A Level Set Approach for Denoising and Adaptively Smoothing Complex Geometry Stereolithography (STL) Files

by

Karthik Kannan

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved November 2014 by the
Graduate Supervisory Committee:

Dr. Marcus Herrmann, Chair
Dr. Yulia Peet
Dr. David Frakes

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

Stereolithography files (STL) are widely used in diverse fields as a means of describing complex geometries through surface triangulations. The resulting stereolithography output is a result of either experimental measurements, or computer-aided design. Often times stereolithography outputs from experimental means are prone to noise, surface irregularities and holes in an otherwise closed surface.

A general method for denoising and adaptively smoothing these dirty stereolithography files is proposed. Unlike existing means, this approach aims to smoothen the dirty surface representation by utilizing the well established levelset method. The level of smoothing and denoising can be set depending on a per-requirement basis by means of input parameters. Once the surface representation is smoothened as desired, it can be extracted as a standard levelset scalar isosurface.

The approach presented in this thesis is also coupled to a fully unstructured cartesian mesh generation library with built-in localized adaptive mesh refinement (AMR) capabilities, thereby ensuring lower computational cost while also providing sufficient resolution. Future work will focus on implementing tetrahedral cuts to the base hexahedral mesh structure in order to extract a fully unstructured hexahedra-dominant mesh describing the STL geometry, which can be used for fluid flow simulations.

ACKNOWLEDGEMENTS

I would like to thank Dr. Marcus Herrmann for not just giving me an opportunity to work with him, but also being immensely patient with me and helping me whenever I hit a roadblock during my research. I would also like to extend my thanks to Dr. Yulia Peet and Dr. David Frakes for agreeing to be a part of this endeavor.

I would also like to thank my lab mate, Carlos Ballesteros, for patiently helping me understand and get around FARCOM, which would have otherwise taken me a great deal of time.

TABLE OF CONTENTS

Page

LIST OF FIGURES v

LIST OF ALGORITHMS vii

CHAPTER

1 Introduction 1

1.1 Background 1

1.2 Motivation and Scope 3

1.3 Applications 4

2 Level Set Method 5

2.1 Introduction 5

2.1.1 The Level Set Scalar 5

2.1.2 Choices of Level Set Function 7

2.2 Advection of the Level Set Scalar 7

2.3 Reinitialization of the Level Set Scalar 9

3 FARCOM: Fortran Adaptive Refiner for Cartesian Orthogonal Meshes . . . 12

3.1 Introduction 12

3.2 General Features 13

3.3 Limitations 16

4 Systematic Numerical Method 17

4.1 Introduction 17

4.2 Establish Triangle Information in Cartesian Grid 17

4.2.1 Preparatory Work 18

4.2.2 Initial Sorting 19

4.3 Adaptive Mesh Refinement 21

4.3.1 Refinement Selector Criteria 21

CHAPTER Page

4.3.2 Re-sorting the Triangulation Data 23

4.4 Initialization of the Level Set Field 24

4.4.1 Computing Initially Accepted Level Set Values 25

4.4.2 Obtaining Full Level Set Field 26

4.5 Smoothing of the Interface 27

4.5.1 Excessive Reinitialization of the Level Set Field 28

4.5.2 Mean Curvature Flow 28

5 Results 31

5.1 Adaptive Mesh Refinement 31

5.2 Initially Accepted Level Set Values 34

5.3 Smoothened Level Set Isosurface 36

6 Conclusion 40

7 Future Work 41

REFERENCES 42

APPENDIX

A Triangle-AABB Intersection Algorithm Fortran Code 44

LIST OF FIGURES

Figure Page

2.1 Implicit Representation of a circle in two dimensions. 6

3.1 Adaptive mesh refinement on sample domain (Ballesteros, 2015). Top left: Domain with interface. Top right: Domain with coarse mesh, with intersecting cells highlighted. Bottom left: Domain with equidistant finer mesh. Bottom right: Adaptively refined mesh, providing same level of resolution as the finer mesh. 13

3.2 Two dimensional unstructured Cartesian AMR (Ballesteros, 2015). Intensity of the color gray indicates level of refinement. 14

3.3 Constraints on the refinement operation (Ballesteros, 2015). Green cells do not pose any refinement restrictions; red cells block the refinement operation; cyan cells are currently being refined 15

4.1 Sorting of triangulations based on location of centroid. Triangle 1 counts as being sorted and intersecting, whereas triangle 2 counts only as intersecting. 19

5.1 A simple sphere in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation 32

5.2 A vase in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation 33

5.3 A twisted knot in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section along a plane cutting through the geometry of the adaptively refined AMR grid based on concentration of surface triangulation 34

Figure Page

5.4 Spray A injector nozzle geometry: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation 35

5.5 Initially accepted zero level set isolines of the various test cases. The black line represents the original STL file, and the orange line represents the reconstructed level set surface 37

5.6 Cross-sectional snapshots of excessive smoothing on the various test cases. The black line represents the original STL file, and the orange line represents the reconstructed level set surface 39

LIST OF ALGORITHMS

Algorithm Page

1 Initial stage sorting of triangle facets to underlying Cartesian mesh 20

2 Refinement selection criteria for adaptively refining a control volume... 23

3 Re-sort triangle data of control volume n into its daughter cells 24

4 Compute minimum distance level set for a mesh cell containing facets . 26

Chapter 1

INTRODUCTION

1.1 Background

With the increasing growth of technology and computing capabilities, the ability to conduct greater levels of computational research has increased, specifically in the field of computational fluid dynamics. This specific field has seen major growth in terms of accurate modeling, higher order approximations, and more efficient algorithms able to run on massively parallel architectures. However, these numerical methods and complex algorithms are as only as good as the underlying mapping mechanism of the problem in hand – the computational mesh. Scientific problems which have a geometry related to them have generally been modeled and meshed by approximate means.

With advances in technology came the ability to create complex geometry models from scratch by means of computer-aided design, or measure existing geometries by experimental measurements. The result of these methods of modeling a complex geometry can be distributed in a variety of formats, but nothing which can be a direct input to a simulation tool. Of the various formats available for distribution, the stereolithography (STL) file format is a generally attractive format for describing a geometry by means of surface triangulations, for being cross-compatible and also non-proprietary. STL also known as *Standard Tessellation Language*, is widely used for rapid prototyping, computer-aided manufacturing; among others. The STL file format describes a triangulated surface by the unit normal and vertices of the triangles using a three-dimensional Cartesian coordinate system. They can be encoded in

both ASCII (American Standard Code for Information Exchange), or binary (Bourke, 1999), with the latter resulting in faster access times and smaller file sizes. The general format of an ASCII STL file is given below:

```
solid name_of_solid
.
.
.
    facet normal nx ny nz
        outer loop
            vertex v1x v1y v1z
            vertex v2x v2y v2z
            vertex v3x v3y v3z
        endloop
    endfacet
.
.
.
endsolid name_of_solid
```

The description of the surface geometry in the STL file is contained within the `solid` and `endsolid` constructs. Each triangle information is specified within the `facet` and `endfacet` constructs, providing both triangle face normals (n_x, n_y, n_z) and vertex coordinates (v_x, v_y, v_z) for each vertex in the triangle. Similarly, a binary STL file format is given below:

```
header
ntriangles
nx ny nz
v1x v1y v1z
v2x v2y v2z
v3x v3y v3z
spacer
.
.
.
```

The first line consists of a 80 byte `header`, which is a comment string produced by the tool used to generate the STL file. The following line consists of a 4 byte integer value `ntriangles`, which is the number of triangles describing the surface geometry.

The triangle face normals and vertex coordinates of each triangle vertex are provided similar to the ASCII format as a 4 byte floating point number. At the end of every facet (triangle face), there is a 2 byte `spacer` present. This pattern follows till all the triangles are described in the file.

1.2 Motivation and Scope

With the ability to describe complex geometries through data files, comes the need to be able to transfer the information onto the computational domain accurately, in order to be able to analyze the geometry by any means necessary. However, the way to do that is not free of obstacles, as often times the STL files resulting from experimental methods, such as x-ray tomography measurements are prone to noise, or holes in an otherwise closed surface. This is a result of the limited resolution available while performing such measurements. This can also be said about STL files resulting from computer-aided design, in which the surface may not seem to be "closed" due to a variety of reasons, one of them being round-off errors.

The available resources to "fix" these holes are, for the most part, by way of commercial softwares (ANSYS, Netfabb, Magics, Uformia Meshup etc.), which resort to fixing these holes by brute geometry patching of the open surfaces. Although this addresses the issue of holes in a closed surface, it may result in rough edges, and undesired changes in topology. Previous work has been done by Furlong and Genzale (2012) on smoothing rough STL geometry by means of spline fits, and some open-source software capability exists in literature (OpenFOAM etc.). In the current work, a level set approach is taken to ensure that the STL geometry can be rid of noise and close open surfaces while maintaining relative surface smoothness. The scope of this thesis can be outlined in four major objectives, mainly: (1) to establish the surface triangulation information on an adaptive Cartesian mesh; (2) to demonstrate smooth-

ing of a surface interface, in order to remove noise; (3) to provide an initial scalar field describing the geometry for multiphase interface capturing methods; and (4) to provide a means of obtaining a high quality hexahedra dominant mesh containing the STL surface geometry.

The thesis is broken down into different chapters, each serving a different purpose. Chapter 2 will provide an overview of the level set method, introduce the level set scalar, and the various level set equations. Chapter 3 will provide background on the adaptive mesh refinement (AMR) library utilized with the current work, discussing some of its features and limitations. Chapter 4 will provide a technical description of the systematic numerical method, starting from the lookup between the triangulations and the cartesian grid, to the end smoothing process. Chapter 5 will provide results of simulations of a few test cases. Chapter 6 will provide a summary of the work performed and what has been demonstrated in the thesis. The final chapter, Chapter 7, will provide an overview of general additions to the existing code architecture, in order to make it more efficient and increase the number of applications the current work can serve.

1.3 Applications

The applications served by the current work are plenty, for example, upon generation of a level set scalar field from a STL geometry, the same field can be utilized as a means of describing the initial surface geometry in multiphase flow simulations. The smoothened geometry level set isosurface can be extracted as a cleaned up STL file, which can then be used in a multitude of applications, ranging from mesh generation for numerical simulations or even computer-aided manufacturing. Additionally, there are other applications, which can be fully realized upon further additions / modifications to the current work, which are discussed in greater detail in Chapter 7.

Chapter 2

LEVEL SET METHOD

This chapter introduces the level set scalar, implicit surfaces and some of the various methods associated with the level set scalar.

2.1 Introduction

The level set method was introduced by Osher and Sethian (1988) as a means of capturing the motion of an interface Γ in either two or three dimensions. While there are many different interface-tracking or interface-capturing methods in the literature; such as front-tracking methods (Tryggvason *et al.*, 2001), and volume-of-fluid methods (Lörstad *et al.*, 2004), among others; level-set methods (Osher and Fedkiw, 2002) have proven to be quite attractive for their high accuracy in capturing and tracking the interface, but have certain issues in volume conservation (Herrmann, 2006). This however, can be overcome by making use of complex level set formulations with better mass conservation properties, such as the conservative level set method (Olsson and Kreiss, 2005). Other known issues with a level set method are with high local curvatures, which can be solved by using a high quality finer mesh, or refine the existing mesh locally in areas of high curvature. The current work involves the latter option, which is discussed in brief in Chapters [3-4].

