Solving the Cartesian Cut-cell Interpolation Problem with a Tetrahedral Mesh

by

Adam Theodore Hendriks

B. Eng, Carleton University (1999)

Submitted to the Department of Aeronautics and Astronautics in partial fulfillment of the requirements for the degree of Master of Science in Aeronautics and Astronautics at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Massachusetts Institute of Technology, MMI. All rights reserved.
Solving the Cartesian Cut-cell Interpolation Problem with a Tetrahedral Mesh

by

Adam Theodore Hendriks

Submitted to the Department of Aeronautics and Astronautics on May 25, 2001, in partial fulfillment of the requirements for the degree of Master of Science in Aeronautics and Astronautics

Abstract

The algorithms for the development of a tetrahedral mesh which can be used to interpolate the data for Cartesian cut-cells are presented. Current visualization techniques for Cartesian based solutions are inaccurate. They rely on the data from the invalid nodes of the cells which cut the body of interest. A tetrahedral mesh connected solely to the valid cell nodes and the nodes inserted along the body’s surface provides a fitting structure for accurate interpolation. Two steps are required to understand and construct the required algorithms. First, an algorithm is developed to apply the basic concepts on a simple 2D cut-cell with a variable surface discretization. Once completed, the full 3D algorithm which generates the tetrahedra is assembled. Special attention is placed on node and tetrahedron removal techniques, with the goal of improving the mesh quality. Finally, the tetrahedra generation algorithms are tested on a body of complex geometry. Additional examples demonstrate the proper interpolation using the tetrahedral mesh.

Thesis Supervisor: Robert Haimes
Title: Principal Research Engineer
Acknowledgments

My past two years at MIT have been two of the best of my life. Not only because of the exciting research that I was able to work on but also because of all the great people I met and the community I was lucky to be a part of. It is these people that made every new day seem better than the last one. While everyone contributed in their own special way, I would like to give special thanks to those who helped me with my work.

First and foremost, I would like to thank my advisor, Bob Haimes. It took me a while to get used to you, but in the end, I wouldn’t want to work with anyone else. Thank you so much for everything. You were always there to help me and made me feel more like a partner than a student. They should change your name to “The Amazing” Bob Haimes.

The Fluid Dynamics Research Laboratory is the place to be. Lots of thanks to Jean Sofronas and David Darmofal for being there for me when I needed you. How could I forget the crazy CFD kids that I hung out with while I was there. You guys did it all. Long live Bertucci’s chicken and prosciutto calzone.

I would also like to thank Mike Aftosmis from Nasa Ames Research Center. You hooked me up with the pictures and the information needed to get the job done. Thanks for the pig.

Finally I would like to thank all of the members of my large family, including my dog and cat. As much as I have loved it here, my best memories are of the times when you came to visit or when I went home. As long as I have you guys, I’ll be the happiest man alive.

This work was partially sponsored by Sandia National Laboratories contract #BE-8246 with Timothy J. Bartel as technical monitor and NASA Ames Research Center contract #NAG2-1458 with Michael J. Aftosmis as technical monitor.
# Contents

Abstract 3

Acknowledgments 5

1 Introduction 15

1.1 Background ...................................................... 16
    1.1.1 Current Grid Generation Techniques ................. 16
    1.1.2 History of Cartesian Grids .............................. 16
1.2 Overview ............................................................ 17

2 Cartesian Grids and Visualization 19

2.1 Theory of Cartesian Grids ........................................ 19
    2.1.1 Volume Mesh Generation ................................. 19
    2.1.2 Cut-cell Intersection ................................... 23
    2.1.3 Examples of Complex Geometry ......................... 25
    2.1.4 Solutions and Boundary Conditions ................. 26
2.2 Background of Visualization .................................... 28
2.3 Visualization Complications ..................................... 29

3 Development of the Tetrahedral Mesh 33

3.1 Proposed Solution ............................................... 34
    3.1.1 Addition of Nodes ....................................... 35
    3.1.2 Additional Goals ......................................... 35
3.2 Two-Dimensional Triangle Generation ......................... 36
List of Figures

2-1 Cartesian cell block .............................................. 20
2-2 Intersection of a body with 2D Cartesian cells .............. 20
2-3 Cell refinement (2D and 3D) ................................... 21
2-4 Angle variation within cut-cells ............................... 22
2-5 Ray casting and winding number ............................... 23
2-6 Polygon clipping ................................................. 24
2-7 Polygon clipping: Case 1 ....................................... 24
2-8 Polygon clipping: Case 2 ....................................... 24
2-9 Polygon clipping: Case 3 ....................................... 24
2-10 Polygon clipping: Case 4 ...................................... 25
2-11 Intersection of cells with thin geometry ................. 25
2-12 Example of a Cartesian mesh with complex geometry ... 26
2-13 Intersection of a simple body with 2D cells .............. 30
2-14 Incorrect visualization of a linear scalar field .......... 30
2-15 Incorrect visualization of a spiraling vector field .... 30
2-16 Intersection of a complex body with 2D cells .......... 31
3-1 Edge insertion 2D cutting primitive .......................... 37
3-2 Triangle insertion 2D cutting primitive .................... 38
3-3 2D triangle edge swapping .................................... 40
3-4 2D example: Discretization .................................. 40
3-5 2D example: Step 1 ............................................. 41
3-6 2D example: Step 2 ............................................. 41
3-7 2D example: Step 3 ................................................. 41
3-8 2D example: Step 4.1 ............................................. 42
3-9 2D example: Step 4.3 ............................................. 42
3-10 2D example: Step 5 ............................................... 43
3-11 2D example: Step 6 ............................................... 43
3-12 Edge insertion 3D cutting primitive ............................. 46
3-13 Face insertion 3D cutting primitive .............................. 47
3-14 Tetrahedron insertion 3D cutting primitive ..................... 47
3-15 3D tetrahedron edge swapping .................................. 51
3-16 Removal of interior nodes ....................................... 56
3-17 Removal of edge nodes ......................................... 56
3-18 Inappropriate 2D triangulation algorithm ....................... 57
3-19 Triangle fanning and stitching operations ...................... 58
3-20 2D triangle generation: Major node identification .......... 58
3-21 2D triangle generation: Concave loop detection ............ 59
3-22 2D triangle generation: Triangle chopping ................... 59
3-23 2D triangle generation: Convex contour ....................... 60
3-24 2D triangle generation: Stitching of convex contour .......... 60
3-25 2D triangle generation: Completed triangulation ........... 61

4-1 Discretized pig .................................................... 64
4-2 Discretized cylinders .............................................. 64
4-3 Discretized pig in a Cartesian mesh ............................ 64
4-4 Cut-cell intersection with the discretized body .............. 66
4-5 Close-up of cut-cell ............................................. 66
4-6 Surface definition within the cut-cell .......................... 67
4-7 Nodal distribution chart ......................................... 68
4-8 Node removal chart ............................................... 69
4-9 Particular cut-cell before node removal ......................... 70
4-10 Particular cut-cell after node removal ......................... 70
4-11 Incorrect interpolation of a scalar field ........................................ 72
4-12 Correct interpolation of a scalar field ........................................ 72
4-13 Incorrect interpolation of a vector field .................................... 73
4-14 Correct interpolation of a vector field ...................................... 73
List of Tables

3.1 Performance of the recursive algorithm .......................... 54

4.1 Mesh reductions due to node removal. ........................... 68

4.2 Mesh reductions for the particular cell due to node removal. .... 71
Chapter 1

Introduction

Computational Fluid Dynamics (CFD) has become a powerful tool in aerodynamic analysis, and many designs can now be tested computationally, without experimentation. Although computer technology seems to be ever increasing and solution algorithms become more and more efficient and accurate, CFD use is still not an integral part of aerodynamic design. Part of the problem lies in the large amounts of time required to generate the grids used in the solution, especially in cases with complex geometry. Recent research on the use of Cartesian approaches for inviscid simulations has moved towards a grid generation process that is automated. While the construction of Cartesian grids is much simpler, the specification of the boundary conditions becomes the focus.

Of equal importance in the design process is the visualization of the results produced by the simulation. Obviously if the data cannot be read properly, then the results are not useful. Most scientific visualization algorithms employ interpolation, but given the nature of Cartesian grids and its treatment of the boundaries, it is difficult to build an interpolant based on the data produced. The goal of this research is to develop an interpolation scheme to accurately visualize this data. This thesis will present the steps taken to build this scheme.
1.1 Background

1.1.1 Current Grid Generation Techniques

There are three major approaches currently being used to generate grids for complex geometry. The first is the use of a structured, body-fitted grid, whose cell faces follow the boundary of the complex geometry in question. These grids allow the use of conventional solution schemes and are easy to compute. Their downfall is that significant time must be spent developing the grids. Another approach is unstructured grids, which uses triangles (in 2D) or tetrahedra (in 3D) to fill the volume around the body. While it could be an automated process, initial surface triangulations must be provided and herein lies a significant amount of work. The third approach is non body-fitted Cartesian grids. This approach consists of uniformly shaped and sized hexahedral cells which are refined near the geometry boundaries. The shape of the geometry is cut out of these cells. The problem with this method lies in the complexity of the boundaries and hence the accuracy of the solutions. Questions also remain on whether full viscous Navier-Stokes simulations can be applied to these grids.

