To the Graduate Council:
I am submitting herewith a dissertation written by Vincent Charles Betro entitled "Fully Anisotropic Split-Tree Adaptive Refinement Mesh Generation with Tetrahedral Mesh Stitching." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computational Engineering.

Steve L. Karman, Jr.

Dr. Steve L. Karman, Jr.
Professor,
Computational Engineering
Major Professor

We have read this dissertation
and recommend its acceptance:

W. Kyle Anderson

Dr. W. Kyle Anderson
Professor,
Computational Engineering

Robert V. Wilson

Dr. Robert V. Wilson
Associate Research Professor,
Computational Engineering

John V. Matthews

Dr. John V. Matthews III
Associate Professor,
Mathematics

Accepted for the Council:

Stephanie Bellar

Dr. Stephanie Bellar
Interim Dean of The Graduate School

FULLY ANISOTROPIC SPLIT-TREE ADAPTIVE REFINEMENT MESH GENERATION USING TETRAHEDRAL MESH STITCHING

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee at Chattanooga

Vincent Charles Betro

August 2010

Dedication

This dissertation is dedicated to Cassandra Betro, my loving wife, who has seen me through the process of starting a family while getting my Doctoral Degree and loved me through all the difficulties life has presented. It is also dedicated to Savannah Betro-Gross, my step-daughter, who has taught me to love unconditionally through everything, because she is the kindest, best person I have ever met. I also dedicate this to my family and friends, who have given so much support through life changes, and to my colleagues, especially Dr. Steve Karman, who have helped me grow as an engineer, educator, and person. I thank God for giving me this opportunity and giving me the support I needed to make it through. Finally, I dedicate this dissertation to Vincent Moses Betro, who should be along shortly, for giving me a deadline...I cannot wait to meet you, my son!

Acknowledgments

Without the tireless help of Dr. Steve L. Karman, Jr., this dissertation would not have been possible. I have learned how to create a useful product, apply my knowledge to Computational Fluid Dynamics (CFD) research, and make my own way as a member of our CFD community. Being invited to work with the MVCE Technical Committee at AIAA, present papers at conferences, and mentor graduate students have been invaluable experiences as I have learned my new role as Dr. Vincent Charles Betro.

This dissertation was also made possible by all the wonderful people at The University of Tennessee SimCenter at Chattanooga: National Center for Computational Engineering, both faculty and fellow graduate students, who have shown me that I am capable of understanding anything that I set my mind to, given me the opportunity to be a part of their research community, and been willing to help me when I have hit road blocks in both my understanding of computational fluid dynamics and growing into the person and professional I am now.

Abstract

Due to the myriad of geometric topologies that modern computational fluid dynamicists desire to mesh and run solutions on, the need for a robust Cartesian Mesh Generation algorithm is paramount. Not only do Cartesian meshes require less elements and often help resolve flow features but they also allow the grid generator to have a great deal of control in so far as element aspect ratio, size, and gradation. Fully Anisotropic Split-Tree Adaptive Refinement (FASTAR) is a code that allows the user to exert a great deal of control and ultimately generate a valid, geometry conforming mesh. Due to the split-tree nature and the use of volumetric pixels (voxels), non-unit aspect ratio meshing is easily achieved. Nodes are not generated until the end which mitigates tolerance issues. The tree is retained coherently, and viscous layers may be inserted in the space between the geometry and the Cartesian mesh before it is tetrahedralized. FASTAR uses tree traversal to determine neighbors robustly, and with the tetrahedralization of only a small amount of space around the geometry, sliver cells and inverted elements are avoided. The code uses Riemannian Metric Tensors to generate geometry-appropriate spacing and is capable of adaptive meshing from a spacing field generated either by the user or from solution data. FASTAR is a robust, general mesh generator that allows maximum flexibility with minimal post-processing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Literature Review

1.1 Importance

In order to solve modern computational fluid dynamics problems, rapid, efficient, and high quality mesh generation must be a point of emphasis. The mesh generation phase of a computational simulation project can be tedious and time consuming, and often a mesh which will give reasonable results may take many iterations to be produced manually, costing a great deal of man hours.

Automation in structured mesh generation is attainable if the blocking process and topology creation can be automated. This becomes difficult when the geometry gets complex with narrow gaps and irregular shaped regions, the square peg in a round hole syndrome.