2.1.1 The Level Set Scalar

Consider an interface Γ which bounds a region Ω; the level set scalar, denoted by ϕ, is a spatial function in the domain of interest, such that ϕ is positive inside Ω and negative outside Ω. It then follows that the ϕ is zero on Γ at all times. This can be

written as:

$$\vec{x} = \begin{cases} \text{Inside } \Gamma & \text{if } \phi(\vec{x}, t) > 0 \\ \text{Outside } \Gamma & \text{if } \phi(\vec{x}, t) < 0 \\ \text{On } \Gamma & \text{if } \phi(\vec{x}, t) = 0 \end{cases} \tag{2.1}$$

The interface captured by the zero isosurface (or isoline in 2D) is now an implicit representation of the actual surface called an *implicit surface*. The location of the interface Γ is not directly calculated but is essentially "captured" by means of interpolation of the zero level set of a smooth (at least Lipschitz continuous) function, $\phi(\vec{x}, t)$.

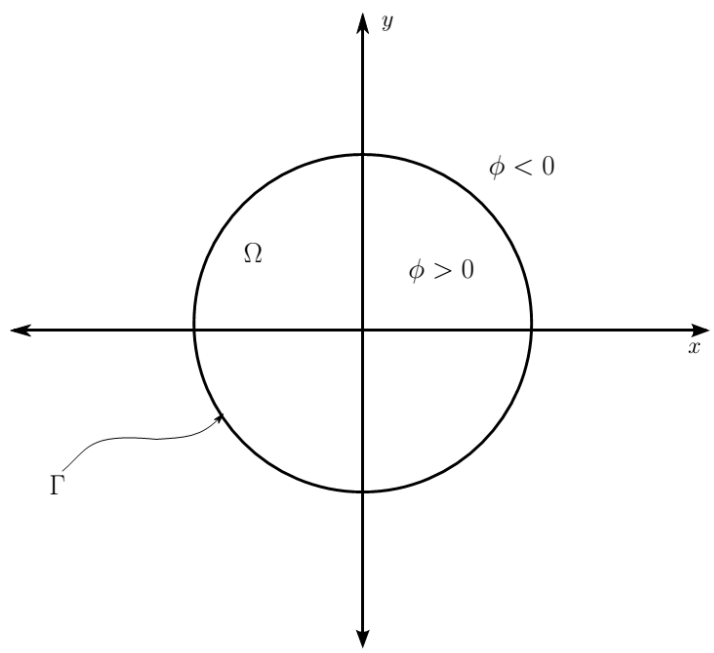


Figure 2.1: Implicit Representation of a circle in two dimensions.

For example, a given arbitrary circle in the Cartesian xy plane, as illustrated in Figure 2.1, can be captured by means of using a level set scalar as a function of the spatial location $\vec{x}$, here $\vec{x} = \vec{x}(x_1, \ldots x_n) \in \mathcal{R}^n$.

2.1.2 Choices of Level Set Function

The choice of a level set function is widely discussed in literature; including the *signed distance level set*, and the *smeared Heaviside* function; among others. The smeared Heaviside function (Olsson and Kreiss, 2005) uses the standard Heaviside function albeit with minor changes to "smooth" out the function, rather than being discontinuous. The signed distance level set function, on the other hand, is defined as:

$$|\phi(\vec{x})| = \min_{\vec{x}_f \in \Gamma} |\vec{x} - \vec{x_f}| \tag{2.2}$$

which is a smooth function with a value equal to the distance to the interface, and the sign is defined as positive on one side of the interface and negative on the other. The definition of the sign is up to preferential convention, but in order to be consistent, the current work will follow the definition stated in Equation 2.1. In contrast to the smeared Heaviside level set function, there are a number of new properties that only signed distance level set functions possess. For example,

$$|\nabla \phi| = 1 \tag{2.3}$$

This is true only for a general sense, and is not true for points that are equidistant from at least two points on the interface (Osher and Fedkiw, 2002). The interface location is defined along the $\phi = 0$ level set isoline. Signed distances are monotonic across the interface and can be differentiated there with high levels of confidence.

2.2 Advection of the Level Set Scalar

One of the primary applications of the level set scalar is tracking the motion of a front or an interface, by means of solving an advection equation. This is, in particular,

a very straightforward approach to combustion modeling, where the interface location ($\phi = 0$) separates the burnt and the unburnt mass fraction of fuel; or in two-phase multiphase modeling, where the interface separates one fluid from another. Starting from an initial level set scalar field, the interface to be captured at a later time is done by locating the set of $\Gamma(t)$ for which ϕ vanishes.

The motion of the interface is captured by convecting the level set scalar field with the velocity field $\vec{v}$. The level set advection equation is given by:

$$\frac{\partial \phi}{\partial t} + \vec{v} \cdot \nabla \phi = 0 \tag{2.4}$$

Here, $\vec{v}$ is the desired velocity on the interface, and is arbitrary elsewhere. Equation 2.4 is a Hamilton-Jacobi partial differential equation, and as such the zero level set surface corresponding to a normal propagating velocity can develop corners in finite time. This is addressed by obtaining the *weak* solution to the partial differential equation. The level set advection equation is an *Eulerian* formulation of the interface evolution, since the interface is "captured " by the implicit function ϕ, as opposed to being "tracked" by interface elements (marker particles, Scardovelli and Zaleski (1999)) in a *Lagrangian* formulation.

Standard *upwinding* finite-difference discretization of Equation 2.4 results in a good amount of numerical errors, owing to the high numerical dissipation of these methods. A better approach is to make use of upwind-biased weighted essentially non-oscillatory (WENO) schemes for Hamilton-Jacobi equations with appropriate flux functions for the spatial derivatives, such as the 5th order WENO with Roe/Local Lax-Friedrichs flux function (Jiang and Peng, 2000); and using a higher order total-variation-diminishing (TVD) Runge-Kutta schemes for time advancement, such as the 3rd order TVD Runge-Kutta method (Shu, 1988). The current work makes use of a modified level set advection in the smoothing of the level set scalar field, which

is discussed in detail in Chapter 4.

2.3 Reinitialization of the Level Set Scalar

During the process of advection of the level set scalar, it is noted that while the level set scalar field is initially a distance function, it need not necessarily remain so (Sussman *et al.*, 1994). In other words, the absolute value of the gradient of the level set scalar will not equal 1, i.e., $|\nabla\phi| \neq 1$. Additionally, ϕ can develop a jump at the interface when interfaces merge. In order to address this, the ϕ solution field at every t needs to be maintained as a distance function, which is often referred to as the *reinitialization* of the level set scalar. Although the current work does not perform an advection of the level set scalar as noted in the previous section, the reinitialization process is utilized to achieve similar results, as described in detail in Chapter 4.

Reinitialization of the level set scalar is done by solving:

$$\phi_t = \mathcal{S}_\epsilon(\phi^{(0)})(1 - |\nabla\phi|) \tag{2.5}$$

with,

$$|\nabla\phi| = \sqrt{\left(\frac{\partial\phi}{\partial x}\right)^2 + \left(\frac{\partial\phi}{\partial y}\right)^2 + \left(\frac{\partial\phi}{\partial z}\right)^2}$$

$$\mathcal{S}_\epsilon(\phi_{(0)}) = \frac{\phi_{(0)}}{\sqrt{\phi_{(0)}^2 + \epsilon^2}}$$

where, $\phi^{(0)}$ is the ϕ from before the initiation of reinitialization; $\mathcal{S}_\epsilon$ is a smoothened version of the sign function; and ϵ is an appropriately small number. Solving Equation 2.5 is often done by rewriting (Sussman *et al.*, 1994) it as:

$$\phi_t + \vec{w} \cdot \nabla\phi = \mathcal{S}_\epsilon(\phi^{(0)}) \tag{2.6}$$

with,

$$\vec{w} = \mathcal{S}_\epsilon(\phi^{(0)}) \left(\frac{\nabla \phi}{|\nabla \phi|} \right)$$

Where, $\vec{w}$ is a unit normal vector pointing outwards from the interface. Equation 2.6 is a nonlinear hyperbolic equation, with its characteristics given by $\vec{w}$. It is solved through numerical iterative methods till steady state by means of the following time advancement discretization:

$$\phi^{(k+1)} = \phi^{(k)} - \Delta\tau \, \mathcal{S}_\epsilon(\phi^{(0)}) \, \mathcal{G}(\phi^{(k)}) \tag{2.7}$$

where, $\Delta\tau$ is the reinitialization iteration time step, and $\mathcal{G}(\phi^{(k)})$ is a flux function defined as:

$$\mathcal{G}(\phi^{(n)}_{i,j,k}) = \begin{cases} \sqrt{\max((a^+)^2, (b^-)^2) + \max((c^+)^2, (d^-)^2) + \max((e^+)^2, (f^-)^2)} - 1 & \text{if } \phi^{(0)}_{i,j,k} > 0 \\ \sqrt{\max((a^-)^2, (b^+)^2) + \max((c^-)^2, (d^+)^2) + \max((e^-)^2, (f^+)^2)} - 1 & \text{if } \phi^{(0)}_{i,j,k} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

with, $x^+ = \max(x, 0)$ and $x^- = \min(x, 0)$. The subscripts i, j, k indicate discrete locations in a three-dimensional domain, with one index for each direction. The remaining undefined variables in Equation 2.8 are defined as: $a \equiv D_x^- \phi_{i,j,k}$, $b \equiv D_x^+ \phi_{i,j,k}$, $c \equiv D_y^- \phi_{i,j,k}$, $d \equiv D_y^+ \phi_{i,j,k}$, $e \equiv D_z^- \phi_{i,j,k}$, and $f \equiv D_z^+ \phi_{i,j,k}$, for three dimensions. The $D_x^\pm$ operator is a standard one-sided first-order finite-difference approximation for the first derivative. However, as discussed in the previous section, making use of lower order finite-difference schemes results in major numerical dissipation, which inherently moves the interface location undesirably.

The means of addressing this issue and providing an improvement comes in the form of implementing a 5th order weighted essentially non-oscillatory (WENO) with Roe/Local Lax-Friedrichs flux function (Jiang and Peng, 2000), as follows:

$$a \leftarrow \left.\frac{\partial \phi}{\partial x}^{(k)}\right|^{-}_{\text{WENO}}$$

$$b \leftarrow \left.\frac{\partial \phi}{\partial x}^{(k)}\right|^{+}_{\text{WENO}}$$

$$c \leftarrow \left.\frac{\partial \phi}{\partial y}^{(k)}\right|^{-}_{\text{WENO}}$$

$$d \leftarrow \left.\frac{\partial \phi}{\partial y}^{(k)}\right|^{+}_{\text{WENO}}$$

$$e \leftarrow \left.\frac{\partial \phi}{\partial z}^{(k)}\right|^{-}_{\text{WENO}}$$

$$f \leftarrow \left.\frac{\partial \phi}{\partial z}^{(k)}\right|^{+}_{\text{WENO}}$$

Coupling this to a 3rd order TVD Runge-Kutta method (Shu, 1988) time advancement scheme will ensure that any undesired interface movement is minimized.

The aforementioned reinitialization method is a PDE-based one, which may result in long runtime when trying to reach a steady state solution. Other alternatives exist in literature such as the Fast Marching Methods (FMM) (Sethian, 1999) which takes advantage of advanced sorting techniques to ensure reduced complexity.