1.1.2 History of Cartesian Grids

Assessment of flow solutions using non body-fitted Cartesian grids began in the late 1970's. Reyhner first looked at the solution of axisymmetric transonic potential flows around inlets in 1976 [33]. Purvis and Burkhalter continued along these lines with solutions of the full potential equation in 2D [32]. Inclusion of the Euler equations were introduced in 1985 by Clarke, Hassan, and Salas [15]. By the end of the 1980's, work by Grossman and Whitaker [22] and Gaffney, Hassan, and Salas [20] led to the first three dimensional inviscid solutions of the Euler equations.

Research on Cartesian approaches waned in the late 1980's, however, as great progress was observed in the development of body-fitted and tetrahedral grids. Work on industrial codes was still being conducted and as a result, Boeing's TRANAIR code, the first successful three dimensional Cartesian approach to solving the full
potential equation, was developed [34]. Also developed was MGAERO, a commercial package which solves the Euler equations [37]. Both of these codes were valuable in gaining experience with surface modeling and large-scale computations.

Until this point, not much work had been done on Cartesian cell refinement, nor on the inaccuracy of the boundary conditions. Several reports by Leveque, Berger, Powell, and others analyzing various boundary condition types as well as adaptive mesh refinement (AMR) produced considerable results [26, 8, 7, 17, 9, 16]. Melton, Enomoto, and Berger produced results for procedures that combine Cartesian Euler approaches with computer aided design/modeling (CAD/CAM) compatible geometry [30]. These results led towards the elimination of the necessity of a surface grid, an important feature which in turn led to the automation of the grid generation process.

Grid generation and flow solving is still a fairly specialized field and significant training is required to become proficient in its application. Furthermore, large amounts of time are spent constructing grids even before solving can begin. For these reasons, it is difficult for designers and engineers to implement CFD techniques in the design process. As a result, the focus has turned towards automation, speed, and robustness and there is a renewed interest in Cartesian methods. Recent work by Melton, Berger, Aftosmis et. al. aimed at reducing the necessary interaction with the entire simulation while maintaining accuracy and robustness [29, 28, 5, 2, 4, 3] has led to the development of Cart3D. It is a highly-automated package which quickly acquires the geometry and generates a grid.

1.2 Overview

First, Chapter 2 presents the theory behind Cartesian grids, including the data structures, the process for creating the cut-cells, and the theory behind the solutions. The basic concepts of visualization methods will be presented. Also, the complications of using these traditional methods with Cartesian grids and hence the motivation for this research is described.

Chapter 3 outlines the concepts of the proposed interpolation scheme to solve the
problem. Constraints and additional goals that will drive the development of the solution are introduced. The solution was developed first for a 2D triangle generator and then a 3D tetrahedra generator. For both techniques the approach, structure of the operations, and algorithms are defined. Construction of a robust 2D triangulation algorithm for closed contours was required to perform certain functions in the 3D algorithm. Its development is also included in this chapter.

In Chapter 4, the results of the tetrahedra generator applied to a test case are shown. Analysis of one cut-cell in particular is included, highlighting some of the improvements that were achieved via node removal. Also provided are some basic interpolations of a set of cylinders in a scalar field, demonstrating the accuracy of the new method.

Finally, Chapter 5 draws conclusions on the successes of the developed scheme. In addition, recommendations on the remaining work are given for improving the scheme, in light of some of its weaknesses.
Performing CFD simulations is an elaborate process consisting of pre-processing, solving and post-processing steps. For the process to be as efficient as possible, it is best if the phases are compatible with each other so that accurate results can be achieved in the least amount of time. This chapter will provide the background for Cartesian meshing (pre-processing) with the aid of Aftosmis’ lecture notes for the Von Karman Institute for Fluid Dynamics [1]. Also included is a brief explanation of the theory behind scientific data visualization (post-processing). It will conclude by showing why current techniques for visualization of data generated from Cartesian grids is inaccurate.

2.1 Theory of Cartesian Grids

2.1.1 Volume Mesh Generation

The first step in building the Cartesian grid is to generate the original coarse volume mesh. It consists of a block of uniformly shaped, hexahedral cells. These cells are all right parallelepipeds (in the three Cartesian directions). The body of interest does not impact the construction of this grid. Consequently, generating the mesh is quite simple and the true task is to develop algorithms that do this as efficiently as possible.

This coarse mesh is the volume in which the flow field will be solved, and it
Figure 2-1: A block of Cartesian cells.

Figure 2-2: A round body intersecting a block of 2D Cartesian cells. The cut-cells are the cells which are intersected by the body.
should contain all of the geometry. The geometry of the body can be defined in several different formats such as CAD format, trimmed Non-Uniform Rational B-Splines (NURBS), stereo-lithography, etc. A common approach is to deal with each component of the entire geometry individually. This method is especially effective when faced with multiple component configurations. Each component's exposed surface can be extracted and used by the mesh generator, thus eliminating the complications that may arise where components overlap. With these surface triangulations, all of the Cartesian cells cut by the triangles must be identified. Voorhies presents an efficient procedure for determining the intersections between hexahedral cells and surface triangulations [38]. It is based on a series of intersection tests and quickly eliminates non-candidate triangles.

Once the cut-cells have been identified, this coarse mesh can be refined according to the features of the geometry inside the volume. A 2D cell will be split into 4 children and a 3D cell into 8 children, as shown in Figure 2-3. Although this refinement is not done for the initial grid, the refinement can be varied to match the changing flow field. This attractive feature proves advantageous while solving.

![Figure 2-3: Cell refinement (2D and 3D).](image)

There are various strategies for deciding whether or not to subdivide a cell. In general, all of the cut-cells will be refined to a pre-determined level. This refinement must be gradual and hence will propagate into the mesh. Further refinement above this level is based upon the features of the geometry within the cut-cells. One such scheme proposed by Aftosmis involves studying the angular variation of the surface normal inside an individual cut-cell and between adjacent cut-cells [1].

When cells are subdivided, the information concerning the triangles to which the parent cell was associated must be properly passed on to its children. This will
depend on the data structure type. There are three distinct types currently in use for the storing of Cartesian mesh data. The first is quadtree (in 2D) or octree (in 3D) connectivity. These types of tree-based structures are well-suited to refined Cartesian grids because of its hierarchical nature. When a “parent” cell is subdivided into “children”, pointers connecting the addresses of each are established and stored in an array.

Another approach is the structured Adaptive Mesh Refinement (AMR) method developed by Berger and Colella [7, 16, 13, 10, 12, 11]. Cells are tagged based on an estimated truncation error and grouped into rectangular patches. Within these patches, new cells are tagged creating smaller patches, and so on. Once the cells are all refined, the result is several levels of block-structured, locally refined, grid patches. The structured arrays associated with these grid patches allow for compact memory storage.

The third and final approach is the use of unstructured data structures, where the connectivity of the refined grids is stored explicitly. In 3D, lists for each of the faces of a given cell are used to point to the adjacent cells. Cells at different levels of refinement are also connected in a similar manner, pointing to both the finer cells and the original coarse cell.

Once the refinement process is complete and the cut-cells have been identified, it is necessary to remove the cells that lie completely inside the body. These cells do not contribute to the solution. Identifying them becomes a point-in-polyhedron...
application, for which there are two popular methods. The first is a ray-casting approach [31]. A ray is cast in an arbitrary direction from a point and the number of times it intersects the body's boundary is recorded. If the point is outside the body, this number will be even and if it is inside, the number will be odd. The other approach makes use of the winding number. This number is calculated by keeping a running total of the signed angles between successive polygon edges (or faces in 3D) with respect to the point of interest. If the point is inside, the winding number will be $2\pi$ and if it is outside, it will be zero.

![Figure 2-5: Identification of whether the cell lies inside the geometry or not. The winding number method on the left and the ray-casting method on the right.](image)

2.1.2 Cut-cell Intersection

Once the volume mesh is completed, treatment of the cut-cells can begin. After they have all been identified, defining the surface that intersects each cell can proceed. One of the most important tools used in defining the surface in cut-cells is polygon clipping. It is a procedure which uses one object as a window and removes the portions of a second object which are not visible inside this window. This procedure is useful in identifying how the flow interacts with the surface inside the cell.

The Sutherland-Hodgman is a typical polygon clipping algorithm used in Cartesian grid generators [18]. It marches from edge to edge in a specified direction around the target polygon (window), and compiles a sequential list of vertices. There are four possible cases for the geometry with respect to the clipping window.
Figure 2-6: Geometry intersecting a clipping window. On the left, the unclipped geometry and on the right, the result of clipping with the window.

Case 1: Edge goes from the inside of the window to the outside of the window. The first node (1) and the intersection node (A) are added to the list.

