (Section 3.5): front advection and surface refinement.

- For the front advection portion, we used a 4^{th} order Runge-Kutta numerical integration method (Section 3.2). In order to apply this higher order integration method, we had to implement multi-block interpolation of the velocity field in space and in time (Section 3.3).
- For the surface refinement portion, we proposed the novel vector angle criterion. This criterion examines the angle between two vectors at a given edge. If the vectors are parallel, we do nothing. If the vectors are converging, we collapse the edge. If the vectors are diverging we split the edge. In section 4.3.4, we defined the various vector-angle scenarios and how to handle each scenario.

The are two main benefits to using the vector angle criterion. First, this criterion allows us to predict when an edge will stretch or shrink. Consequently, we can refine a surface before advancing it; we refer to this as pre-refinement. When using the conventional edge length criterion, we are using post-refinement. Pre-refinement reduces interpolation error because we start refining the surface early, and hence we reduce the growth of interpolation errors. In section 4.5.2, we compared the effect of pre-refinement and post-refinement on a coarse spherical seed surface and showed that the interpolation errors were smaller in the case of pre-refinement. Second, a surface refined using the vector angle criterion is visually better than a surface refined with the edge length criterion. Using the edge length criterion, the resulting triangular mesh will have an even distribution of vertices throughout the surface. However, using the vector angle criterion, the mesh will have fewer vertices when the vector field is flat and more vertices where the vector field is curved and turbulent. One drawback to the vector angle method is that it is slower compared to the edge length method.

Another contribution of our research is that we used the time surfaces in a novel way to evaluate Ottino's mixing criteria, and to quantify mixing quality of fluids. The time surfaces that we generated represent the material surfaces that are stretching and folding according to the fluid flow patterns. Taking advantage of the geometric information of the time surfaces, we were able to evaluate surface areas and volumes and easily compute Ottino's mixing quantities. We computed approximation to the area stretch of a material surface $\eta(t)$, the volume stretch of a material surface $\upsilon(t)$, the rate of area stretch, and the rate of volume stretch. We demonstrated these quantities for the oil spill dataset. We were able to track the the stretching of the oil plume and the rate at which it was stretching. In addition, we were able to evaluate the volume of the oil that was gushing out of the broken well.

To demonstrate our time surfaces algorithm and its applications, we developed a new software called *Dessine-Moi*. Our software supported efficient storage and handling of multi-block data using the HDF5 binary format and a hybrid implementation of the CGNS data model. We used the Qt UI framework to design a clean and easy-to-use GUI that allows the user to open a dataset, load variables, create seed surfaces, define time surfaces parameters, compute time surfaces, and render and examine time surfaces. Once a time surface is generated, the user can then compute and visualize Ottino's mixing criteria.

We also computed and rendered the curvature (eq. 2.10) and torsion (eq. 2.11) of the vector field itself to show regions of high and low mixing using a multi-scale colormap. Unlike a uniform colormap, a multi-scale colormap is a piecewise linear mapping that allows one to better observe the entire range of values of a given scalar field, including small, medium, and large values. We applied this color coding scheme on the stirred tank dataset. Using VISH, we rendered orthoslices of the two scalar fields that we computed, as described in section 3.6.2. We designed multi-scale colormaps for the selected orthoslices and we were able to show that regions of high mixing (i.e.

high curvature or high torsion) were typically located near the baffle regions, and regions of low mixing (i.e. low curvature or low torsion) were located in the interior regions of the tank.

7.2 Future Work

Several improvements and additions can be made to the time surfaces algorithm and to *Dessine-Moi*. Some of these improvements are listed below:

- *Improve the performance of the edge collapse algorithm.* By design, our edge collapse operation performance should be linear in the number of edges, just like our edge split and edge collapse. However, due to incorrect relabeling of edge-to-faces, the performance of our collapse operation is quadratic in the number of edges.

- *Parallelization.* First, we would like to support multi-patch surfaces. This type of surfaces is described in the CGNS data model. To support it in *Dessine-Moi*, we need to change how the surface refinement operations iterate through the mesh data structure. This should not be very difficult. Second, once we parallelize the surface data structure, we would like to parallelize the edge split, edge flip, and edge collapse. Here, special care is needed when an edge lies at the boundary of two surface patches. Third, we would like to parallelize the vector field interpolation to improve the performance of the vector angle criterion.

- *Implement curvilinear interpolation in Dessine-Moi.* This would allow us to test and analyze a wider range of CFD datasets.