Chapter 3

FARCOM: FORTRAN ADAPTIVE REFINER FOR CARTESIAN ORTHOGONAL MESHES

3.1 Introduction

FARCOM (Ballesteros, 2015), which stands for *Fortran Adaptive Refiner for Cartesian Orthogonal Meshes* is a general adaptive mesh refinement (AMR) library used with the approach presented in this thesis, which allows for the reduction in computational cost of the entire process by restricting the use of small grid cells only to regions where the problem requires higher resolution. For example, the library is used in this approach to provide higher resolution (smaller mesh cell sizes) near the location of the triangulation surface location, while providing lower resolution (larger mesh cell sizes) away from it.

The library provides a variety of tools to select the type of adaptive mesh to produce, as it is largely dependent on the type of geometry in question. Although the reduction in computational cost is tied to the problem itself (Calder *et al.*, 2000), there will be a drastic improvement in time taken to solution when compared to an equidistant mesh.

For example, consider a sample two dimensional domain as show in Figure 3.1. Overlaying a simple, equidistant coarse mesh will result in capturing the macroscopic properties of the curve, but will fail to resolve the "kink" present in it. A naive, first-step approach would be to overlay an equidistant mesh, this time finer, such that it does resolve all the properties of the curve. This, however, results in a very high increase in computational cost, by providing higher levels of resolution farther

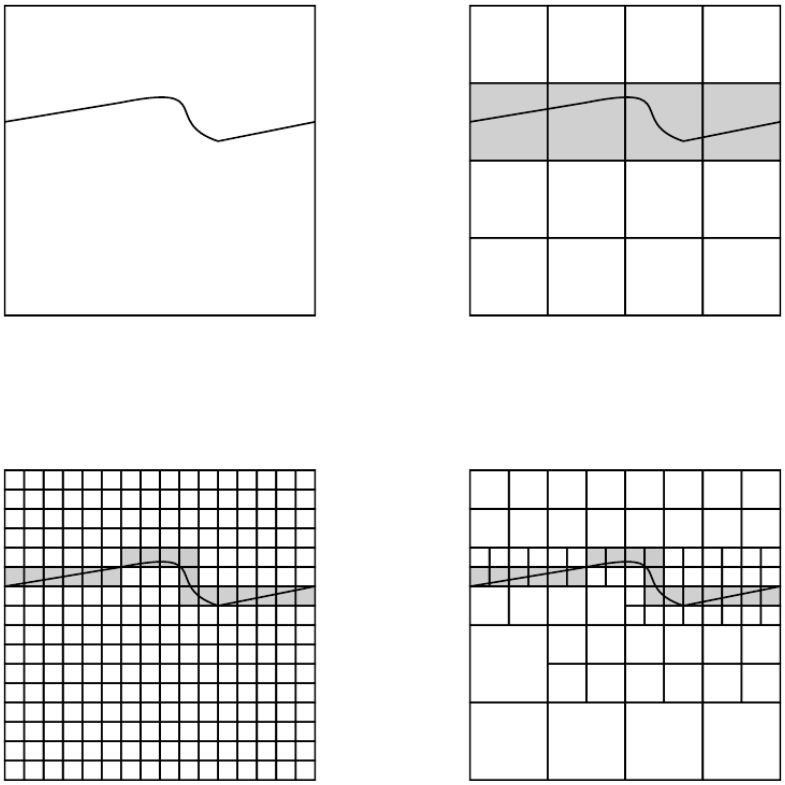


Figure 3.1: Adaptive mesh refinement on sample domain (Ballesteros, 2015). Top left: Domain with interface. Top right: Domain with coarse mesh, with intersecting cells highlighted. Bottom left: Domain with equidistant finer mesh. Bottom right: Adaptively refined mesh, providing same level of resolution as the finer mesh.

away from the curve, where it is not desired. The more elegant way to approach this would be to only *refine* the mesh in the neighborhood of the kink in the curve, which provides the same level of resolution as an equidistant fine mesh, but has an increased savings in computational cost.

3.2 General Features

The general methods used for adaptive meshing by most adaptive mesh refinement (AMR) tools are *Block-Based AMR*, *Structured Tree AMR*, *Unstructured Tetrahedral AMR* and *Unstructured Cartesian AMR* — each with its own advantages, uses and drawbacks. FARCOM produces a fully unstructured mesh in three dimensions, and

supports cell-centered, co-located solution variable storage for use with finite volume and finite element flow solvers (Ballesteros, 2015).

The unstructured Cartesian AMR approach uses hexahedral meshes, which are generally more accurate than tetrahedral meshes with the same element count. It uses quadtree (in 2D) or octree (in 3D) data structures to retain information about the entire grid in memory, which is required during any *refinement* or *coarsening* operations. The process of bisecting a given cell in each coordinate axis resulting in a set of *daughter cells* (4 in 2D, or 8 in 3D) at one level finer is called refinement, whereas the process of combining a set of cells of the same refinement level (again, 4 in 2D, or 8 in 3D) to produce one cell at one level coarser is referred to as coarsening.

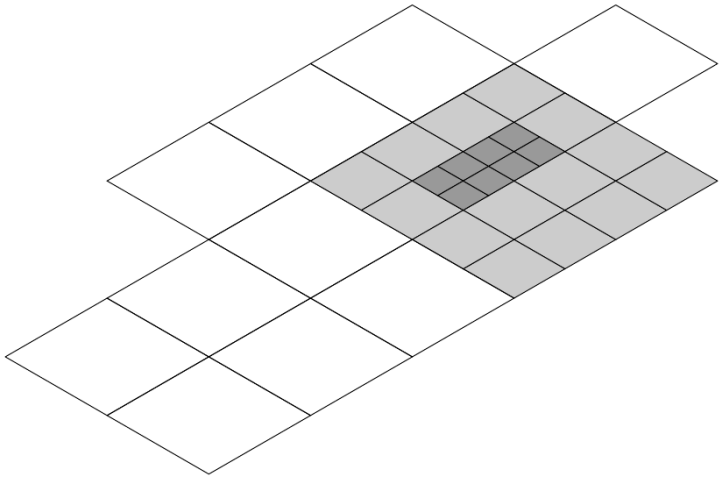


Figure 3.2: Two dimensional unstructured Cartesian AMR (Ballesteros, 2015). Intensity of the color gray indicates level of refinement.

For example, in Figure 3.2, the gray cells are a result of refining the white cells, with the difference in the level of refinement between them being 1. Four dark gray (finer) cells can be collectively coarsened to get back a gray (coarser) cell. Although FARCOM is capable of refining and coarsening cells depending on the nature of the problem on-the-fly, for the purposes of the approach presented in this thesis, only refinement operations are required. The library allows for selective refinement based

on user-defined criteria, depending on the requirement of the problem in hand. The criteria requirement for the current work is discussed in detail, in Chapter 4.

The mesh structure used in FARCOM is a hybrid compressed sparse row (CSR) data structure, where each cell in the discretized domain is treated as a control volume, containing all metadata information about the control volume itself and also the ability to be linked to cell-centered variables, which are utilized in the current work.

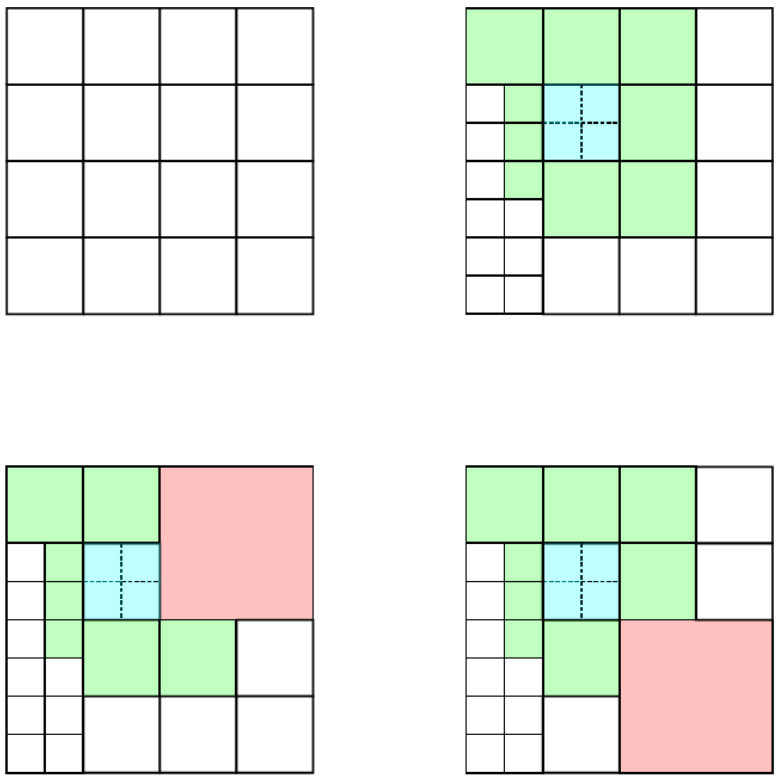


Figure 3.3: Constraints on the refinement operation (Ballesteros, 2015). Green cells do not pose any refinement restrictions; red cells block the refinement operation; cyan cells are currently being refined

Upon completion of a successful refinement or coarsening operation, FARCOM can expand or compress the solutions variable arrays' memory allocations, to suit the current mesh.

3.3 Limitations

There are a few limitations, or rather, things to bear in mind while utilizing FARCOM, and the ones that will affect the current work are discussed in this section. Firstly, a drawback of using an unstructured Cartesian AMR meshing approach is that in contrast to structured meshes, unstructured Cartesian AMR meshes require additional memory storage to store the adjacency graph of each cell. The effect of this on computational cost does not necessarily overpower the effect of having a non-AMR mesh, however, this may show up during very complex geometries when the number of triangulations in the stereolithography (STL) file is very large.

The refinement operation on any mesh cell will be successful only if:

1. cells sharing a face with a parent cell have a difference in their level of refinement by less than two.

2. cells sharing a edge with a parent cell have a difference in their level of refinement by less than two.

As seen in Figure 3.3, whenever there is a difference in level of refinement between two neighboring cells of more than one, the refinement operation fails, with the coarser cells "blocking" the operation termed as *blocking cells*. While this might result in a mesh with very smooth gradients, or otherwise a "graded" mesh, the *blocking cells* need to be taken care of, and have to be refined initially before proceeding to perform a refinement operation of the original cell in question.

Chapter 4

SYSTEMATIC NUMERICAL METHOD

4.1 Introduction

In this chapter, the various numerical methods used in the thesis are discussed in detail. Algorithms (pseudo-codes) are provided in places where it may be required to get a better understanding of the method implemented. The first section provides an overview of the steps taken in order to obtain an interconnectivity between the triangles and the underlying Cartesian mesh. The second section provides an overview of the methods in order to refine the underlying mesh to better adapt to the topology of the surface geometry. In section three, the process of obtaining the initial level set values and obtaining a full level set scalar field are discussed. The final section contains methods of smoothing and denoising the interface of the surface geometry.