Figure 2-7:

Case 2: Edge is entirely outside of the window. None of these nodes (3, 4, 5, and 6) are added to the list.

Figure 2-8:

Case 3: Edge goes from the outside of the window to the inside of the window. The intersection node (B) and the last node (7) are added to the list.

Figure 2-9:
Case 4: Edge is entirely inside the window. Both nodes (7 and 1) are added to the list.

Figure 2-10:

Aftosmis notes that while there are many different ways of implementing these algorithms in code, it is their efficiency which is of utmost importance for Cartesian applications [1]. He recommends the use of outcodes for the vertices, identifying the regions around the cell and performing bitwise operators to determine whether edges intersect the target window or not.

A situation which causes complications for Cartesian grids is thin components which split cut-cells into multiple, independent regions. This often occurs when modeling thin objects such as fins or wing trailing edges. Locating these cells and treating them properly is important since it often occurs. One suggested approach is to first identify all of the edges that outline the sections of the cell which separate the body from the flow [29]. With these edges, polygons on the cell faces are formed and grouped to form the independent regions, which are treated separately.

Figure 2-11: A thin component splitting the cut-cells.

2.1.3 Examples of Complex Geometry

To show how much of a time savings the Cartesian approach provides, an example of a complex, asymmetrical mesh is presented here in Figure 2-12 with permission of
Mike Aftosmis [1]. It consists of three twin-tailed fighter geometries and a helicopter. This configuration contains 121 components, described by 807000 triangles. After 6 minutes and 30 seconds of computation time on a MIPS R10000 workstation with a 195 MHz CPU, the final mesh was obtained. The total number of cells created was 5.61M and required a maximum 365 Mb to compute.

![Image](image)

Figure 2-12: A large scale, Cartesian grid with multiple levels of refinement.

### 2.1.4 Solutions and Boundary Conditions

Flow solvers for Cartesian grids solve the 3D Euler equations. These equations describe the conservation of mass, momentum, and energy for ideal, compressible, inviscid fluid flows in 3D. A simple approach that can be used to solve these equations is a cell-centered finite volume scheme.

\[
\frac{d}{dt} \iiint_{vol} w \, dxdydz = - \oint_{S} \mathbf{f} \cdot \mathbf{n} \, dS \tag{2.1}
\]

where \( w = \text{state quantities} = (\rho, \rho u, \rho v, \rho w, \rho E) \)
\[ dV = \text{incremental volume of the cell} \]
\[ \mathbf{f} = \text{flux quantities} \]
\[ \mathbf{n} = \text{normal vector, with respect to the face} \]
\[ dS = \text{incremental area of the face} \]

\[ \mathbf{f} \cdot \mathbf{n} = \begin{cases} 
\rho \mathbf{v} \cdot \mathbf{n} \\
\rho u \mathbf{v} \cdot \mathbf{n} + pn_x \\
\rho v \mathbf{v} \cdot \mathbf{n} + pn_y \\
\rho w \mathbf{v} \cdot \mathbf{n} + pn_z \\
(\rho E + p) \mathbf{v} \cdot \mathbf{n} 
\end{cases} \quad (2.2) \]

where \( \rho = \text{mass density} \)
\( \mathbf{v} = \text{velocity vector} \ (u, v, w) \)
\( \mathbf{n} = \text{normal vector} \ (n_x, n_y, n_z) \)
\( p = \text{static pressure} \)
\( E = \text{total energy} \)

In typical grids, the majority of the cells are regular with no body intersections and so it is easy to apply this scheme. For the cells which intersect the boundary of the geometry, the summation of the flux contributions is not as simple since the flux must now consider the flow around these solid boundaries. There are different ways to impose these boundary conditions. One of the most often used methods is to calculate the volume centroid of the cut-cell and the area centroids of all the cell’s faces so that a linear extrapolation can be made [6].

One of the drawbacks of using a Cartesian approach is the complications involved in producing viscous solutions. Although research on this topic continues, current methods for resolving boundary layers and viscous features is inefficient. Another drawback is the specification of the boundary conditions for solving. This is the side-effect of significantly simplifying the grid generation process. While they are
difficult to develop, there are many different schemes and there have been several papers documenting their accuracy [16, 12].

2.2 Background of Visualization

Visualization of the data produced from the solution is an essential procedure for CFD analysis. It permits engineers and researchers to assimilate the large amounts of data provided by the flow solver and hopefully understand the flow field. The first widely used 3D visualization package, PLOT3D, was developed at NASA Ames Research Center [24]. At the time, it dealt with steady data on structured grids. Shortly after, Giles and Haimes from M.I.T. created VISUAL3 which could handle unsteady data on both structured and unstructured grids [21].

Flow solvers calculate flux quantities such as mass density, momentum and energy. This data is given at the centroid of each cell for Cartesian meshes and then distributed onto the supporting nodes for the cell. With it, scalar fields for other quantities such as temperature, pressure, and velocity can be described. Furthermore, quantities which have components, such as velocity, can be combined to build vector fields. Most scientific visualization algorithms are finite-element based. Therefore, they use interpolation of the calculated fields at the nodal locations to specify information at a point in the volume. Typically, a form of linear interpolant is used for all three directions.

There are several visualization techniques designed to help understand fluid flows. Some of the basic techniques are surface rendering, geometric slicing, streamlining, and particle tracing. While these techniques are useful, understanding complex fluid flow is still a daunting task. Over the past decade, there has been significant effort to develop a means for extracting specific flow features[23]. Some of the features of interest are shock formations, separation and re-attachment lines, vortex cores, recirculation zones, and boundary layers.

To produce geometric cuts and iso-surfaces in a 3D volume, the resultant surface in each cell must be determined. To do this effectively a lookup table is constructed.
A lookup table records a binary number which identifies which of the edges are intersected by the iso-surface or geometric cut and which are not. For a hexahedral cell in a Cartesian mesh, the lookup table is 256 in length \(2^8\) where 8 is the number of nodes for the cell). The index to this table is based upon whether the scalar value at each node that supports the cell is above the value of the surface cut. This scheme assumes that only one intersection along an element edge is possible and its placement is linearly interpolated using the scalar values of the nodes.

To perform streamlining and particle tracing, integration schemes which track the particles movements with time are employed. These schemes require the velocity returned at a specified location within the domain. Rarely is this location exactly at a node that supports the mesh. Interpolation is again required to determine the value.

### 2.3 Visualization Complications

As has been mentioned, the majority of the cells in a Cartesian mesh do not intersect with the geometry of the body. For these cells, applying the normal visualization algorithms will produce accurate results. However, applying them to the cut-cells will yield questionable results. This is problematic because the data in regions close to the body is of great importance to the engineers and designers.

Figure 2-13 is an enlargement of a 2D Cartesian grid near the surface of the body (the interior of the body is the shaded region). The nodes of the grid which are inside the body are denoted by a cross and those outside or situated upon the surface are denoted by a solid circle.

Only nodes outside the body or along the surface will have valid data since they are in the computational domain. It is feasible that they be used for interpolation of flow quantities within the cells. Nodes inside the body, however, will have no data and instead be filled with values that are useful for the solver. Despite this fact, many of the current visualization packages utilize whatever value is recorded at these invalid nodes for interpolations, clearly producing an erroneous result. Figure 2-14
shows the result of a cut surface through two cylinders placed in a linear scalar field. It illustrates how the use of these invalid nodes produces incorrect results of the data near the body. The vertical gradient lines which correspond to different scalar values are no longer uniform near the body. In Figure 2-15, we calculated a streamline in a spiraling vector field. Once again, the vector field quantities at the invalid nodes were used for the interpolation outside of the body. The result is a streamline which cuts through the body and emerges further downstream. This is not physically correct.

The interpolation problem is clearly depicted in Shultz, et. al.[35] in their Figure 3. Here the streamlines pass through the car body, a non-physical result. The authors attempt to remedy the problem by stopping the integration where the path intersects
the body. As far as the solver is concerned, fluid does not enter nor leave the car hood, but flows around it. This indicates that the data handling is incorrect, not the solution.

At first glance, it would appear that it would be possible to develop an interpolant for the cells depicted in Figure 2-13 by using the the four valid nodes for each cell (two on the surface and two in the computational domain). It would however only be valid for cases where the intersection of the surface is a simple line segment through the cell. Consider the cells in Figure 2-16 with the more complex body definition. It would be impossible to accurately interpolate the flow near the surface using only four nodes for each cell. This problem is further amplified in 3D where the geometry has a higher level of complexity.

![Figure 2-16: Intersection of a 2D body by the cut-cells. The body definition is much more complex.](image)

For the sake of argument, suppose it is possible to come up with some form of interpolation for the cut cell based upon the cell vertices that are valid as well as those that support the body discretization. This could produce a non-simple and possible concave element, which would prohibit the use of the lookup table for geometric cuts and iso-surfaces. An additional complication associated with this idea is that the size of the look up table is $2^n$ where $n$ is the number of nodes. It would not be out of the question to get more than a 20 node element in 3D. Generation, storage, and reference to this table would all be difficult to implement.