While commercial meshing packages such as Pointwise [Pointwise, 2010] and HARPOON [CEI, 2010] exist, these require the user to be highly knowledgeable of the software package and can become cumbersome due to time spent on manual manipulation and the iterative nature of mesh refinement. In order to allow those running simulations to concentrate on modifying the flow solver and interpreting results, the best option is to have a program which generates quality meshes with little user input. Additionally, the limitations

on the types of meshes these programs can generate are great, and while some geometries are impossible to fully recover, others simply take an incredible amount of time to initialize.

1.2 Types of Meshing

There are four principal types of mesh generation: Cartesian Hierarchical, Delaunay, and Extrusion. In the subsections below, the rationale, history, and uses of these approaches are detailed.

1.2.1 Extrusion

Extruding a mesh from a given geometry simply means adding layer after layer of cells, working out from a surface mesh generated on the geometry. This technique originated in structured meshing, but it can be accomplished in unstructured meshes using triangle-based cells [Lohner et al., 1988], [Lohner, 1996] or quadrilateral-based cells [Blacker et al., 1991], depending on the nature of the geometry and desired final mesh. Often, viscous meshes composed of prisms lend themselves best to most viscous flow calculations [Ito et al., 2004].

While extrusion is quite useful in prismatic meshing and viscous layer extrusion as seen in Figure 1.1, in practice it is very difficult and time consuming to generate an entire volume mesh in this fashion due to crossing elements, smoothing each layer, and attempting to fit cells together in the region where the advancing fronts collide [Karman, 1995 (NASA)]. The difficulties created by stacking cells in tight areas and having to avoid cell edge intersections, such as the crossing seen in Figure 1.2, are among the reasons that this technique is usually limited to adding small regions of cells [Wang et al., 2004]. Thus, Cartesian mesh generation is preferable to fully extruded methods on complex geometries [Aftosmis et al., 1997].

1.2.2 Overset

Another option for generating meshes on large, complex geometries is the multi-block overset mesh. Multiple different overlapping structured or unstructured meshes are created around a given geometry, and boundary information is periodically exchanged through some method of conveyance, such as Chimera interpolation, as the flow solver searches for a solution

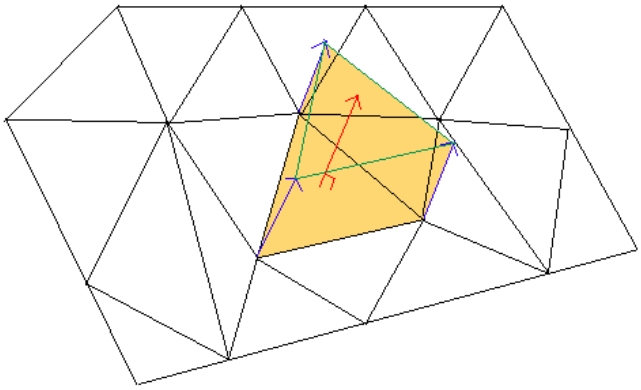


Figure 1.1: **Extrusion From Triangular Facets** – Surface normals are produced from the centroid, seen in red, and these are then translated to each vertex of the triangle. Then, nodes are distributed using a geometric progression factor along each normal, and these nodes are connected (green lines) to form prism elements (peach colored).

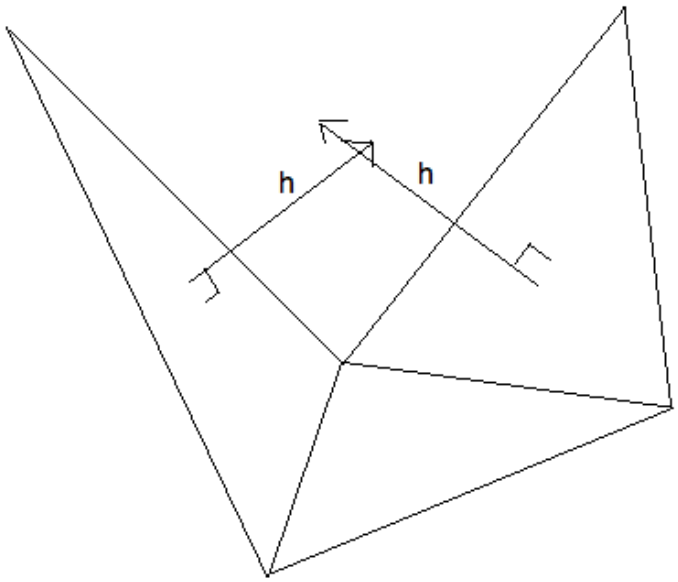


Figure 1.2: **Crossing During Extrusion From Triangular Facets** – Surface normals are produced from each element with magnitude h, which is the first layer height, and they become entangled. This would cause elements to overlap and make the mesh invalid.