4.2 Establish Triangle Information in Cartesian Grid

In this section, the triangulation information carried by the STL file will be established in the underlying Cartesian grid, while ensuring that the interconnectivity data is present. There are a couple of pre-requisites to this step: (1) the availability of a valid STL file, and (2) the availability of an STL file reader. The former is a strict pre-requisite, as the STL file needs to adhere to its file format. Additionally, there exists another restriction placed upon the triangles in the file, which is that all adjacent triangles must share two common vertices. This is a general requirement concerning the validity of the file format and not compatibility of the current work in itself. The second pre-requisite is not discussed in this thesis, as creating a file reader

for STL file formats is fairly straightforward, provided the formatting of the file (from Chapter 1) is known. Alternatively, there exists open source code in various computing languages, distributed under the GNU LGPL license (Burkardt, 2005), which can also be used.

4.2.1 Preparatory Work

Once the triangle data from the STL input file is read in by the code, it has to be stored by any appropriate means. For this work, each triangle facet was treated as a `stlface`, a Fortran derived-data type structure which contains metadata of the facet including the triangle vertex node indices which make up the triangle, referred to from now on as `nodes`; the unit normal vector of the facet, referred to as `normal`, the centroid of the facet, referred to as `center`; the control volume index in which the facet is sorted into, referred to as `cvnum`; and others, which will be introduced as necessary. Since adjacent triangles share two common vertex nodes, including them within the `stlface` structure will create an increase in memory usage. For this reason, an additional structure for storing the coordinates of each node is created; referred to as `stlnode`.

The look up between each `stlface` and its vertex node coordinates is fairly straightforward: obtain the node index number of the triangle vertex node from the `stlface` structure, and use this number to query the `stlnode` structure for the coordinates. It is highly desired to pre-compute the centroid of each facet at this point, as it will be accessed numerous times during the procedure. Additionally, a refinement length-scale for each facet is to be computed; this length-scale is equal to the minimum edge length of the facet. This facet metadata is referred to as `length_scale` in this thesis.

4.2.2 Initial Sorting

At this point, it is required to *sort* the triangle facets into the underlying Cartesian mesh, the process by which the location of each facet in the mesh is known at all times. The base mesh is started off a coarse mesh, sufficient enough to hold the macroscopic level details of the surface geometry. The *sorting* of the facets into the individual mesh cells of this coarse mesh is done by two separate means:

1. by location of the centroid of the triangle facet; and

2. by intersection of the facet with the control volume of the Cartesian mesh cell

To obtain the data for the first condition of sorting, it is required to first loop over all the triangle facets *stl*, and obtain the bounding box of the current triangle facet.

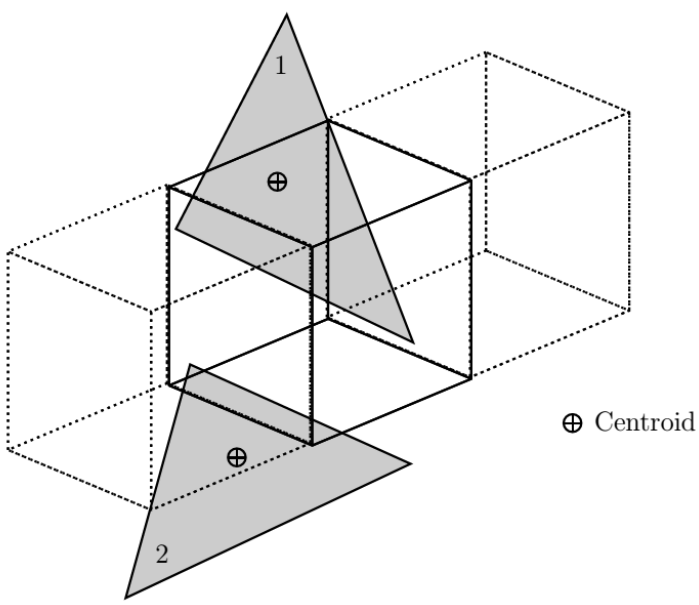


Figure 4.1: Sorting of triangulations based on location of centroid. Triangle 1 counts as being sorted and intersecting, whereas triangle 2 counts only as intersecting.

It is then required to loop over all the control volumes which intersect with this bounding box `cv`, and obtain the bounds of each control volume cell. A check is

performed to see whether the `center` of the facet lies within the bounds of the control volume mesh cell. This facet is now termed as "sorted" within the control volume mesh cell. The second condition is satisfied if the facet intersects the control volume mesh cell in any way. This can be performed by making use of an axis-aligned bounding box (AABB) and triangle intersection algorithm, such as the one by Akenine-Möller (2001), which is a fast and efficient intersection testing algorithm based on the *separating axes theorm*.

The base algorithm of the initial sorting routine is given in Algorithm 1, where `cv` and `stl` denote the total number of control volumes and total number of facets, respectively. Two new data structures are introduced here namely, `nstl` and `nstlcut`, which are structures corresponding to the current adaptive mesh. For each control volume, `nstl` holds the information of number of facets sorted by means of centroid location, in that control volume. Similarly, `nstlcut` holds the information of number of facets intersecting the control volume.

Algorithm 1 Initial stage sorting of triangle facets to underlying Cartesian mesh

```
for m = 1 → stl do                                          ▷ Loop through facets
    Obtain bounding box of m

    for n = 1 → cv do          ▷ Loop through control volumes in bounding box

        if (facet_center within cv_bounds) then
            sorted ← true                                          ▷ facet is sorted
            cvnum ← n                                  ▷ store control volume number
            nstl(n) ← nstl(n) + 1
        end if

        if (triangleAABBintersection = true) then
            nstlcut(n) ← nstlcut(n) + 1
        end if
    end for
end for
```

These two structures play an important role in selectively refining the mesh, and also help in keeping track of all the facets during refinement process, so as to ensure no triangle information is lost. Although the pseudo-code in Algorithm 1 seems trivial, additional sweeps across the mesh are required to store the triangle index number sorted in each control volume, referred to as `falim`; and triangle index numbers intersecting each control volume, referred to as `facut`. The advantages of using adaptive meshes is highlighted here once again, as using an equidistant fine Cartesian mesh would be multiple mesh sweeps with nested facet sweeps extremely computationally intensive, especially in the case of highly complex geometries with an enormously high triangulation count.

4.3 Adaptive Mesh Refinement

Upon completion of the sorting procedure, which ensures that the location of the facets within the initially coarse mesh is known, it is time to adaptively refine the mesh appropriately such that the resolution provided by the mesh near the interface is high and sufficient, while maintaining a lower resolution away from the interface. The refinement operation is handled by FARCOM, as described in Chapter 3, however, the refinement criteria are tailor-made to suit the problem in hand.

4.3.1 Refinement Selector Criteria

The question of which control volume is selected to be refined, can be answered by monitoring the triangle facet data present within the control volume. This is where the data present in the structures `nstl`, `nstlcut`, `falim`, and `facut`, come into play. There are three main refinement criteria which will determine whether a given control volume will be refined or not, they are:

1. The number of facets sorted in a control volume may not exceed a specified

limit, referred to as `max_face_limit`.

2. The number of facets intersecting a control volume may not exceed a specified limit, referred to as `max_face_cut`.

3. The computed refinement threshold must be less than a specified tolerance limit, referred to as `refine_tol`.

It has to be noted that all three specified limits: `max_face_limit`, `max_face_cut` and `refine_tol` are tuning parameters, which vary depending on the surface geometry in question. The refinement threshold is computed in the following way: consider a control volume mesh cell with equal dimensions in all three directions, Δ. The refinement threshold is then,

$$\text{threshold} = \frac{\Delta}{\texttt{length\_scale}} \tag{4.1}$$

where, `length_scale` is the refinement length scale computed for each facet, as described in section 4.2.1. The pseudo-code for the refinement selection process is given in Algorithm 2.

A note of interest here is that the refinement process is not fully cyclic, as every time a control volume mesh cell gets refined, resulting in *daughter cells*, the triangle data present in the *parent cell* is no longer valid, and there is data missing from the daughter cells. This issue can addressed by the process of *re-sorting*, which is discussed in the following section. Another point of interest while utilizing adaptive meshes is that the refinement process may be "blocked" by neighboring cells if they are relatively too coarse, as discussed in Chapter 3. This can be rectified by refining the *blocking cells* before proceeding to refine the interested control volume cell, n, which however, will result in additional re-sorting of the triangle data.

Algorithm 2 Refinement selection criteria for adaptively refining a control volume

```
Given a control volume n
if (nstl(n) > max_face_limit) then                    ▷ refinement condition #1
    Refine n
end if

if (nstlcut(n) > max_face_cut) then                   ▷ refinement condition #2
    Refine n
end if

for all stlface in n do
    lscale ← min(lscale, length_scale)                ▷ minimum length scale of all facets
end for
threshold ← Δ/lscale
if (threshold < refine_tol) then                      ▷ refinement condition #3
    Refine n
end if
```

The refinement process is complete once every mesh cell that can be refined, has indeed been refined to the limit specified by the refinement conditions.

4.3.2 Re-sorting the Triangulation Data

As discussed in the previous section, the process of refinement results in multiple cells at different levels of refinement having incorrect or missing triangulation data. This issue can be addressed by the process of re-sorting, which is identical to the initial sorting procedure, as discussed in section 4.2.2, albeit through a much more quicker process.

The re-sorting procedure is called every time a control volume mesh cell is refined; the triangle data contained within the *parent cell* is stored and sent to the re-sorting algorithm. This follows the idea that all triangles which are sorted in any arbitrary given mesh cell, will be sorted into one of its own refined *daughter cells*. A simple pseudo-code for this procedure is given in Algorithm 3.

It can be seen that the procedure is exactly identical, except for a few changes

Algorithm 3 Re-sort triangle data of control volume n into its daughter cells

```
Given a control volume n
Obtain daughter cells
Clear incorrect data
for n = 0 → dcv do                                  ▷ Loop through daughter cells
    for m = 1 → stl do                              ▷ Loop through facets from parent

        if (facet_center within cv_bounds) then
            sorted ← true                           ▷ facet is sorted
            cvnum ← n                               ▷ store control volume number
            nstl(n) ← nstl(n) + 1
        end if

        if (triangleAABBintersection = true) then
            nstlcut(n) ← nstlcut(n) + 1
        end if
    end for
end for
```

namely, before starting the sorting process, the first step is to get the control volume index numbers of all the daughter cells of the parent cell n. This should be a straightforward call to the AMR library being used. After this, the triangle information from the parent cell must be stored in memory, and all data from both the parent and its corresponding daughter cells must be erased. From here on, the procedure is the same as the initial sorting, only instead of sweeping across the entire mesh and a nested sweep across all the facets, the mesh sweep is only across the daughter cells and the facet sweep is only across the triangles from the parent cell.

4.4 Initialization of the Level Set Field

Upon completion of the adaptive mesh refinement operation, the current state of the domain should be an AMR mesh refined adaptively to capture the interface, with finer mesh cells near to the interface and relatively coarser mesh cells away from the interface. This mesh should indicate the final mesh required for the rest of the method, and can now be utilized to generate a level set field. The choice of level set

field for the purposes of this work is chosen to be a signed minimum distance level set function, as described in Chapter 2.

4.4.1 Computing Initially Accepted Level Set Values

Even with a highly desirable AMR mesh underneath the triangulation data, computing minimum distances for every control volume mesh cell to the surface geometry will be a computationally expensive task. In order to avoid this, only the control volume mesh cells in the immediate vicinity of the surface geometry will have a minimum distance computed for them. These mesh cells will be referred to as the *initially accepted* mesh cells or `ACCEPTED` in short.