One could imagine using a higher order interpolant that could be developed to provide better definition of the field quantities within the cut-cell. Accompanying
this solution however are several complications in actually constructing the function. This would preclude us from using the lookup table, however, because there could be multiple crossings along an edge.

Another complication to the visualization of results from Cartesian systems is that there is a form of hierarchical embedding used for the refinement of the cells near the body. When a parent cell is split into children, new nodes are created along its edges. For the cells neighboring the parent which were not refined, there are hanging nodes on the edges. Interpolating near these nodes while inside the non-refined cells becomes a problem since the resultant values will not be contiguous.
Chapter 3

Development of the Tetrahedral Mesh

The use of Cartesian approaches to solving fluid flows has been growing significantly in recent years, due to its ability to rapidly generate grids for bodies with complex geometry. Despite requiring special treatment of the boundary conditions at the cut-cells which intersect the body, Cartesian approaches speed up the overall CFD process. One of the areas that remains to be improved for these approaches however are the visualization schemes. Within the cut-cells, current schemes inaccurately interpolate data near the body’s surface. This incorrect visualization poses a problem to engineers and designers understanding the fluid behavior, especially since this region is of significant interest.

For this reason, a new technique must be developed to handle the cut-cells which allows the use of existing visualization algorithms. The solver’s results must be represented to the investigator with the correct imagery. The development of this technique will be the focus of the remainder of the chapter. First, the concept for the proposed solution along with the constraints involved are presented. A 2D algorithm was developed and tested so that the fundamental concepts could be understood before proceeding to the more complex 3D algorithm. Both of these approaches will be presented. Also included in this chapter is the development of an algorithm which triangulates an arbitrary contour in 2D, given a set of constraints, which was used by
the 3D algorithm.

### 3.1 Proposed Solution

The proposed solution is to generate a body-fit, tetrahedral mesh inside each cut-cell. This is counter-intuitive since the Cartesian approach is designed to be a non body-fitted method. The technique used to solve this problem is one that always maintains a properly filled volume. It does so by employing specified cutting primitives on the tetrahedra. These cutting primitives form the foundation of the solution. When a tetrahedron is cut, it is split into several smaller tetrahedra such that the sum of the volumes of the smaller ones is exactly that of the original volume.

A Cartesian cut-cell (a right parallelepiped) is the starting point. This element is simple and convex and can always be broken up into either 5 or 6 tetrahedra. The body discretization is imprinted into the cube. Tetrahedra are formed along it using the basic cutting primitives and supported by the nodes from the body surface and the corner nodes of the cut-cell. Once created, the tetrahedra inside (or outside) of the body are removed since they are not in the computational domain, leaving a body-fit mesh bounded by the cut-cell.

Where the existing interpolation schemes failed to properly provide information near the surface of the body, this new scheme will succeed. The problem with existing schemes is that they use the invalid nodes which are inside the geometry. Furthermore, if only the valid nodes were used, there are not enough of them in a single Cartesian cut-cell to accurately describe what could be a very complex geometry inside the cell. With the new scheme, all of the nodes that describe the surface within the cell are used to generate tetrahedra. These nodes represent valid points since the information is described via the boundary conditions and the solution. The generated tetrahedra provide a fitting structure to interpolate the required information.
3.1.1 Addition of Nodes

The importance of the validity of the solution at the supporting nodes is the largest constraint on the tetrahedra generation. Since only nodes which have a solution provided for them may be used to properly interpolate, nodes cannot be arbitrarily inserted inside the cut-cell region being meshed. This precludes us from inserting them anywhere in the flow except for the corner nodes of the cut-cell. The only new nodes we can insert must be on the body discretization, since the solution is known along it.

It is well known that any 2D region defined via an outer collection of ordered segments can be filled with triangles. There are a number of algorithms in computational geometry that will perform this task very rapidly and robustly with no additional nodes required. It is also well known that this is not the case in 3D. Many circumstances are present[14] which prevent the filling of a husk of nodes defined by the closed triangular discretization of surfaces with tetrahedra. A number of unstructured grid generators exist that can routinely fill arbitrary volumes with tetrahedra and they are able to overcome this volume fill problem by inserting nodes into the volume of interest. This is clearly something that cannot be done in this case, since these arbitrarily inserted nodes will not be valid while interpolating. This complication makes 3D tetrahedra generation quite a daunting task.

3.1.2 Additional Goals

Along with the constraints related to the insertion of nodes, additional goals were specified according to which the construction of the algorithms would be based. The first goal was the ability to generate the tetrahedral meshes quickly. Ideally, the technique would generate this visualization mesh on the fly. In this way, no additional memory would be required to hold the resultant body-fit mesh. This goal matches well with the Cartesian approach which is to avoid lengthy processes.

The second goal was to develop a robust algorithm. It must be deterministic (require no user intervention) and always function even when applied to the most
complex of geometries. When working in complex geometry, all special cases must be considered due to the strong likelihood that each situation will be encountered. This goal also matches the Cartesian goal to keep the entire process automated.

3.2 Two-Dimensional Triangle Generation

Before tackling the more complex problem of tetrahedra generation in 3D, a simpler version of the problem was dealt with in 2D. Only the basics of the proposed solution were used in the testing to ensure that there were no fundamental errors. No Cartesian grids nor any discretized bodies were required. The test geometry consisted of a single square cut-cell and a set of specified nodes which represented the discretization of a body through this particular cell. With such a simple model, it was easy to quickly develop a working 2D code and start testing various discretization patterns to see which algorithms would work best.

3.2.1 Basic Cutting Primitives

At the backbone of the 2D algorithm are the basic cutting primitives. These cutting primitives are applied to a triangle when an edge is crossed or a node is anywhere inside. The most important feature of these primitives is that the area of the triangle, before it is cut, is identical to the total area of all the smaller triangles spawned from its cutting. This will ensure that the total area of the geometry inside the cut-cell remains the same despite changes. This method will not produce any negative area triangles.

1. Edge insertion:

   The first cutting primitive is the edge insertion, shown in Figure 3-1. A node is created on the edge of a triangle. This node is connected to the other triangle node which is not part of the edge being cut. This new connection creates an edge which splits the original triangle into two smaller triangles. There are two ways that this situation will arise. First, if one of the discretization nodes falls
directly along the edge of an existing triangle, then this triangle must be split via edge insertion. The other possibility is if a ray cuts through the triangle. Upon crossing an edge and entering or exiting the triangle, a node is created at the intersection of the ray and the edge. With this new node, the triangle can be cut. It is important to note that in this case the neighboring triangle which shares the edge where the node is created must also be split using edge insertion. This keeps the topology simple; each edge separates only two triangles.

Figure 3-1: Edge insertion 2D cutting primitive.

2. Triangle insertion:

Triangle insertion is the other cutting primitive used in the 2D algorithm, shown in Figure 3-2. A node is inside the triangle and does not fall along one of the edges. In this case, the newly inserted node is connected to all three of the original nodes. These resulting connections form three edges which delimit the three new triangles that replace the original one. Triangle insertion can only occur via insertion of discretization nodes. No neighboring triangles are impacted by this operation.

The special case that remains is what to do to when a newly inserted discretization node coincides exactly with an existing node. The approach taken is to represent the location with a single node.
3.2.2 Algorithm

Presented here are the steps of the 2D algorithm used to test the concepts which will be used in 3D.

1. Separate the square cell into two right triangles along the square’s diagonal.

2. Identify where the discretization from the neighboring cell intersects with the original triangles. Use the edge insertion cutting primitive, inserting a node at the intersection.

3. Insert all the discretization nodes that are inside the boundaries of the cell (in consecutive order). Use the corresponding cutting primitive to split the existing triangles with respect to the newly inserted nodes.

4. The first discretization node inside the cell becomes the end node and the discretization node immediately before it (which was created in step 2 on the cell edge) becomes the start node. Connect the start node to the end node using a ray (straight line).

4.1. If this ray cuts the edge of a triangle before reaching the end node, insert a new node at the intersection and cut the triangles accordingly using the edge insertion cutting primitive. Continue along this ray until the end node is reached.
4.2. If no edges are intersected, the start node will connect directly to the end node (i.e. they are both part of the same triangle and the edge between them is the discretization line).

4.3. Once the end node has been reached, it becomes the start node. The following discretization node becomes the end node and steps 4.1 through 4.3 are repeated. Continue doing so until the final discretization node (which should be outside of the cell) is reached. The intersection of the discretization body with the cell is now complete.