- *Identify topological changes in a time surface.* We can combine front capturing approaches with our current front tracking method. Front capturing methods are better at detecting when an evolving surface splits or when two separate surfaces merge into one. A good starting point would be to implement something similar to what Wojtan et al. presented in [38].

- *Render scalar fields on the time surfaces.* For example, the datasets at our disposal also store the pressure field of the flow. We could use a uniform or a multi-scale colormap and render the pressure on the time surfaces. This would require that we interpolate the scalar field onto the time surface. Other scalar fields worth considering are the curvature and torsion of the vector field. This would require migrating the code we developed from VISH to *Dessine-Moi*.

- *Implement transparent rendering of the time surfaces*. This would enhance the visual quality of the time surfaces. Furthermore, we would better see and understand the deformations and changes in the time surfaces.

References

[1] John D Anderson. *Computational Fluid Dynamics. The Basics with Applications*. McGraw-Hill, 1995.

[2] Werner Benger. *Visualization of general relativistic tensor fields via a fiber bundle data model*. Lehmanns Media-LOB. de, 2005.

[3] Werner Benger, Georg Ritter, and René Heinzl. The concepts of vish. In *4th High-End Visualization Workshop, Obergurgl, Tyrol, Austria*, pages 26–39, 2007.

[4] Bidur Bohara, Werner Benger, Marcel Ritter, Nathan Brener, S Sitharama Iyengar, Bijaya Karki, Somnath Roy, and Sumanta Acharya. Time-curvature and time-torsion of virtual bubbles as fluid mixing indicators. *Proceedings of the IADIS Internation Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2010 (CGVCVIP 2010), ISBN (Book) 978-972-8939-22-9*, 2010.

[5] Bidur Bohara, Farid Harhad, Werner Benger, Nathan Brener, S Sitharama Iyengar, Bijaya B Karki, Marcel Ritter, Kexi Liu, Brygg Ullmer, Somnath Roy, and Sumanta Acharya. Evolving time surfaces in a virtual stirred tank. *Journal of WSCG*, 18:121–128, 2010.

[6] Nathan Brener, Farid Harhad, Bijaya Karki, Werner Benger, Sumanta Acharya, Marcel Ritter, and S Sitharama Iyengar. Multi scale color coding of derived curvature and torsion fields on a multi-block curvilinear grid. In *WSCG2014 22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2014, Plzen, Czech Republic*, 2014.

[7] Richard L Burden, J. Douglas Faires, and Albert C. Reynolds. *Numerical Analysis*. Prindle, Weber & Schmidt, 1978.

[8] Kai Burger, Florian Ferstl, Holger Theisel, and Rüdiger Westermann. Interactive streak surface visualization on the gpu. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1259–1266, 2009.

[9] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324. ACM Press/Addison-Wesley Publishing Co., 1999.

[10] Chen-shi Dong and Guo-zhao Wang. Curvatures estimation on triangular mesh. *Journal of Zhejiang University SCIENCE*, 6(1):128–136, 2005.

[11] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116, 2002.

[12] Laurene Fausett. *Numerical Methods Using MathCad*. Printice Hall, 2002.

[13] Stanley Osher Ronald Fedkiw. Level set methods and dynamic implicit surfaces. 2003.

[14] Ryan Hafen, William S Cleveland, and David S Ebert. Automated box-cox transformations for improved visual encoding. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 19(1), 2013.

[15] Cyril W Hirt and Billy D Nichols. Volume of fluid (vof) method for the dynamics of free boundaries. *Journal of computational physics*, 39(1):201–225, 1981.

[16] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26. ACM, 1993.

[17] Jeffrey PM Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of the 3rd conference on Visualization'92*, pages 171–178. IEEE Computer Society Press, 1992.

[18] Xiangmin Jiao, Andrew Colombi, Xinlai Ni, and John Hart. Anisotropic mesh adaptation for evolving triangulated surfaces. *Engineering with Computers*, 26(4):363–376, 2010.

[19] Sandeep Khurana, Nathan Brener, Bijaya Karki, Werner Benger, Somnath Roy, Sumanta Acharya, Marcel Ritter, and S Sitharama Iyengar. Multi scale color coding of fluid flow mixing indicators along integration lines. In *WSCG2012-20-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, June 25-28, 2012, Plzen, Czech Republic*, 2012.

[20] Gordon Kindlmann and James W Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86. ACM, 1998.

[21] Hari Krishnan, Christoph Garth, and Kenneth I Joy. Time and streak surfaces for flow visualization in large time-varying data sets. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1267–1274, 2009.