A control volume mesh cell is considered to be an `ACCEPTED` mesh cell, if: (1) the mesh cell has facets intersecting through it, or (2) any one of its immediate neighbor mesh cells has a facet intersecting through it. The neighbor mesh cells to be considered are cross-face neighbors, cross-edge neighbors and cross-node neighbors, which results in 26 neighboring mesh cells. The need for checking whether a mesh cell has facets sorted into them is circumvented by the fact that all triangles which have a centroid located within a cube, will be intersecting the cube. Computing a minimum distance to the surface geometry for the `ACCEPTED` cells becomes fairly simple, once they are identified. For example, if a mesh cell satisfies condition (1) i.e., it has facets intersecting through it, then the minimum distance can be computed as shown in Algorithm 4.

The minimum distance at the mesh cell center is calculated as the minimum of the normal projection distance to the facet, the distance to the vertex nodes of the facet and the distance to the edges of the facet. The correct sign of the minimum distance value is automatically computed by the way in which the distance formulas have been written. The set of triangle facets for which the minimum distance is to be

Algorithm 4 Compute minimum distance level set for a mesh cell containing facets

```
for m = 1 → stl do                                          ▷ Loop through facets in cell
    compute unit normal vector, n̂
    proj_dist ← n̂ · (x_0 − x_i)                             ▷ point-plane distance
    get node coordinates of facet
    compute distance from cell center to node, pp_dist      ▷ point-point distance,
for each node
    compute distance from cell center to edge, pe_dist      ▷ point-line distance, for
each edge
    d(m) ← min(proj_dist, pp_dist, pe_dist)
end for
φ ← min(d)
```

computed is the set of triangle facets belonging to the current mesh cell and the set of triangle facets belonging to all of the 26 neighboring mesh cells. In contrast, if a mesh cell has been `ACCEPTED` by satisfying condition (2) i.e., one of its neighbor mesh cells has facets intersecting through them, then the above process for computing the minimum distance level set is slightly modified. The distances will be computed from the center of the `ACCEPTED` mesh cell, n to the facet(s) present in the neighboring mesh cell(s). If there are more than one neighboring cells with facets in them, then ϕ will take the minimum value of them.

4.4.2 Obtaining Full Level Set Field

Upon computing initially accepted level set values, it is desired to distribute the level set field everywhere within the computational domain. This is an effect of not computing level set values everywhere in the computational domain, which is ideal as it is less computationally intensive. A full level set field can be obtained by solving the level set reinitialization equation on the entire computational domain. This is possible due to the fact that a minimum distance level set function has the property, $|\nabla\phi| = 1$ inherently built in. Before the reinitialization equation is solved for, the sign of each mesh cell has to be set correctly. This means, as described in Chapter 2,

ϕ is positive within the surface geometry and negative outside of it.

The correct setting of the sign of the level set scalar field can be done by brute-force means, that is, every mesh cell will take the sign of its cross-face neighbor. The drawback of this method is that it requires multiple sweeps across the entire domain, which is highly undesirable. An alternative to this, is to start from the `ACCEPTED` band of mesh cells and set the signs to different "bands" of cells growing outwards in the interface normal direction, similar to the band layer growth algorithm in Herrmann (2008). This is carried out by making use of specialized "zone" data structures.

Once the correct sign has been set, the standard PDE-based reinitialization, as described in Chapter 2 can be carried out with the `ACCEPTED` level set values set as Dirichlet boundary conditions. This will ensure that the full scalar field will adjust to satisfy $|\nabla\phi| = 1$, based on the `ACCEPTED` values, which will remain "fixed" for this process. It has to be noted that the value of the level set scalar in the non `ACCEPTED` cells does not play a role in the reinitialization, only the correct sign of the value is required. In the current work, an arbitrary high value is selected and set everywhere in the non `ACCEPTED` cells.

4.5 Smoothing of the Interface

Having a full level set scalar field means that there is adequate means of describing the original surface geometry. However, the full field is an extension of the initially accepted level set values, which means for a really noisy STL file input, the initially accepted level set values mirror the noisy nature of the triangulation. This can be addressed by smoothing the surface, which now being represented by a smooth level set function can be done quite natively.

4.5.1 Excessive Reinitialization of the Level Set Field

The first and most intuitive way to approach smoothing of the level set field is by means of excessive reinitialization of the level set field. This is identical to solving the reinitialization for obtaining the full field as discussed in the previous section, however, for the purposes of smoothing the interface, the `ACCEPTED` level set values are not set as Dirichlet boundary conditions anymore. The boundary conditions at the domain boundaries can be set to Neumann boundary conditions with the inclusion of numerical ghost mesh cells.

This will allow the `ACCEPTED` level set values to change along with the level set values away from the interface, all the while trying to approach a weak, but smooth solution. The level of smoothing is controlled by the number of reinitialization iterations to be computed.

4.5.2 Mean Curvature Flow

An alternative means of smoothing the level set values near the interface is by means of performing a mean curvature flow calculation, as described in Malladi and Sethian (1995). This involves solving the level set advection equation (2.4), with the velocity term being replaced by:

$$\vec{v} = (\tilde{\kappa} - \kappa)\,\hat{n} \tag{4.2}$$

where, κ is the local mean curvature, $\tilde{\kappa}$ is the locally filtered curvature and $\hat{n}$ is the outward unit normal vector at Γ, respectively. The filter size of the locally filtered curvature should be set equal to the size of the local irregularities. For the current work, the mean value of curvature is set to zero, and hence equation (4.2) becomes $\vec{v} = -\kappa\hat{n}$. The effect of this change will be demonstrated in Chapter 5. The outward

unit normal vector the level set field is given by:

$$\hat{n} = -\frac{\nabla\phi}{|\nabla\phi|} \tag{4.3}$$

with the gradients of the level set field computed using second-order central finite-difference approximations. The curvature is given by:

$$\kappa = \nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} \tag{4.4}$$

or in other words, curvature is the divergence of the outward unit normal vector of the level set field. There are different ways of computing curvature outlined in the literature, but the standard way is to discretize equation (4.4) with finite-difference operators after analytically modifying the equation, resulting in equation (4.5). This approximation is however, a first-order approximation to the curvature of the surface geometry. This issue can be overcome by either using *height functions* (Cummins *et al.*, 2005), or by using the *direct front curvature* (Herrmann, 2008) method, as a means of estimating second-order accurate curvature.

$$\kappa = \frac{\phi_{xx}^2(\phi_y^2+\phi_z^2)+\phi_{yy}^2(\phi_x^2+\phi_z^2)+\phi_{zz}^2(\phi_x^2+\phi_y^2)}{(\phi_x^2+\phi_y^2+\phi_z^2)^{3/2}} - 2\frac{\phi_{xy}\phi_x\phi_y+\phi_{yz}\phi_y\phi_z+\phi_{xz}\phi_x\phi_z}{(\phi_x^2+\phi_y^2+\phi_z^2)^{3/2}} \tag{4.5}$$

The current work makes use of the *direct front curvature* method, as it is a direct extension of equation (4.5), and is found to produce good estimates in the region close to the interface. Although *height functions* produce good estimates as well, they is more suited for volume of fluid (VoF) methods, and requires a translation of the level set scalar field to a volume of fluid scalar field (van der Pijl *et al.*, 2005), and a possible rewrite of the method better suited for adaptive meshes.

Once the curvature and the outward unit normal vector is computed, they replace

the velocity vector in equation (2.4), by substituting in equation (4.2). This results in the mean curvature flow equation, as given below:

$$\frac{\partial \phi}{\partial t} - \kappa |\nabla \phi| = 0 \tag{4.6}$$

with the solution of ϕ valid for all $t > 0$. As described in Ley (2004), the mean curvature flow will work even if the starting level set scalar field is not regular enough, or contains singularities. This provides an almost natural way of smoothing the surface and eliminating higher frequency noise primarily, while also preserving the overall shape of the geometry, described by its curvature and outward unit normal vector.

Chapter 5

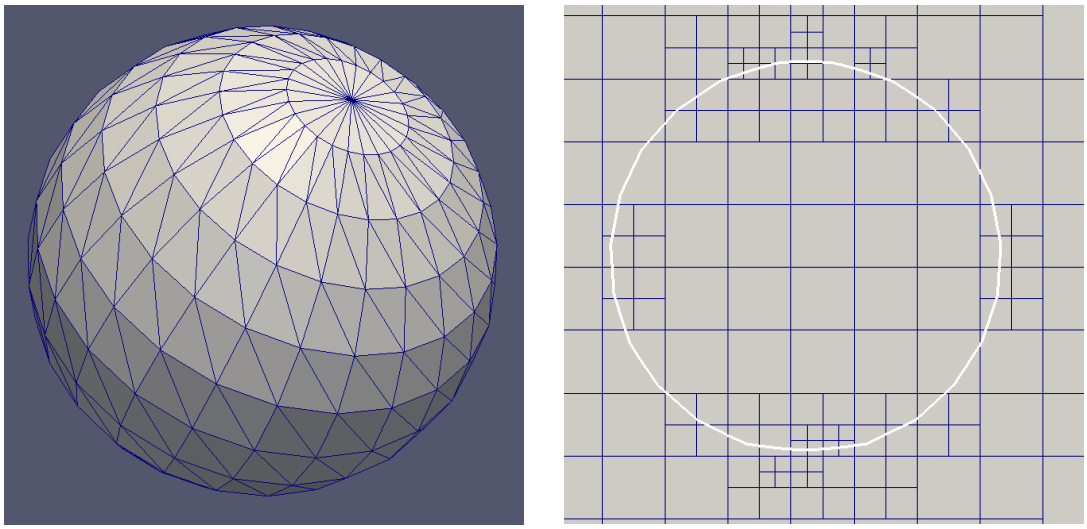
RESULTS

In this chapter, some results obtained by the previously discussed method for smoothing surface geometries are presented. Various test cases have been chosen with increasing geometric complexity, namely – a simple sphere, a vase, a twisted knot, and finally a Spray A injector nozzle geometry (#210675, Pickett (2014)). Results obtained at various points of the process have been shown to detail the capability of the presented work.

5.1 Adaptive Mesh Refinement

For all the test cases, the results of adaptive mesh refinement indicate that the refinement criteria is indeed good for the purposes of the current work. Observed results show that the mesh cells near the interface are much finer than the mesh cells further away.

The first test case to be considered is a sphere, which is the least complex geometry among the test cases, having both a simple geometry and very low triangulation count. The number of triangles contained in the STL file describing this sphere is 550. As seen in Figure 5.1, the mesh cells are relatively finer near the poles of the surface geometry, which contains the highest concentration of triangulations. As with the case of all geometries, the level of refinement can be changed as per requirement (as discussed in Chapter 4); the level of refinement is not strictly limited as shown in Figure 5.1. The total number of mesh cells at the end of the refinement operation is 1580. It should be noted that although a sphere is a symmetric shape by itself, the STL file geometry is not due to the way the triangulation is performed.