5. Move along the described boundary, identifying and marking which triangles are inside the body.

6. Sweep through the remaining unmarked triangles and mark those that are inside the body.

7. Swap triangle edges wherever necessary.

One of the most popular 2D triangulation methods is the Delaunay triangulation. Given a set of points, the plane is separated into domains based on the distance between points (known as a Voronoi diagram) and using circles built with the nodes of each triangle, a good quality mesh is produced. Delaunay triangulations, however, are computationally expensive. The triangulation produced by the above algorithm often results in bad aspect ratio triangles. To achieve the quality of a Delaunay triangulation while avoiding the computation time, edge swapping algorithms are employed [39, 25]. Typical algorithms look at the angles between edges on each triangle in their original configuration. It then compares these angles to the case where the opposite nodes of the triangle set are connected (dashed line in the left of Figure 3-3). The configuration that is selected is the one that lowers the maximum angle (or increases the minimum angle). Only edges that appear inside both triangles were considered for swapping. It should also be noted that the swap operation was only performed for triangles inside the computational domain, and hence none of the discretization edges were swapped.
3.2.3 Example with Illustrations

To help clarify the operation of the 2D algorithm, an example cut-cell is presented. Each step of the algorithm is given with a description of what is being done and an accompanying illustration. The algorithm was linked with GV, a graphic display tool for viewing geometry.

The discretization that will be used for this example is shown in Figure 3-4. There are two discretization nodes that fall within the cell boundaries. Typical cells do not have a jagged body definition as depicted here. It was used in this example since it helps illustrate the various steps of the algorithm.

The first step is to separate the original square cut-cell into triangles (step 1) as shown in Figure 3-5. This is not done via the cutting primitives, but is a standard procedure applied to each cut-cell before proceeding with the rest of the algorithm. There are no hanging nodes present in this example, but if there were, the triangles
would have to be split accordingly. Note that the total area of the cut-cell is conserved.

The discretization pierces the cell on the left at location A in Figure 3-6. With the edge insertion operation, the triangle is split into two smaller triangles (step 2).

Now, the discretization nodes are inserted (step 3). Using the face insertion cutting primitive, the triangles which these nodes fall into are split into three smaller triangles. Note that nodes are always inserted in sequential order and so node B is inserted first and then node C. Although this has no significance in this example, it would affect the triangulation if the nodes were close together.

Once the basic triangulation is complete, we can begin to connect the discretization nodes (step 4). Nodes A and B are already connected, and so we can proceed
to the next set of discretization nodes. Node B now becomes the start node and C the end node. Tracing a ray from the start to the end results in an intersection at location D. A node is inserted here and the edge insertion operation is used to cut the triangle.

![Figure 3-8: Step 4.1.](image)

As the ray continues from node D, it again crosses an edge, resulting in the creation of node E. When this edge is split, it is connected to node C, which is the end node and so this step is complete. Node C now becomes the start node and the end node is the next consecutive node, which is outside the cell. Tracing from start to end, it crosses the boundary of the cut-cell and node F is created, as are two new triangles via edge insertion.

![Figure 3-9: Step 4.3.](image)

Now that the entire discretization of the body has been defined inside the cell, the triangles on either side of the discretization are marked (step 5). The triangles inside the body which touch this line are shaded in Figure 3-10.

The remaining triangles which are inside the body are marked (step 6). This is trivial once the triangles inside the body along the discretization line have been identified. Finally we have a cut-cell where the body is defined and there are several triangles in the computational domain which can be used to interpolate the values.
anywhere. No edge swapping was performed on any of these triangles since it was not necessary to improve the mesh quality.

**3.3 Three-Dimensional Tetrahedra Generation**

Development of the 2D algorithm demonstrated that creating triangles to be used for interpolation will work. Its development also helped identify many aspects of the algorithm and code writing that will be of importance when developing the 3D algorithm. Some of the more notable issues were the connectivity between cells and their neighbors, the orientations of cells, the node numbering conventions, and the numerical precision of the information.

With these issues in mind, the 2D algorithm was extended to a 3D algorithm. The 3D algorithm is much more complete in that it will take a complete, water-tight body discretization and the corresponding Cartesian mesh and produce the tetrahedra for all of the cut-cells of the mesh. As was done for the 2D algorithm, the 3D algorithm was linked to GV to allow visualization of the 3D graphics.
Of the obstacles encountered in generating the 3D algorithm, two stood out as being the most challenging. The first involved spatial visualization and understanding of the geometry in 3D. For example, in 2D, tracing from node to node is easy to track because the ray can only go in one direction. In 3D, each triangle has two directions that the ray can follow and tracking both is much more difficult. The other difficulty was the book-keeping and storage of the necessary information. Special data structures were required for all of the triangles, tetrahedra, and nodes so that the algorithm performed efficiently.

3.3.1 Data Storage

Nodes are the support of the grid structure for the Cartesian mesh, the discretized triangles, and the tetrahedral mesh. They are also created by the 3D algorithm under several different circumstances. In light of the complications involved with dealing with 3D geometry, it was decided that it would be cleaner (and more expedient when providing the mesh for interpolation) to categorize the different node types. Four possible node types were decided upon. The first two (cell vertices and discretization vertices) are defined and supplied by the user since they support the initial geometry. The last two (edge nodes and interior nodes) are created by the 3D algorithm. They are the results of cutting operations and help support the generated tetrahedral mesh. Here is a description of each:

1. Cell Vertices:
   These are the nodes that support the Cartesian cells and cut-cells. Only those that are actually part of the volume of interest (i.e. in the computational domain) are used. This also includes any hanging nodes.

2. Discretization Vertices:
   These are the points that make up the body discretization. Collections of three of these nodes form the triangular tessellation that makes up the body. It is assumed that this complete tessellation holds water. Also, all triangles that
make up the surface must have the same orientation so that the normals either point in a direction that is into or out of the body.

3. Edge nodes:
The 3D algorithm constructs these nodes. They are formed by cutting through the edges of the existing tessellation triangles. They are made up of two node indices, each of which must be a discretization vertex with an associated weight.

4. Interior nodes:
These nodes are also formed by the 3D algorithm. They are generated in the interior of the body discretization triangles. The index to the parent tessellation triangle in question is stored as well as the weights (to two of the three nodes) so that the linear interpolant inside the triangle can be applied.

Another storage issue which was originally encountered in the 2D algorithm was how to record the connectivity between tetrahedra. It was determined that keeping a list of the neighboring tetrahedra is important from a performance standpoint since techniques can be applied that use this information to avoid volume searches.

3.3.2 Basic Cutting Primitives
As was the case in 2D, there are a set of basic 3D cutting primitives at the heart of the 3D algorithm. They are applied to a tetrahedron during ray crossing, during insertion of the discretization triangles, or when a node is interior. The most important feature of these primitives is that the volume is maintained, similar to the area in 2D. This will ensure that the total volume of the geometry inside the cut-cell remains the same despite changes to the tetrahedra. Again, the methods described will not produce any negative volume tetrahedra.

1. Edge insertion:
The first cutting primitive is edge insertion, shown in Figure 3-12. A node is created on the edge of a tetrahedra. This node is connected to the two nodes which do not touch the edge being cut. This new connection is used
to split the original tetrahedron into two smaller tetrahedra. There are three possible situations where this primitive will be required. First, if one of the discretization vertices falls directly along the edge of an existing tetrahedron, then this tetrahedron must be split. Second, if a ray cuts directly through the edge of a tetrahedron, edge insertion must be applied. Finally, and the most typical scenario, is when a tessellation triangle cuts through a tetrahedron. The node is created where the planar triangle crosses the tetrahedron's edge.

Any number of tetrahedra can come together at an edge and so it is not possible to specify the number of neighboring tetrahedra that must be split using this same cutting primitive. Therefore, once the first tetrahedron being cut is located, successively examining and cutting the appropriate neighbors until returning back to the original tetrahedron is required.

2. Face insertion:

Face insertion is shown in Figure 3-13. A node is created on the face of the tetrahedron being cut. This node is connected to the three corner nodes of this face, splitting it into three. An additional connection is made with the node opposite the face which completes the splitting of the original tetrahedron into three smaller tetrahedra. Face insertion can occur in one of two manners. First, it is required if one of the discretization nodes falls directly on the face of an existing tetrahedron. It is also required when the edge of a tessellation triangle or a ray pierces through a face. The node is created at the point where the face is pierced.
For the face insertion primitive, if the face is interior (not exposed to the outside of the cut-cell), a single neighboring tetrahedron must be split about the same node. Face insertion is again required. This insures simple neighbor connectivity. Each inserted node therefore creates four new tetrahedra.

3. Tetrahedron insertion:
   The final cutting primitive is tetrahedron insertion (see Figure 3-14). A node falls entirely inside a tetrahedron without touching any of its edges or faces. This node is connected to all four of the nodes that support the tetrahedron. These connections form the edges that split the tetrahedron into four smaller tetrahedra. None of the neighbors of this tetrahedra are affected by the splitting.

For all of these primitives, the neighboring information is updated during the operation if necessary. The special cases in 3D are the same as those in 2D (when a new node coincides exactly with an existing node). They are treated the same way as well (the coinciding nodes are replaced with a single node).
3.3.3 Algorithm

Although this technique may have a few additional features, it is similar to the 2D algorithm described in Section 3.2.2. Because of the complex nature of 3D geometry, producing pictures of this algorithm will not illuminate the procedure as was done in 2D.

1. Generate the box

   1.1. The lower left and upper right set of coordinates are used to generate the eight nodes that support the cut-cell (it is a right-regular parallelepiped).