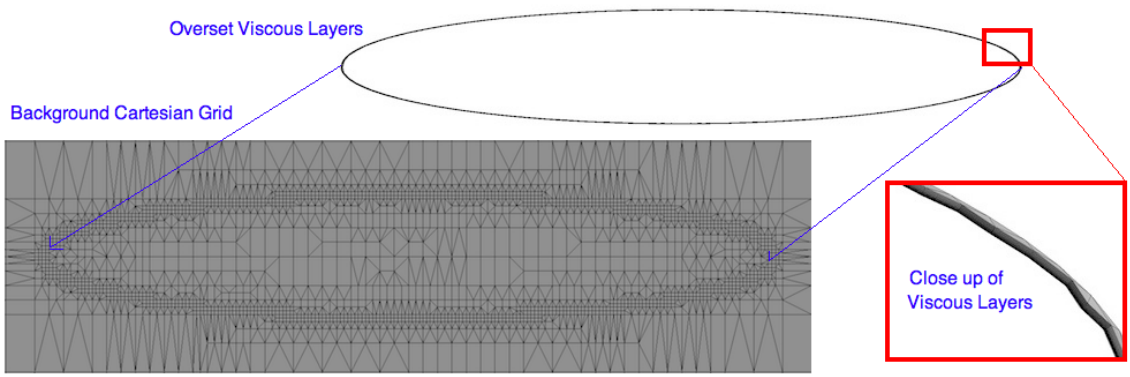


Figure 1.3: **Unstructured Overset Mesh Creation Process** – Viscous layers are generated on an ellipsoid and then superimposed onto a volume mesh which was generated respecting the geometry spacing to make interpolation more accurate.

[Djomehri et al., 2003]. While this can be done very practically, many geometries are not well suited to structured meshes, and unstructured overset meshes are relatively new in general use.

As shown in the work of O'Shea, et al., the use of unstructured overset methods requires that special boundary conditions be used at the mesh interfaces, such as the immersed boundary method [O'Shea, et al., 2008]. One option is to use the approximate volume of the background mesh cell at the interface that is inside the computational domain of the overset mesh and balance the fluxes appropriately. Another is to use source terms to satisfy the no-slip or free-slip conditions when computing the body force components.

The main drawback to overset meshes is that the laws of conservation cannot be enforced at the interpolated points, which can lead to inaccurate solutions. Also, the process is not fully scalable to be implemented in parallel due to high communication needs for interpolation.

1.2.3 Delaunay

Another type of mesh that is relatively easy to generate on large, complex geometries is a Delaunay tessellation [Delaunay, 1934]. In two dimensions, this is defined as a triangulation wherein a circumcircle is placed around each potential element. Then, points are added such that in finality no point from any other triangle is within the circumcircle of a given triangle

[Ruppert, 1995]. In three dimensions, this technique involves generating a tetrahedralization wherein no tetrahedron's circumsphere contains a point not on that tetrahedron. While many tetrahedral meshes can be generated robustly, so long as the geometry need not be preserved, they require a great deal of post-processing, generate far too many elements, and in order to conform to user parameters, require a structured mesh first be generated in the background or have some other method of determining spacings through the generation process [Liu et al., 2000].

The main pitfall of this method arises when attempting to recover the boundaries given by the original surface tessellation once the initial tetrahedralization is completed [Joe, 1992]. Boundary segments can always be recovered in two dimensions [Shewchuk (Triangle), 1996], and one can usually reconnect the boundary facets as faces of tetrahedra in three-dimensions, but the Delaunay nature of the mesh is often degraded. Using the algorithm developed by Bowyer and Watson [Watson, 1981], if a boundary facet is to be made part of a given tetrahedron and this causes the Delaunay criterion to be invalidated, the offending tetrahedra are deleted and the convex hull remaining is re-meshed, if it can be. Often, this cannot be accomplished without the insertion of Steiner points, which then changes the surface tessellation [George et al., 1991], [Weatherill, 1994]. On the other hand, Lawson's algorithm [Lawson, 1977] deals with this situation by flipping edges until all boundaries are recovered, regardless of the Delaunay criterion.