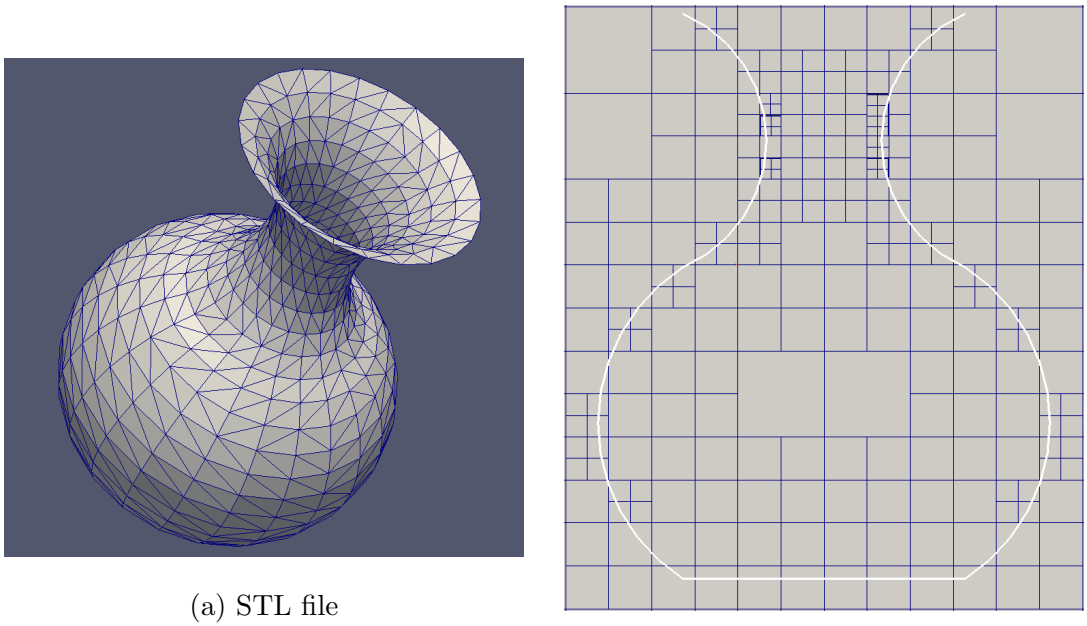
(a) STL file (b) Adaptively Refined Mesh

Figure 5.1: A simple sphere in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation

The next test case with a relatively increase in geometric complexity is a vase in three dimensions. Similar to the previous test case, even though the geometry by itself is symmetric, the STL file representation by way of surface triangulations does not show any inherent symmetry. The number of triangles contained in the STL file describing this vase is 1174. As shown in Figure 5.2, this test case has a definite increase in the triangulation count, as compared to the sphere, and also possesses a more varied shape in general. The total number of mesh cells at the end of the refinement operation is 4720. The adaptively refined mesh shown here is very indicative of the surface geometry itself with finer mesh cells near the "neck" of the vase, which has greater triangulations. The flat surface present at the bottom of the surface geometry contributes towards the generally coarser mesh in that region.

With a drastic increase in surface triangulation count, the next test case is a twisted knot structure in three dimensions. Along with greater number of triangles, this geometry also features multiple surfaces in close proximity to one another, as

shown in Figure 5.3. The number of triangles contained in the STL file describing this knot is 76428. This increase in geometric complexity was reflected on the computational time required to perform the sorting and location of triangles with the mesh, and in adaptively refining the mesh. The total number of mesh cells at the end of the refinement operation is 224360. As shown in Figure 5.3 (a), the STL file shows high concentration of triangles near the three smaller protrusions from the surface of the geometry. The resulting mesh from the process of adaptive refinement shows good agreement with the surface described in the STL file itself. Generally more finer mesh cells are found near the surface, and even more so in the regions with multiple surfaces close to one another.



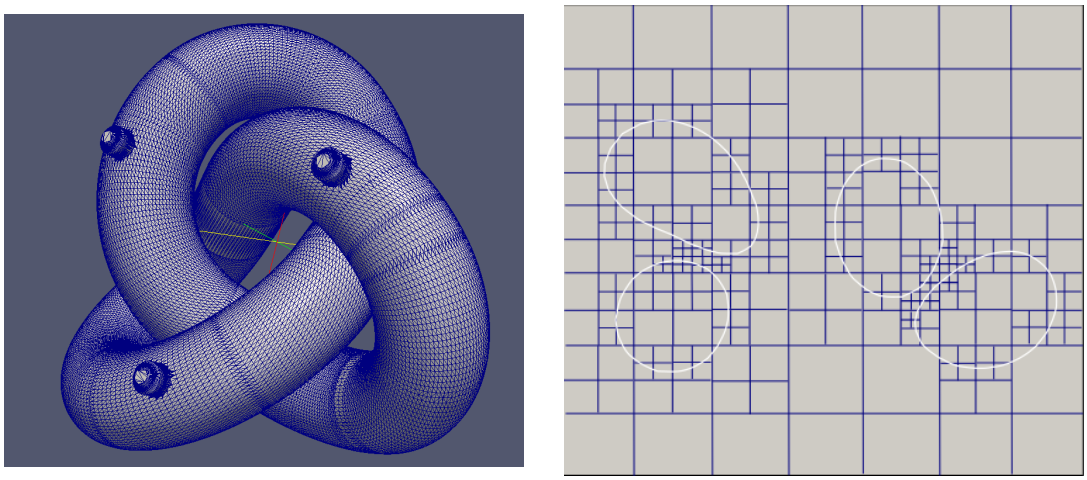
(a) STL file

(b) Adaptively Refined Mesh

Figure 5.2: A vase in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation

The final test case is a "Spray A" single-hole injector nozzle geometry, which is used in Spray A research (Pickett, 2014). This geometry comes as a result of high-resolution x-ray tomography, and is known to contain holes and other surface

irregularities. This test case contains the highest triangle count among all the test cases presented in this work, and is proven to be quite computational taxing. The number of triangles contained in the STL file describing this vase is 496658. The sorting and refinement operation on this highly detailed STL file did not complete fully, as desired due to limited computational power. The current work is written in a sequential code, thereby limiting its full potential. However, although the refinement operation did not got through fully, it can be noted that the general refinement requirements seem to be satisfied (Figure 5.4).



(a) STL file

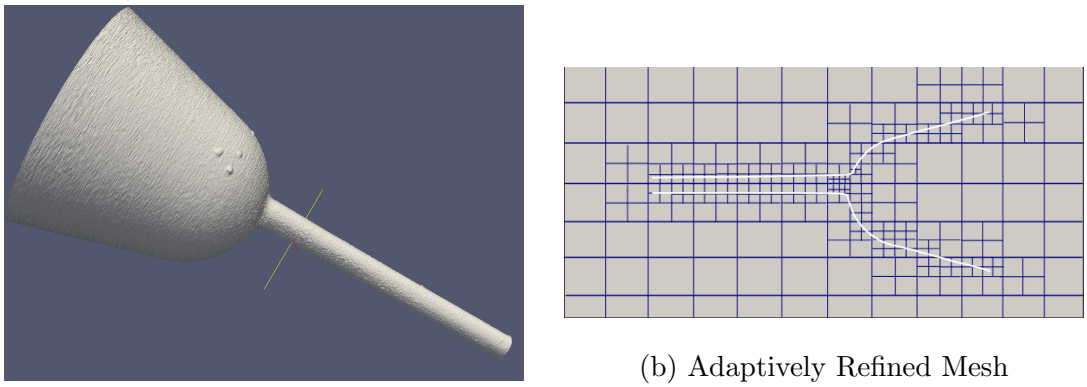
(b) Adaptively Refined Mesh

Figure 5.3: A twisted knot in three dimensions: (a) shows a snapshot of the actual STL file, (b) shows a cross-section along a plane cutting through the geometry of the adaptively refined AMR grid based on concentration of surface triangulation

This a good indication that the procedure seems to be working as expected, only limited by computational power. A means of overcoming this limiting factor would be to write the code in parallel, thereby making use of modern multi-core architecture.

5.2 Initially Accepted Level Set Values

The level set values obtained from computing the minimum distance to the surface geometry are shown in Figure 5.5. Upon close inspection of the results, it can be



(a) STL file

(b) Adaptively Refined Mesh

Figure 5.4: Spray A injector nozzle geometry: (a) shows a snapshot of the actual STL file, (b) shows a cross-section of the adaptively refined AMR grid based on concentration of surface triangulation

observed that the reconstructed interface through implicit representation of the level set scalar closely follows the geometric representation of the surface triangulations. This means that in case of test cases where the number of triangulations were not high, thereby resulting in a coarse grain geometric representation of the surface geometry, the zero level set surface also follows the same coarsened approximation. This is important so as to not artificially generate topology changes in the surface geometry. This results in a coarser approximation while making use of low quality STL files, and better approximation when using higher quality STL files with a greater triangle count.

The sphere being the more easily representable geometry of the test cases, shows good agreement between the original STL geometry file and the implicit level set representation. An initial inspection shows almost no difference in both the line representations. When it comes to the vase test case, it can be noticed that the level set follows the "coarse" representation of the triangulation almost perfectly. In other words, if the 2D projection of the vase can be represented exactly by a set of lines, the level set representation mimics it accurately. To provide a quantitative means of

measuring how accurate the representation of the original STL geometry is, the error between the curvature of the original STL file and the zero level set were computed, as shown in 5.1.

Test Case	$L_2(\mathrm{E}_\kappa)$	$L_\infty(\mathrm{E}_\kappa)$
Sphere	3.36e-3	7.44e-3
Vase	9.14e-4	9.83e-4
Knot	5.32e-3	6.16e-3

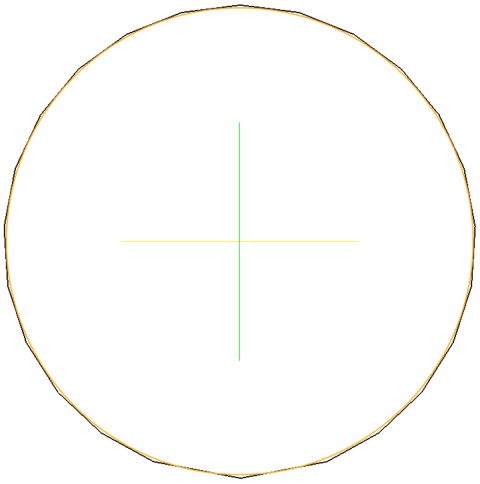
Table 5.1: Error in curvature for level set reconstructed surface geometries

The final test case is that of the twisted knot, and a cross-sectional view of the zero level set isoline is shown in Figure 5.5 (c). In this view, the 2D projection is a collection of elliptical cross-sections. The level set representation shows accurate capturing of this projection, while not introducing further undesired topology changes.

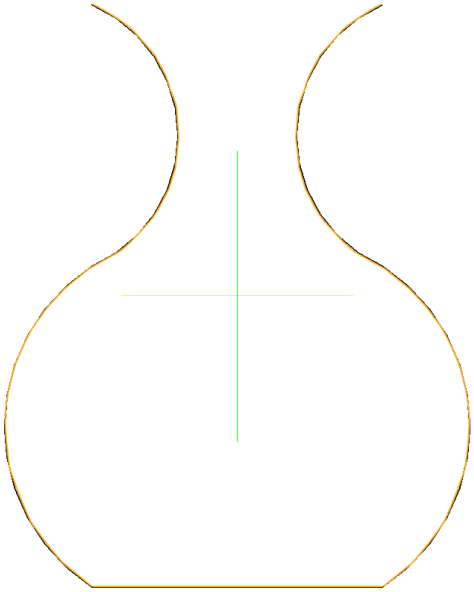
5.3 Smoothened Level Set Isosurface

Smoothing the surface can be done in one of two ways as discussed in Chapter 4: (a) by excessive reinitialization, or (b) mean curvature flow. Performing excessive reinitialization can be used to smooth the surface as the inherent numerical dissipation of the method will result in diffusing the level set scalar field, and in the process, smooth the zero level set isosurface. In addition, the reinitialization performed in this work is based on a first-order finite difference scheme, which will result in greater numerical diffusion. In order to avoid creating unwanted topology changes in the surface geometry, the mean curvature flow method is chosen and some of the results follow.