   1.2. This box is subdivided into six tetrahedra. While the hexahedron cut-cell could have been cut into five tetrahedra instead (which would have resulted in a lower number of tetrahedra), it is cut into six so that the directions of the diagonals on opposite faces match. This feature will be important if neighboring cells are patched together.

2. Insert the hanging nodes

   Any hanging nodes are included in the volume of the cut-cell. Cutting primitives 1 and 2 are used where necessary.

3. Compute the intersection of the box and the tessellation triangles

   A bounding box for each tessellation triangle is created. The width, height, and depth of the box are defined to contain the entire triangle. The box for each triangle is compared with that of the cut-cell. If there is any overlap, the triangle is considered for the following phases. It is important to note that even though all of the nodes for the bounding box of a triangle are outside of the cut-cell, it is still possible that the triangle intersects the cut-cell.

4. Insert the discretization vertices

   The vertices are inserted. Once placed, any of the three cutting primitives are used to split the existing suite of tetrahedra depending on the location of the vertex. It is also possible that the special case where discretization vertices
overlap box nodes exists (as was mentioned earlier). In this case, the node at this location is identified as a discretization vertex. It is assumed that the vertices that make up the body do not coincide.

5. Scribe the tessellation edges

5.1. The tessellation edges are inscribed in the volume by insuring that there is coincidence with tetrahedral edges. This is done by examining each tessellation edge and determining if one of its two nodes is outside the cut-cell. If either is outside, an edge node is inserted where the edge intersects and cutting primitive 1 or 2 or is used to split the tetrahedra appropriately. Again the special case exists where this node matches with a box vertex. If so, the box node is overwritten with the edge node.

5.2. The scribing of all the edges within the cut-cell is done with ray tracing. This method was demonstrated earlier in the 2D algorithm. Starting from a tetrahedron which contains the starting node (may be an edge node or a discretization vertex) a ray is cast toward the following discretization vertex (identified as end node).

5.2.1. If the end node is also on this tetrahedron, then the tessellation edge is already scribed.

5.2.2. If not, the intersection of the ray and the tetrahedron is found. Cutting primitive 1 is used if it intersects an edge along the opposite face. Cutting primitive 2 is used if it intersects one of the tetrahedron's faces. This ray casting function continues until an edge connecting the start node to the end node is complete.

Efficient operation of the scribing operation depends on the ability to find the neighboring tetrahedron to be pierced by the ray. A recursive algorithm was developed to perform this function, and it will be discussed further in Section 3.3.4.
6. Cut the tessellation triangles

6.1. This process slices the volume so that the faces of the tetrahedra match the faces of the body tessellation. There is an outer loop over all the triangles that intersect the current cut-cell. Each tetrahedron inside the cut-cell is examined. Tetrahedron nodes (either discretization or edge) that match the nodes for that triangle or match the triangle index itself are marked. If all 3 nodes are marked, then this tetrahedron is complete.

6.2. There is a special circumstance which rarely occurs that complicates the cutting operation. This problem does not exist in 2D which makes it more difficult to understand. What happens is that matching tetrahedra faces are not found for all the tessellation triangles. This is due to additional geometry, present from previous constructions, which interferes with the triangles. To solve this problem, the equation of the plane (that is supported by the triangle) is constructed. If this plane is found to intersect the tetrahedron (and the intersection point(s) are within the triangle) then the tetrahedron is cut using cutting primitive 1. If there are no valid intersections for this triangle, the tetrahedron is finished.

7. Mark the orientation of the tetrahedra that touch the tessellation

Another loop through all the triangles that intersect the cut-cell is done to look for the tetrahedra that have matching faces. When one is found, the node not part of the face is tested with the equation of the plane for the triangle. If it is found to be greater than the intercept, then the tetrahedra is marked with a positive orientation. If the result is less than the equation’s intercept, the tetrahedron is inside the body and marked as such.

8. Flood the orientation

A volume flood of all the tetrahedra is performed to find those that have not been assigned a value, as was described in the 2D algorithm.
9. Cleave the inside away from the outside

To perform the actual slicing of the volume, the neighboring information along the triangulation surface is removed. It is replaced with pointers to the owning triangle.

10. Remove possible interior and edge nodes

All interior and edge nodes are examined. Any of them that are completely contained within tetrahedra that have one face exposed and a matching node opposite to this face can be removed. This helps reduce the count of inserted nodes and also the number of tetrahedra. The node removal involves producing an outer loop that must be triangulated which is in turn extruded to the opposite node to form tetrahedra. This process is further discussed in Section 3.3.5.

11. Face swap

A face swapping operation is used on the desired volume to produce better tetrahedra and hence a better interpolant. The procedure is first done (in 2D) on the exposed box faces. This will ensure that the neighboring boxes will match at their internal faces. Then the 3D analogue (seen in Figure 3-15) is done for all interior faces[25, 19].

![Figure 3-15: 3D tetrahedron edge swapping. The original configuration is on the left with the candidate edge to be swapped represented by a dotted line. The result of the swapping is on the right.](image)
3.3.4 Recursion in Ray Tracing

In both 2D and 3D, one of the most often used features is ray tracing. It is the operation by which a ray is cast, connecting two consecutive discretization nodes together. If this ray intersects any geometry between these two nodes, it must be cut accordingly.

When originally developed, finding the nearest tetrahedron pierced by the ray consisted of looping through all of the tetrahedra. As the ray exited this tetrahedron, the next one was found by again looping through the entire list. Several loops through the tetrahedra list may be required before reaching the final discretization node. Loops are used in several places throughout the developed algorithms to find specific objects in different situations, such as discretization triangles and cut-cells. While loops are not the most efficient search methods, they are very simple and in many cases effective. Considering the much larger number of tetrahedra compared to other types of geometry, as well as the considerable frequency that the ray tracing operation is used, the use of loops is perhaps not the most effective method of locating the tetrahedra to cut. Given that one of the original goals was to have quick mesh generation, part of the focus of the research was to improve the speed of the ray tracing process.

Presented below is the algorithm for the first approach at a search algorithm which finds the pierced tetrahedra.

1. Perform an initial loop through the tetrahedra and stop when one is found which touches the starting node. This is the current tetrahedron.

2. Create a point an incremental distance away from the starting discretization node. This point falls directly on the ray which connects the starting and final nodes.

3. With this point, determine its weights with respect to the current tetrahedron. These weights indicate where the point is relative to this tetrahedra. They will be used to weave through the tetrahedra along the ray until the final node is reached.
3.1. If the point is inside the current tetrahedron and this tetrahedron also has the starting node as one of its nodes, it is to be split using the appropriate cutting primitive. A node is created where the ray exits this tetrahedron and it becomes the new start node. The current tetrahedron becomes the last one created as a result of the split.

3.2. If the point is inside the current tetrahedron but this tetrahedron does not contain the starting node then the incremental distance is too large. The tetrahedron being looked at is too far away. A new point a shorter distance away is created.

3.3. If the point is not inside the current tetrahedron, one of the neighboring tetrahedron becomes the current tetrahedron. Which one in particular depends on the sizes of the weights.

4. Steps 2 through 3 are looped through until a tetrahedron which contains the final node is found, and the connection is complete.

There proved to be a fatal error with this algorithm. Due to the available numerical precision of the data, tolerances were set to help determine whether points were inside or outside of the tetrahedra. Also, some of the tetrahedra created by the algorithm were very small (near zero volume). The combination of these two factors resulted in circumstances where the point lay in an infinitesimally small tetrahedron, the current tetrahedron was its neighbor, and the tolerances always indicated that the point was inside the neighbor. It was impossible to exit the current tetrahedron and move to the tetrahedron with the start node.

Because of this failure, a second approach was taken in speeding up the search process. This new approach was to use recursion to scan through all of the local tetrahedra and find the one that is pierced. Recursion is the ability of a function to call itself. It turns larger problems into smaller, repetitive ones which are easier to solve. Each time it is called, the new function gets a new set of specific variables. Presented below is the algorithm used to find the pierced tetrahedron recursively:
1. Trace the ray from the start node to the finish node.

2. Perform an initial loop through the tetrahedra and stop when one is found which is made of the starting node. This is the current tetrahedron.

3. Determine whether any of the faces of this tetrahedron are pierced by the ray.

   3.1. If the ray pierces the current tetrahedron it is split using the appropriate cutting primitive. A node is created where the ray exits this tetrahedron and it becomes the new start node. The current tetrahedron becomes the last one created as a result of the split.

   3.2. If the ray does not pierce the current tetrahedron, its neighbors are checked, one by one, recursively. The neighbors are not candidate to be checked if they do not contain the starting node. Each neighbor itself has four more neighbors and so the recursion continues until the tetrahedron which is pierced is found.

4. Step 3 is repeated until the tetrahedron which has the finish node is found.

This algorithm functions properly. Comparing its performance to the original method which looped through all the tetrahedra was done with PIXIE. PIXIE is a standard unix tool which monitors the performance of a computer program. It creates a profile of the amount of time spent performing each instruction. The profiles of the two methods are presented in Table 3.1.