[22] Tony McLoughlin, Robert S Laramee, and Eugene Zhang. Easy integral surfaces: a fast, quad-based stream and path surface algorithm. In *Proceedings of the 2009 Computer Graphics International Conference*, pages 73–82. ACM, 2009.

[23] Julio M Ottino. *The kinematics of mixing: stretching, chaos, and transport*, volume 3. Cambridge University Press, 1989.

[24] Hanspeter Pfister, Bill Lorensen, Chandrajit Bajaj, Gordon Kindlmann, Will Schroeder, Lisa Sobierajski Avila, KM Raghu, Raghu Machiraju, and Jinho Lee. The transfer function bake-off. *Computer Graphics and Applications, IEEE*, 21(3):16–22, 2001.

[25] Diane Poirier, Steven R Allmaras, Douglas R McCarthy, Matthew F Smith, and Fancis Y Enomoto. The cgns system. *AIAA Paper*, pages 98–3007, 1998.

[26] Somnath Roy and Sumanta Acharya. Study on flow and turbulence inside a stirred tank and investigation on the effects of macroinstability on trailing vortex structures. In *ASME 2007 International Mechanical Engineering Congress and Exposition*, pages 79–89. American Society of Mechanical Engineers, 2007.

[27] Christopher L Rumsey, Bruce Wedan, Thomas Hauser, and Marc Poinot. Recent updates to the cfd general notation system (cgns). In *50th AIAA Aerospace Sciences Meeting*, volume 10, pages 6–2012, 2012.

[28] Tobias Schafhitzel, Eduardo Tejada, Daniel Weiskopf, and Thomas Ertl. Point-based stream surfaces and path surfaces. In *Proceedings of Graphics Interface 2007*, pages 289–296. ACM, 2007.

[29] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit. An Object-Oriented Approach To 3D Graphics*. Kitware Inc., 2006.

[30] James A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[31] Alexander J Smits and TT Lim. *Flow Visualization, Techniques and Examples*. Imperial College Press, 2000.

[32] Rob Sturman, Julio M Ottino, and Stephen Wiggins. *The mathematical foundations of mixing: the linked twist map as a paradigm in applications: micro to macro, fluids to solids*, volume 22. Cambridge University Press, 2006.

[33] Gabriel Taubin. Curve and surface smoothing without shrinkage. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 852–857. IEEE, 1995.

[34] Grétar Tryggvason, Bernard Bunner, Asghar Esmaeeli, Damir Juric, N Al-Rawahi, W Tauber, J Han, S Nas, and Y-J Jan. A front-tracking method for the computations of multiphase flow. *Journal of Computational Physics*, 169(2):708–759, 2001.

[35] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson Education, 2007.

[36] Wolfram Von Funck, Tino Weinkauf, Holger Theisel, and H-P Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1396–1403, 2008.

[37] T Weinkauf and H Theisel. Curvature measures of 3d vector fields and their applications. *Journal of WSCG*, 10(2):507–514, 2002.

[38] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Deforming meshes that split and merge. In *ACM Transactions on Graphics (TOG)*, volume 28, page 76. ACM, 2009.

Vita

Farid Harhad was born in Baton Rouge, Louisiana, in July 1979. He completed his undergraduate studies with honors in Computer Science at Louisiana State University, in December 2004. In 2005, he worked for a web services company called VoterVoice. In Fall 2006, he join the computer science doctorate program at Louisiana State University. There he worked with Dr. Bijaya Karki and Dr. Sumanta Acharya on flow visualization research topics. His research area is in scientific visualization. From 2008 to 20012, he was the recipient of the NSF-IGERT on Mulitiscale CFD fellowship. As an IGERT fellow, he was able to learn from and interact with many bright professors and students who studied a wide variety of CFD problems that appeared in many disciplines such as astrophysics, chemical engineering, and oceanography, to name a few.

While in graduate school, Farid worked several times for Schlumberger as an intern. In June 2007, he worked as a summer intern at Schlumberger Information Solutions, Houston. There, he worked with the Petrel Drilling Visualization team on a 3D graphical module used to plan well drilling and asses drilling risks. In the summer of 2008, he joined the Schlumberger Riboud Product Center, in Clamart, France. There, he developed a 3D visualization model of the process of primary well cementing for *WellBook*. And in June 2010, he joined WesternGeco (Schlumberger), in Houston. There, he developed a module that allows a user to filter a seismic volume on the GPU with *Petrel*.