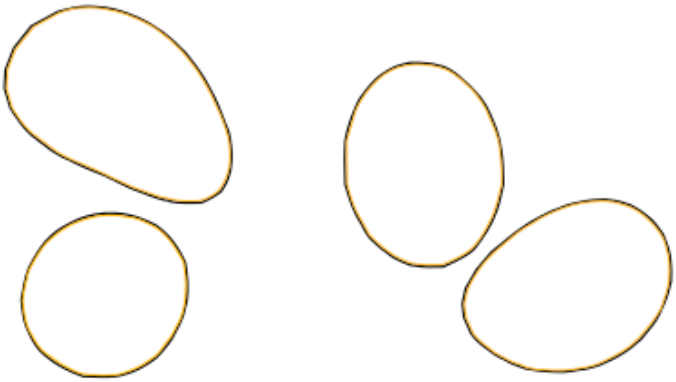
The test cases which have been discussed so far are a result of computer-aided



(a) Sphere



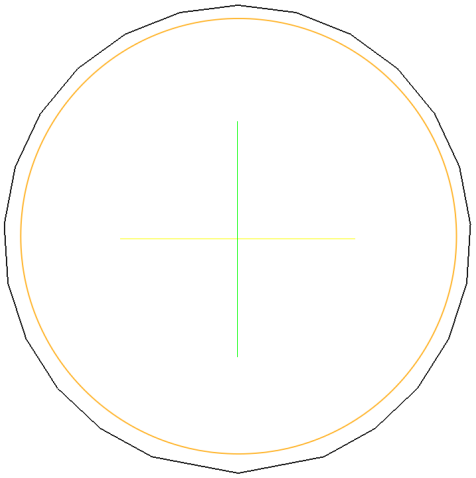
(b) Vase



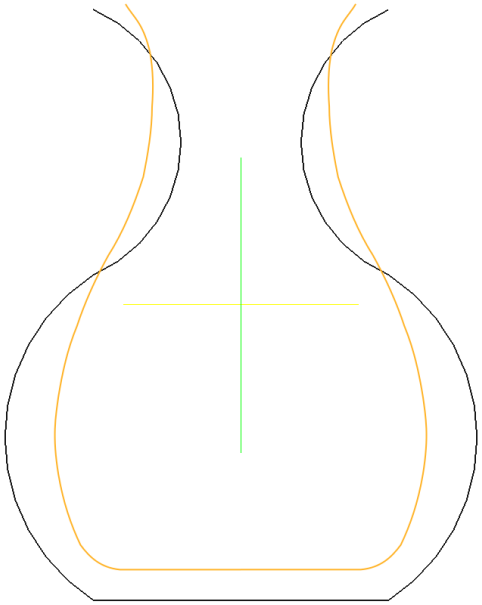
(c) Knot

Figure 5.5: Initially accepted zero level set isolines of the various test cases. The black line represents the original STL file, and the orange line represents the reconstructed level set surface

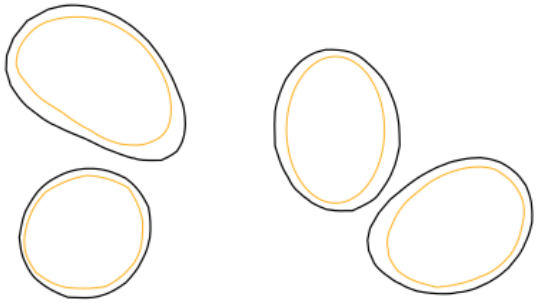
design, and thus show no signs of noise or surface irregularities. In this section, the functioning of mean curvature flow is demonstrated and the application of the same is discussed. It has been shown that under mean curvature flow (Ley, 2004), a surface with regions of high curvature will tend to shrink towards a finite point. This means that as seen in Figure 5.6, a sphere will shrink towards its center with time. This general behavior under mean curvature flow can be observed in other test cases also, wherein regions of high curvature undergo changes in the level set scalar field resulting in regions of lower curvature. It has to be noted that the results shown in Figure 5.6 are the result of excessive smoothing, forcing the drastic changes as shown. The idea to be transferred to noisy surfaces is that, surface irregularities can be treated as regions of local high curvature which can be treated in the same way as shown here, by performing controlled smoothing, thereby ensuring minimal topology change to the overall surface geometry.



(a) Sphere



(b) Vase



(c) Knot

Figure 5.6: Cross-sectional snapshots of excessive smoothing on the various test cases. The black line represents the original STL file, and the orange line represents the reconstructed level set surface

Chapter 6

CONCLUSION

A general approach for smoothing and denoising complex surface geometries using a level set method was demonstrated in this work. The resulting surface geometry had all the characteristics of the input surface geometry but showed the ability to smooth the surface of the geometry to various levels as desired. The use of level sets helped maintain the smoothness of the function, thereby providing a natural means of getting rid of any rough topology in the surface geometry. It was shown that the capabilities of the presented method can be improved to suit real world applications by implementing a parallel version of the existing code.

The presented method showcases a general method for not only smoothing the input geometry, but also providing a means of describing the surface geometry through an implicit representation, which can be extracted and used as an arbitrary input condition for complex two-phase flow simulations.

Chapter 7

FUTURE WORK

The need for parallelizing the existing code infrastructure was briefly discussed in Chapter 5. Future work will focus on making the code be able to run on parallel computing architecture, in order to make the presented approach viable for extremely complex geometries, which may arise from real-world applications.

Further work needs to be done, in order to extract the level set isosurface in a more elegant way. The proposed way to do that would be to incorporate a marching tetrahedrons algorithm to the existing code infrastructure, which will create tetrahedral cuts to the hexahedral mesh, while also extracting the isosurface. Including this algorithm will require the addition of a more robust interpolation algorithm to get a level set scalar value at the mesh cell vertex nodes, as opposed to the present cell-centered values.

With the tetrahedral cuts in place, a connectivity between the underlying hexahedral mesh to the tetrahedral cuts can be made by means of adding pyramid elements. This will ensure complete connectivity to the interface, and can be exported as a mesh file which can be used in fluid flow applications.

REFERENCES

Akenine-Möller, T., "Fast 3D Triangle-Box Overlap Testing", Tech. rep., Chalmers Institute of Technology, department of Computer Engineering (2001).

Ballesteros, C., *An Adaptive Mesh Refinement Library Using a Quadrature-Free Runge-Kutta Discontinuous Galerkin Method*, Doctoral dissertation, in preparation, Arizona State University (2015).

Bourke, P., "STL format", `http://paulbourke.net/dataformats/stl/` (1999).

Burkardt, J., "STLA IO - Read and Write Routines", `http://people.sc.fsu.edu/ jburkardt/f_src/stla_io/stla_io.html` (2005).

Calder, A. C., B. C. Curtis, L. J. Dursi, B. Fryxell, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Turan, M. Zingale and G. Henry, "High Performance Reactive Fluid Flow Simulations Using Adaptive Mesh Refinement on Thousands of Processors", in "Proceedings of the 2000 ACM/IEEE Conference on Supercomputing", SC '00 (IEEE Computer Society, Washington, DC, USA, 2000).

Cummins, S. J., M. M. Francois and D. B. Kothe, "Estimating curvature from volume fractions", Comput. Struct. **83**, 425–434 (2005).

Furlong, T. and C. Genzale, "Educated Spray A Geometry", Engine Combustion Network (2012).

Herrmann, M., "Two-phase flow", CTR Summer Program, Center for Turbulence Research, Stanford University (2006).

Herrmann, M., "A balanced force refined level set grid method for two-phase flows on unstructured flow solver grids", J. Comput. Phys. **227**, 2674–2706 (2008).

Jiang, G.-S. and D. Peng, "Weighted ENO Schemes for Hamilton-Jacobi Equations", SIAM J. Sci. Comput. **21**, 6 (2000).

Ley, O., "Motion by mean curvature and level-set approach", `http://ley.perso.math.cnrs.fr/ley-muroran04.pdf` (2004).

Lörstad, D., M. Francois, W. Shyy and L. Fuchs, "Assessment of volume of fluid and immersed boundary methods for droplet computations", International Journal for Numerical Methods in Fluids **46**, 2, 109–125 (2004).

Malladi, R. and J. A. Sethian, "Level Set Methods for Curvature Flow, Image Enhancement, and Shape Recovery in Medical Images", in "Proc. of Conf. on Visualization and Mathematics", pp. 329–345 (Springer-Verlag, Heidelberg, Germany, 1995).

Olsson, E. and G. Kreiss, "A conservative level set method for two phase flow", J. Comput. Phys. **210**, 1, 225–246 (2005).

Osher, S. and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Applied Mathematical Sciences (Springer, 2002), 2003 edn.

Osher, S. and J. A. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations", J. Comput. Phys. **79**, 1, 12–49 (1988).

Pickett, L., "Spray A Injector Nozzle Geometry", `http://www.sandia.gov/ecn/cvdata/targetCondition/injectorNozGeom.php`, Sandia National Laboratories (2014).

Scardovelli, R. and S. Zaleski, "Direct Numerical Simulation of Free-Surface and Interfacial Flow", Annu. Rev. Fluid Mech. **31**, 1, 567–603 (1999).

Sethian, J. A., "Fast Marching Methods", SIAM Review **41**, 2 (1999).

Shu, C.-W., "Total-Variation-Diminishing Time Discretizations", SIAM J. Sci. Stat. Comput. **9**, 6, 1073–1084 (1988).

Sussman, M., P. Smereka and S. Osher, "A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow", J. Comput. Phys. **114**, 146–159 (1994).

Tryggvason, G., B. Bunner, A. Esmaeeli, D. Juric, N. Al-Rawahi, W. Tauber, J. Han, S. Nas and Y.-J. Jan, "A Front-Tracking Method for the Computations of Multiphase Flow", J. Comput. Phys. **169**, 2, 708–759 (2001).

van der Pijl, S. P., A. Segal, C. Vuik and P. Wesseling, "A mass-conserving Level-Set method for modelling of multi-phase flows", International Journal for Numerical Methods in Fluids **47**, 4, 339–361 (2005).