<table>
<thead>
<tr>
<th></th>
<th>Original Algorithm</th>
<th>Recursive Algorithm</th>
<th>Total Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time (in seconds)</td>
<td>3.978</td>
<td>3.297</td>
<td>-0.681</td>
</tr>
<tr>
<td>Percentage of Overall Run-time</td>
<td>4.0</td>
<td>3.3</td>
<td>-0.7</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of the original algorithm's performance to the new recursive algorithm's performance.
Use of the recursion algorithm results in an 18% improvement in the speed of the ray tracing operation. These savings did not speed up the overall process as much as was anticipated. It was originally believed that the majority of the computer’s time was spent in this searching loop but it is actually spent in other parts of the overall 3D algorithm, namely the cutting, marking and imprinting operations. Ray tracing only occupies about 4% of the computer’s resources and so additional work must be done to increase the speed of the whole process.

### 3.3.5 Node Removal

After examining the initial test cases, it was clear that with this new approach, very large numbers of tetrahedra and nodes are created. This is a function of the complex nature of the geometry within the cut-cells. As the level of complexity increases, there is a steep increase in the geometry created. To try and keep these numbers as low as possible, strategies are needed to try and remove nodes and tetrahedra (step 10 in the 3D algorithm). Since the supplied nodes (discretization vertices and cell vertices) are not removable, we must deal only with the inserted nodes (edge nodes and interior nodes).

Interior nodes are generated on the interior of the body discretization triangles. It is safe to assume that the majority of the tetrahedra connected to this node will have a face that matches this triangle and hence additional nodes will share this common plane. If all of the nodes surrounding the interior node connect directly to a single back node, then the interior node can be removed (see Figure 3-16). It would then be possible to re-triangulate the contour of nodes and extrude them to the same fourth node, creating new tetrahedra. This process results in an overall decrease of one node and two tetrahedra.

By extension, it would also be possible to remove edge nodes. These nodes exist on the edges between triangles. This situation is similar to that of interior nodes, except that instead of a single contour on one triangle, there are two half-contours on two different triangles (see Figure 3-17). The same conditions for creation of tetrahedra apply for each of the two contours and the end result is again the same.
3.4 Triangulation of 2D Contours

One of the aspects of the 3D algorithm which seems deceptively trivial is the triangulation of the 2D contours in the node removal step. When the algorithm was originally developed, an existing fast polygon triangulation algorithm based on a paper by Seidel [36] was inserted to triangulate the contours. While it performed adequately for almost all cases, its weaknesses became apparent through testing on more complex cases. Because of the ray tracing operation, there are strings of colinear nodes that can be quite close together created through the tetrahedral mesh. These colinear nodes become part of the 2D contours and the triangulation algorithm has difficulty dealing with them. It often generates colinear triangles (zero area) and these become troublesome for the rest of the algorithm.

A new 2D triangulation algorithm was developed to specifically rectify this situ-
The primary goal of this algorithm is to guarantee that there will be no zero area triangles upon triangulation of the 2D contour. While it may lead to bad aspect ratio triangles, it is not as bad as zero area triangles. Furthermore, once the contour has been triangulated, edge swapping can be used to improve the quality of the mesh.

The original attempt at developing this algorithm was not successful. It consisted of looking at every node around the contour and determining whether the two edges connected to it formed a concave (more than 180°) or convex (less than 180°) loop. If the entire loop was convex without any colinear nodes, triangulation was trivial and the nodes where stitched together. If a segment was colinear or concave, following nodes where looked at until convex sections where found, and new triangles created accordingly (see Figure 3-18).

![Figure 3-18: An example of the triangulation of a 2D contour. Angles ABC, ABD, ABE, ABF are concave and so a special triangulation is required until we reach the convex section ABG.](image)

While it was adequate for most cases, this method was deemed unacceptable. Too many special cases were encountered, and often meshes would be concentrated entirely on one node, depending on which node the algorithm started with.

The second approach is more methodical. Two fundamental operations form the base of this algorithm. They are triangulation schemes which deal specifically with
strings of colinear nodes. These operations are fanning and stitching, shown in Figure 3-19.

Figure 3-19: The fanning (on the left) and the stitching (on the right) operations for colinear nodes.

Presented here is the algorithm for 2D polygon triangulation.

1. Identify all of the major, non-colinear nodes that exist around the 2D contour.

   Figure 3-20: Major nodes are circled.

2. Calculate the angles of all the elbows (where two edges meet a node) of the contour.

   2.1 If all of the elbows are convex, proceed to step 3.

   2.2 If there are concave elbows, find the one with the largest angle (the candidate node).
Figure 3-21: Concave elbows are circled.

2.2.1. Examine the angles of the elbows before and after the candidate node. The one with the largest angle is connected to the candidate node to form a triangle. This triangle is chopped from the contour and the back node is removed from the contour.

Figure 3-22: The largest elbow is at node A and the elbows to which it can connect are circled (B and C). Once connected, the shaded triangle is chopped from the contour, as is node D.

2.2.2. All colinear nodes within the triangle being chopped are found. Using a fanning or stitching operation, the colinear nodes are connected to form all non-zero area triangles.

2.2.3. With the new contour, step 2 is repeated until a convex contour is found which has no colinear sections.

3. Stitch the major nodes of the convex contour to form large triangles.

4. For each large triangle from the stitching operation, determine if there are colinear nodes along its edges.
Figure 3-23: All concave sections have been removed (shaded areas) leaving only a convex contour.

Figure 3-24: Stitching of convex contour.

4.1. If there are no colinear nodes on any of the edges, the triangle is complete.
4.2. If there are colinear nodes on only one of the edges, a fanning operation is required to produce the non-zero area triangles.
4.3. If there are colinear nodes on two or more edges, a stitching operation is required, followed by an additional fanning operation (if required).
4.4. This step continues until all of the large triangles created in step 3 have been processed.
Figure 3-25: Completed triangulation.
Chapter 4

Tetrahedra Generation and Interpolation: Test Cases

Throughout the construction of all the algorithms, two different geometries were used to test all aspects. While the algorithms were thought out and developed on paper beforehand, they never function perfectly the first time and these cases were the perfect testbed to identify the shortcomings. Furthermore, the use of GV to view the complex geometries related to the tetrahedra and the cut-cells was invaluable. It provided a visual clue which is crucial when working in 3D. The first test case was a discretized pig for which a full Cartesian mesh was already specified (see Figure 4-1). The other test case was two connected discretized cylinders (see Figure 4-2) for which the location of the cut-cell intersection was manually specified.

Since the discretized pig has cut-cells specified for its entire surface, it is used to study the performance of the algorithms. The large variety of intersections with the cut-cells will give representative data concerning tetrahedra counts and timings. Particular analysis of one cut-cell gives an accurate depiction of the generated tetrahedra. This cell can also be tested to see the impact of the node removal procedure that was implemented in the algorithms. As for the discretized cylinders, they are placed in simple fields to test the performance of the new interpolation schemes. All of these results will be presented in the remainder of this chapter.
Figure 4-1: Discretized pig.

Figure 4-2: Discretized cylinders.

Figure 4-3: Discretization of a 3D pig in a Cartesian mesh.
4.1 Tetrahedron Generation for a Discretized Body

The discretized pig is a closed, conformal, water-tight geometry consisting of 7040 triangles. Surrounding it is a Cartesian mesh refined to various levels, as shown in Figure 4-3. There are 21,314 cut-cells that intersect the body.

4.1.1 General Results

As expected, the total number of tetrahedra generated was large. This is because each intersection spurs others and the cutting primitives continuously generated more and more tetrahedra. The total number of tetrahedra generated outside the body is 840,333, giving an average of 39.4 per cut cell. The largest number of tetrahedra that any cell contains (in the computational domain) is 7689.

Timings were conducted to determine the amount of time required to construct the tetrahedra for all of the cut-cells surrounding the body. This is considered an important measure of performance since one of the original goals was to be able to generate grids on the fly. Using a 250 MHz R10000 Octane, all of the 21,314 cut-cells were filled with tetrahedra in 58 seconds. It should be noted that the 733 cut-cells which intersected the most geometry took 24 seconds to fill. Although this is not a bad result, the timings are not good enough to consider not storing the results in a pre-processing step.

4.1.2 Results for a Particular Cell

To illustrate how the tetrahedra are connected to the cell vertices and the surface discretization nodes, one particular cut-cell was studied. It is shown in Figure 4-4. This cell was specified by the user and was not part of the original Cartesian mesh. Note that it encapsulates more geometry than usually found in a normal Cartesian cut-cell. It contains 16 discretization vertices, 98 triangle edges, and 57 discretization triangles.

A more detailed view of this cell in Figure 4-5 shows the numerous tetrahedra edges that connect the vertices created along the discretization surface to the cell.
vertices. As can be seen in the figure, the face that does not intersect with the body displays only two tetrahedra. The two tetrahedron faces at this end of the cube are supported by the four cell vertices, which match those of the neighboring cube in the computational domain. The cut-cell in this example is supported by only 8 cell vertices, and hence has no hanging nodes. If it did, the number of tetrahedra at the face of the cube would change correspondingly so that the nodes always match with neighboring cubes.