TetGen, which was written by Hang Si, uses the Bowyer-Watson algorithm and Steiner points, thus often meaning that the boundary cannot be exactly recovered [Si, 2006]. While this can make for simple, fast generation, it creates a great deal more points and elements than a Cartesian scheme, and viscous solutions are hard to develop with tetrahedral elements. An example of a Delaunay tetrahedralization from Pointwise [Pointwise, 2010], which has a similar Delaunay algorithm implemented, on an outlet nozzle of a high speed catamaran FSF-Seafighter can be seen in Figure 1.4.

1.2.4 Cartesian

As posited by Chalasani, the best metric for a good mesh is a successful flow solution [Chalasani et al., 2005]. This is especially true for Cartesian Hierarchical schemes that often

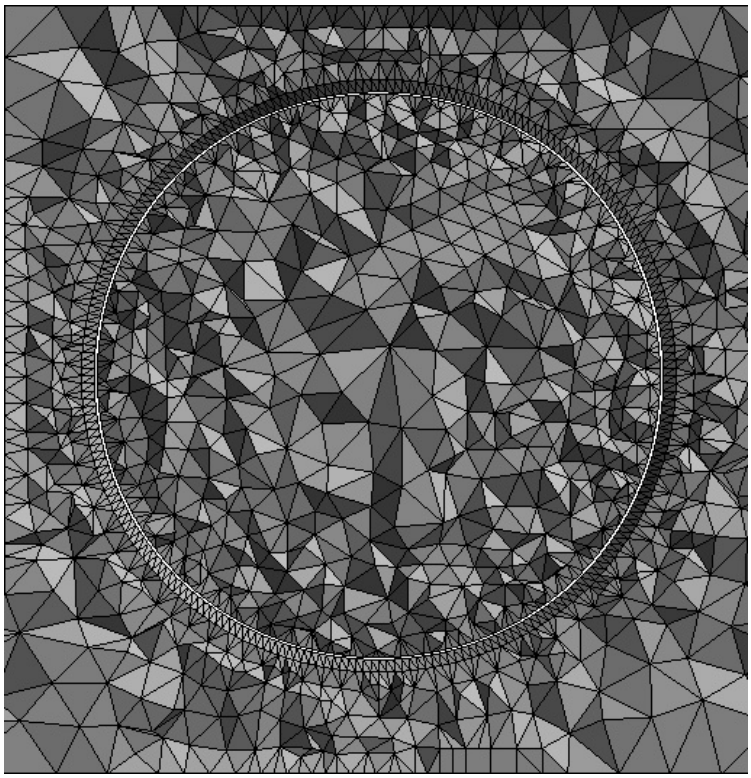


Figure 1.4: **Delaunay Tetrahedralization** – A Delaunay tetrahedralization on an outlet nozzle from a high speed catamaran FSF-Seafighter.

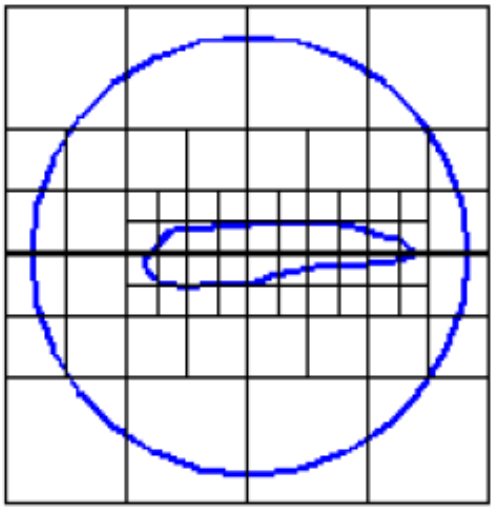


Figure 1.5: **Hierarchical Cartesian Refinement on an Airfoil** – The above illustration shows how a root cell can be subdivided recursively (to varying degrees in different areas based on desired resolution) to create a series of squares that provide a background grid. These will help to eventually resolve the airfoil and create elements around it of both a triangular and quadrilateral nature [Karman et al., 2007].

result in polyhedral element types. In this case, grid metrics can be difficult to obtain in the form of condition numbers and aspect ratios, which are difficult to consistently compute due to the random shapes of the elements. The mesh can be converted into the four basic types to compute metrics, but it is often not of the same quality. However, Cartesian meshes often yield excellent solutions as they can refine in pertinent areas [Dawes, 2006] and often can allow cells to be aligned with flowfield characteristics and not smear these characteristics in regions of high gradient [Berger et al., 1998].