APPENDIX A

TRIANGLE-AABB INTERSECTION ALGORITHM FORTRAN CODE

The fast overlap algorithm for detecting an intersection between a triangle and an axis-aligned bounding box (AABB) is given by Akenine-Möller (2001). A Fortran version based on the original version by Akenine-Möller (2001) is presented below:

```
! ######################################################################### !
! ######################################################################### !
! This function implements a fast plane/AABB overlap algorithm, which only
! tests the two diagonal vertices, whose direction is most closely aligned
! to the normal of the triangle.
!
! Original Algorithm in C by:
!               Tomas Akenine-Moller
!               Department of Computer Engineering,
!               Chalmers University of Technology
!               http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/
!
! This function returns a logical depending on whether the plane overlaps
! the box or not.
function planeBoxOverlap(normal, vert, maxbox) result(res)
  implicit none
  ! Arguments
  real(WP), intent(in) :: normal(3)        ! Normal vector of plane
  real(WP), intent(in) :: vert(3)          ! Coordinates of vertex 1 of triangle
                                           ! (point in plane)
  real(WP), intent(in) :: maxbox(3)        ! = (dx,dy,dz)*0.5 of box
  logical :: res
  ! Other variables
  real(WP) :: vmin(3), vmax(3), v
  integer(IP) :: i

  res = .false.

  do i = xdir, zdir
     v = vert(i)

     if (normal(i) > 0.0_WP) then
        vmin(i) = -maxbox(i) - v
        vmax(i) =  maxbox(i) - v
     else
        vmin(i) =  maxbox(i) - v
        vmax(i) = -maxbox(i) - v
     end if
  end do

  if (dot_product2(normal, vmin) > 0.0_WP) then
     ! Box and plane DO NOT overlap
     res = .false.
     return
  end if

  if (dot_product2(normal, vmax) >= 0.0_WP) then
```

```
      ! Box and plane overlap!
      res = .true.
      return
   end if

   return
end function planeBoxOverlap

! ########################################################################## !
! ########################################################################## !
! Perform one of the nine axis test types as specified by _testtype_.
!
! Original Algorithm in C by:
!               Tomas Akenine-Moller
!               Department of Computer Engineering,
!               Chalmers University of Technology
!               http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/
!
! The function returns a 1 if the test is passed (overlap) or a 0 if it
! failed (no overlap).
subroutine AABB_axistest(nAABB1, nAABB2, fa, fb, testtype, rr)
   ! Arguments
   real(WP), intent(in) :: nAABB1                 ! Normal #1 of AABB
   real(WP), intent(in) :: nAABB2                 ! Normal #2 of AABB
   real(WP), intent(in) :: fa                     ! Abs. triangle edge #1
   real(WP), intent(in) :: fb                     ! Abs. triangle edge #2
   character(len=*), intent(in) :: testtype       ! Test case switcher
   integer(IP), intent(out) :: rr                 ! Result of operation
   ! Other variables
   real(WP) :: rad
   real(WP) :: vmax
   real(WP) :: vmin
   real(WP) :: p1, p2, p3

   select case(testtype)
   ! ================ X-tests ====================
   case("X12")
      ! Project triangle vertices onto AABB normals
      p1 = (nAABB1*v1(ydir)) - (nAABB2*v1(zdir))
      p3 = (nAABB1*v3(ydir)) - (nAABB2*v3(zdir))

      ! Get min, max values
      if (p1 < p3) then
         vmin = p1
         vmax = p3
      else
         vmin = p3
         vmax = p1
      end if
```

```
   ! Compute "radius" of box projected
   rad = (fa * bhs(ydir)) + (fb * bhs(zdir))

   if ((vmin > rad) .or. (vmax < -rad)) then
      ! No overlap
      rr = 0
      return
   end if

case("X3")
   ! Project triangle vertices onto AABB normals
   p1 = (nAABB1*v1(ydir)) - (nAABB2*v1(zdir))
   p2 = (nAABB1*v2(ydir)) - (nAABB2*v2(zdir))

   ! Get min, max values
   if (p1 < p2) then
      vmin = p1
      vmax = p2
   else
      vmin = p2
      vmax = p1
   end if

   ! Compute "radius" of box projected
   rad = (fa * bhs(ydir)) + (fb * bhs(zdir))

   if ((vmin > rad) .or. (vmax < -rad)) then
      ! No overlap
      rr = 0
      return
   end if

! ================ Y-tests ====================
case("Y13")
   ! Project triangle vertices onto AABB normals
   p1 = (-nAABB1*v1(xdir)) + (nAABB2*v1(zdir))
   p3 = (-nAABB1*v3(xdir)) + (nAABB2*v3(zdir))

   ! Get min, max values
   if (p1 < p3) then
      vmin = p1
      vmax = p3
   else
      vmin = p3
      vmax = p1
   end if

   ! Compute "radius" of box projected
   rad = (fa * bhs(xdir)) + (fb * bhs(zdir))
```

```
   if ((vmin > rad) .or. (vmax < -rad)) then
      ! No overlap
      rr = 0
      return
   end if

case("Y2")
   ! Project triangle vertices onto AABB normals
   p1 = (-nAABB1*v1(xdir)) + (nAABB2*v1(zdir))
   p2 = (-nAABB1*v2(xdir)) + (nAABB2*v2(zdir))

   ! Get min, max values
   if (p1 < p2) then
      vmin = p1
      vmax = p2
   else
      vmin = p2
      vmax = p1
   end if

   ! Compute "radius" of box projected
   rad = (fa * bhs(xdir)) + (fb * bhs(zdir))

   if ((vmin > rad) .or. (vmax < -rad)) then
      ! No overlap
      rr = 0
      return
   end if

! ================ Z-tests ====================
case("Z23")
   ! Project triangle vertices onto AABB normals
   p2 = (nAABB1*v2(xdir)) - (nAABB2*v2(ydir))
   p3 = (nAABB1*v3(xdir)) - (nAABB2*v3(ydir))

   ! Get min, max values
   if (p3 < p2) then
      vmin = p3
      vmax = p2
   else
      vmin = p2
      vmax = p3
   end if

   ! Compute "radius" of box projected
   rad = (fa * bhs(xdir)) + (fb * bhs(ydir))

   if ((vmin > rad) .or. (vmax < -rad)) then
      ! No overlap
      rr = 0
```

```
         return
      end if

   case("Z1")
      ! Project triangle vertices onto AABB normals
      p1 = (nAABB1*v1(xdir)) - (nAABB2*v1(ydir))
      p2 = (nAABB1*v2(xdir)) - (nAABB2*v2(ydir))

      ! Get min, max values
      if (p1 < p2) then
         vmin = p1
         vmax = p2
      else
         vmin = p2
         vmax = p1
      end if

      ! Compute "radius" of box projected
      rad = (fa * bhs(xdir)) + (fb * bhs(ydir))

      if ((vmin > rad) .or. (vmax < -rad)) then
         ! No overlap
         rr = 0
         return
      end if

   case default
      call farcomError("stl_toolbox_m::AABB_axistest", &
         "Invalid _testtype_ value '"//testtype//"' requested.")
   end select
end subroutine AABB_axistest

! ########################################################################## !
! ########################################################################## !
! Use separating axis theorm to test overlap between triangle and box
! Need to test for overlap in these directions:
!     1. The {x,y,z}-directions (actually, since we use the AABB of the triangle,
!        we do not even need to test these -- but still having it)
!     2. Normal of the triangle
!     3. cross_product(edge from tri, {x,y,z}-direction).
!        This gives 3x3=9 more tests.
!
! Original Algorithm in C by:
!               Tomas Akenine-Moller
!               Department of Computer Engineering,
!               Chalmers University of Technology
!               http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/
!
! If all tests pass, i.e., there is no separating axis, then the triangle overlaps
! the box. The function returns a logical based on this condition.
```

```
function triBoxOverlap(boxcenter, boxhalfsize, triverts) result(res)
  implicit none
  ! Arguments
  real(WP), intent(in) :: boxcenter(3)        ! Position vector of center of box
  real(WP), intent(in) :: boxhalfsize(3)      ! (dx,dy,dz)*0.5 of box
  real(WP), intent(in) :: triverts(3,3)       ! Coordinates of triangle vertices
  logical :: res                              ! Result of operation
  ! Other variables
  real(WP) :: fex, fey,fez
  real(WP) :: normal(3), e1(3), e2(3), e3(3)
  integer(IP) :: intres

  intres = 3                                ! Initialize as something

  ! Copy over stuff
  bhs(:) = boxhalfsize(:)

  ! Move triangle over so that the boxcenter is in (0,0,0)
  v1(:) = triverts(1,:) - boxcenter(:)
  v2(:) = triverts(2,:) - boxcenter(:)
  v3(:) = triverts(3,:) - boxcenter(:)

  ! Compute triangle edges
  e1(:) = v2(:) - v1(:)
  e2(:) = v3(:) - v2(:)
  e3(:) = v1(:) - v3(:)

  ! ---------------------------------------------------------------
  ! Perform the axis tests to check for overlap
  ! [9 tests in total]
  fex = abs(e1(xdir))
  fey = abs(e1(ydir))
  fez = abs(e1(zdir))
  call AABB_axistest(e1(zdir), e1(ydir), fez, fey, "X12", intres)

  if (intres == 0) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  call AABB_axistest(e1(zdir), e1(xdir), fez, fex, "Y13", intres)

  if (intres == 0) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  call AABB_axistest(e1(ydir), e1(xdir), fey, fex, "Z23", intres)

  if (intres == 0) then
```

```
   res = .false.                  ! Does not overlap, quit
   return
end if

fex = abs(e2(xdir))
fey = abs(e2(ydir))
fez = abs(e2(zdir))
call AABB_axistest(e2(zdir), e2(ydir), fez, fey, "X12", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
end if

call AABB_axistest(e2(zdir), e2(xdir), fez, fex, "Y13", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
end if

call AABB_axistest(e2(ydir), e2(xdir), fey, fex, "Z1", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
end if

fex = abs(e3(xdir))
fey = abs(e3(ydir))
fez = abs(e3(zdir))
call AABB_axistest(e3(zdir), e3(ydir), fez, fey, "X3", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
end if

call AABB_axistest(e3(zdir), e3(xdir), fez, fex, "Y2", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
end if

call AABB_axistest(e3(ydir), e3(xdir), fey, fex, "Z23", intres)

if (intres == 0) then
   res = .false.                  ! Does not overlap, quit
   return
```

```
  end if

  ! ---------------------------------------------------------------
  ! Test overlap in the {x,y,z}-directions. Find min, max of the
  ! triangle each direction, and test for overlap in that direction
  ! -- this is equivalent to testing a minimal AABB around the
  ! triangle against the AABB.
  ! [3 tests in total]

  ! Test in x-direction
  if ((minval((/ v1(xdir), v2(xdir), v3(xdir) /)) > boxhalfsize(xdir)) .or. &
       (maxval((/ v1(xdir), v2(xdir), v3(xdir) /)) <  -boxhalfsize(xdir))) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  ! Test in y-direction
  if ((minval((/ v1(ydir), v2(ydir), v3(ydir) /)) > boxhalfsize(ydir)) .or. &
       (maxval((/ v1(ydir), v2(ydir), v3(ydir) /)) < -boxhalfsize(ydir))) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  ! Test in z-direction
  if ((minval((/ v1(zdir), v2(zdir), v3(zdir) /)) > boxhalfsize(zdir)) .or. &
       (maxval((/ v1(zdir), v2(zdir), v3(zdir) /)) < -boxhalfsize(zdir))) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  ! ---------------------------------------------------------------
  ! Test if the box intersects the plane of the triangle
  ! Compute plane equation of triangle: normal*x + d = 0
  ! [1 test in total]
  normal(:) = cross_product(e1, e2)

  if (.not. planeBoxOverlap(normal, v1, boxhalfsize)) then
     res = .false.                  ! Does not overlap, quit
     return
  end if

  res = .true.                      ! Box and triangle overlaps
  return
end function triBoxOverlap

! ########################################################################## !
! ########################################################################## !
```

The above snippet of code was used throughout the presented work, in detecting if a triangle and a mesh cell intersected or not.