Figure 4-5: Close-up of the cell intersecting the pig foot.
This cut-cell produced a relatively large number of tetrahedra - 3026 in total, of which 1624 are in the computational domain. There were 451 new edge nodes and 215 new interior nodes created for this cell. Removing the tetrahedra that are inside the body, we are able to look at those that will be used for visualization (see Figure 4-9). This figure clearly shows the complex nature of the surface discretization. Keep in mind that all of the nodes created using this algorithm are on the discretization surface.

Figure 4-6: The surface is defined by the generated tetrahedra inside the cut-cell.

4.1.3 Node Removal

The node removal algorithms required the majority of the work during construction of the 3D algorithms. It was recognized that the number of tetrahedra generated would be large and that reducing these large numbers would be of great importance. Once the algorithm was complete, the following analysis was performed on the cut-cells of the pig discretization to measure the success of the node and tetrahedra removal.

Overall, the total number of interior nodes was reduced by 7%, the total number of edge nodes was reduced by 26%, and the total number of tetrahedra was reduced by 19%. There is a decrease of 35% in the number of tetrahedra when comparing the cells with the largest number of tetrahedra. All of this is summarized in Table 4.1.
<table>
<thead>
<tr>
<th></th>
<th>Before Node Removal</th>
<th>After Node Removal</th>
<th>Percent Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number of Tetrahedra</td>
<td>840333</td>
<td>682376</td>
<td>19</td>
</tr>
<tr>
<td>Cell with Highest Tetrahedra Count</td>
<td>7689</td>
<td>4981</td>
<td>35</td>
</tr>
<tr>
<td>Total Interior Nodes</td>
<td>199772</td>
<td>185778</td>
<td>7</td>
</tr>
<tr>
<td>Total Edge Nodes</td>
<td>254448</td>
<td>187660</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 4.1: Mesh reductions due to node removal.

Figure 4-7: Distribution of all interior and edge nodes versus the number of nodes per cell.
Figure 4-8: Distribution of removable nodes versus the number of nodes per cell.
While these numbers are not very impressive, they are slightly misleading. To understand why, consider first the distribution of the nodes among the cells. Figure 4-7 shows the distribution of interior nodes and edge nodes as a function of the number of nodes per cut-cell. A large majority of the nodes created through the tetrahedra generation process exist in cells which have less than 25 nodes (in the computational domain).

Consider now the distribution of removable nodes. Figure 4-8 shows the distribution of removable nodes as a function of the number of nodes per cell. As the number of nodes per cell increases, so does the ability to remove them.

These two facts explain why the percentages are lower than originally hoped. The majority of the nodes exist in cells with less than 25 nodes and these cells have the worst node removal percentages. With such a low number of nodes, these cells do not contain as much geometry and are generally less complex. The cells with higher degrees of complexity have much better node removal, which is encouraging since these are the cells which need it the most.

To illustrate this fact, we will revisit the particular cut-cell from the pig discretization in Section 4.1.2. Figures 4-9 and 4-10 compare the cell before node removal is applied and after. Clearly, certain regions of the cell where there were large concentrations of bad aspect ratio tetrahedra have been improved, which is the desired result.

Table 4.2 lists the data before and after the node removal for this cell. The removal percentages agree closely with those displayed in Figure 4-8. Further examination of other cut-cells shows that it is not out of the question to observe upwards of 70% of the nodes being removed in cases where there are large numbers of nodes per cell.

4.2 Interpolation

Up to this point, the developed algorithms have proven that they can reliably generate a tetrahedral mesh in the cut-cells of arbitrary bodies. The remaining step is to determine if these tetrahedra can be used to accurately interpolate the data near the
body, which is the original goal of this body of work. To do this, we will place the test case of the two cylinders in two different fields to see how accurate the interpolation is. These two fields are a linear scalar field and a spiraling vector field (first introduced in Section 2.3). Both are very simple in nature and so any obvious discrepancies in the interpolation will be immediately noted.

As was shown in Section 2.3, the linear scalar field was improperly displayed. The vertical gradient lines passing through the cylinder were distorted near the body, proof of the improper interpolation scheme (see Figure 4-11). Using the generated tetrahedra to interpolate this near-body region, we see in Figure 4-12 that the gradient lines are now consistent. Added to Figure 4-12 is the outline of the outside of the cut-cells which cut through the cylinders.

Table 4.2: Mesh reductions for the particular cell due to node removal.

<table>
<thead>
<tr>
<th></th>
<th>Before Node Removal</th>
<th>After Node Removal</th>
<th>Percent Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number of Tetrahedra</td>
<td>1624</td>
<td>1182</td>
<td>27</td>
</tr>
<tr>
<td>Total Interior Nodes</td>
<td>215</td>
<td>140</td>
<td>35</td>
</tr>
<tr>
<td>Total Edge Nodes</td>
<td>451</td>
<td>305</td>
<td>32</td>
</tr>
</tbody>
</table>
Revisiting the spiraling vector field, we see that improper interpolation leads to a streamline which cuts through the cylinders and emerges further downstream (see Figure 4-13). This is a non-physical result. Interpolating using the new scheme and the generated tetrahedra, we see in Figure 4-14 that the streamline follows the contour of the cylinder without piercing it. This is the proper result, further evidence that this new interpolation scheme is successful.
Figure 4-13: Incorrect interpolation of a vector field.

Figure 4-14: Correct interpolation of a vector field.
Chapter 5

Conclusions

A set of algorithms were developed which, when combined, generate a tetrahedral mesh inside the cut-cells from a Cartesian mesh. With these tetrahedra, classical finite-element based, visualization techniques can be applied to interpolate and view the scientific data for Cartesian meshes. Until now, visualization of the data near the boundaries of discretized bodies in these meshes was inaccurate. The algorithms were tested on different test cases with complex discretizations and proved to accurately interpolate the data. This is encouraging with respect to the future use of Cartesian techniques in CFD.

While the algorithms have performed well so far, some investigation remains. The two principle areas of concern are the count of the tetrahedra and the quality of the tetrahedral meshes being generated.

5.1 Tetrahedra Count

The most glaring issue is the large number of tetrahedra being generated. In some cases where single cut-cells intersected a significant amount of discretization geometry, there were upwards of 10,000 tetrahedra generated. A good tetrahedral mesh should have between 5 and 6 tetrahedra per node [27] and it is not unheard of to see examples of 2 to 3 tetrahedra per node in the test cases. This does not bode well for bodies that could potentially be surrounded by several hundred thousand cells.
Node removal techniques appear to be able to reduce the overall number of nodes and tetrahedra somewhere between 20% and 30%. This is still not enough since the number of tetrahedra generated will easily be near 1M in many cases. One plan of action that has not yet been implemented but will improve this situation is the repetition of the node removal step. The algorithm currently attempts to remove potential nodes only once. After this first pass, it is likely that more nodes would be in a position to be removed. A loop through this process until interior nodes and edge nodes can no longer be removed will help lower the overall node and tetrahedra count.

Another plan which promises to drastically lower the number of tetrahedra being generated will be implemented as part of the future work on the existing algorithms. The idea consists of aligning the tetrahedra edges along the discretization edges and surfaces [1]. It would be implemented in the ray tracing operation and involve swapping tetrahedra as the algorithm proceeds from the start node to the finish node. Proper manipulation of the tetrahedra will reduce the number of piercing operations and in turn will greatly reduce the number of nodes and tetrahedra generated.

Besides these improvements, an additional factor impacting the number of tetrahedra generated was identified while experimenting with the test cases. It was determined that the tetrahedra count is geometrically linked to how much of the discretized body intersects a given cut-cell. For example, if the entire discretization of the pig was contained in one cut-cell, the algorithms would produce a tetrahedral mesh but it would be composed of an extravagant number of tetrahedra. The reason being that scribing the surface discretization would result in several geometric intersections, which themselves would propagate due to all the 3D geometry present. It is important that an informed user set-up the Cartesian mesh, ensuring that a suitable number of triangles (not more than \( \approx 100 \)) is inside each cut-cell.
5.2 Mesh Quality

The nature of the algorithms is such that bad aspect ratio tetrahedra are inevitable, as illustrations presented earlier have shown. Ray tracing and the long strings of colinear nodes are most responsible for these sliver elements. On the one hand, these tetrahedra do not pose a problem for the visualization of data. The odds that the point of interest falls inside one of them are slim, and in the event that they do, an interpolated value can still be obtained using the values at the four nodes. On the other hand, it is highly unlikely that these grids could ever be used by solvers because of the major stability issues related to solving with these grids. Perhaps once the number of tetrahedra and nodes are reduced, the aspect ratios will be more acceptable.

Another result of the new scheme that must be investigated is the matching of the interfaces between neighboring cubes. Because there are some instances where they will not match up correctly, some sort of face swapping will be required. This would be a solver issue if these meshes were used directly and has little significance on the visualization of the data.
Bibliography