Thus, Cartesian meshing provides the most flexibility with the fastest mesh creation [Thompson et al., 1999]. Pointwise [Pointwise, 2010] is capable of creating curvilinear body fitted structured meshes using transfinite interpolation and elliptic smoothing. However, in order to create an unstructured mesh on any given geometry, the preferred method is Cartesian hierarchical refinement which merely creates a supercell and subdivides it recursively until the desired spacing is reached at the finest level of refinement which should be in the vicinity of the geometry. This principle is seen in Figure 1.5, where the most refined cells are in the vicinity of the airfoil itself.

Additionally, having a Cartesian Hierarchical mesh allows for the natural creation of a tree, often omni-tree or octree, which aids in searching for elements as well as eventually choosing areas of refinement for adaptation and dynamic meshing. In Figure 1.6, the options for refining a hexahedron are provided. The first row shows one directional refinement, which is used to create the split-tree, which allows for rapid searching of the tree since only one direction needs to be compared to determine which branch to proceed down. While this allows for refinement in only one direction, isotropic refinement can still be achieved, just over multiple steps. This method preserves tree integrity, which means that the same number of refinements on any given root cell will yield cells of the same volume. This attribute makes tree-traversal and neighbor searching rapid due to the fact that only one direction needs tested to determine node placement based on the cut parameter of the cell.

The second row in Figure 1.6 shows two directional refinement. An omni-tree approach may use this type of refinement, as well as one directional and isotropic refinement. While the omni-tree provides maximum flexibility for refinement, it causes issues with tree coherency since all cells that are at the same level of refinement may not have the same volume if each cell is not filled to its limit with adopted children of its children. This can be time consuming and inefficient.

The third row in Figure 1.6 is isotropic or octree refinement. While this is the fastest type of refinement due to its consistent nature and guaranteed neighbor orientation [Yerry, 1984], [Shephard, 1991], it only allows unit aspect ratio (or the aspect ratio of the original super cell) meshes to be created.

Once the overall mesh is created around the geometry, the cells and/or nodes that are not inside the computational domain must be turned off and the leftover space between the volume mesh front and the geometry must somehow be converted into acceptable elements. One manner of accomplishing this is to intersect the geometry facets with the cells and create a cut cell with the remaining edges and faces of the intersected cell that are inside the computational domain; this allows a stand-alone mesh for use with multiple solvers [Dawes, Kellar, et al., 2009]. Another alternative is to have the mesh generator coupled with the flow solver in order to mitigate the need for reforming elements from the leftover cut cell pieces (general cutting), thus simply supplying the solver with an approximate area

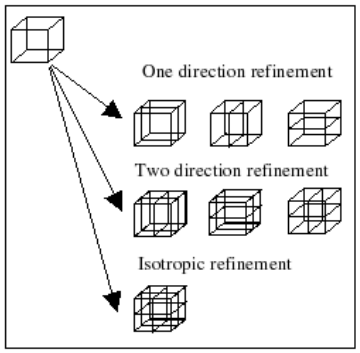


Figure 1.6: **Subdivision Refinement of a Cartesian Cell in Three Dimensions** – Refinement options in three dimensions. The choice used to create the split-tree is one-directional refinement.

or volume of the leftover space inside the domain from the original cells for calculations [Ishida et al., 2009].

Some have chosen to use adaptive-precision arithmetic to avoid tolerance issues in general cutting [Shewchuk (Adaptive), 1996], but the fact remains that irrational point values and other anomalies can still occur and be a detriment to the final mesh quality. While general cutting has been used by many mesh generators [Coirier et al., 1996], the product of general cutting is often sliver cells (cells created from volume mesh remnants that often have nearly zero volume) that need to be optimized before a solution run, as seen in Figure 1.8.

Others have created non-unit aspect ratio, body conforming, Cartesian methods using different means. For instance, Park is able to generate body-conforming Cartesian meshes with general cutting, but the control of refinement is generally relegated to post-solution knowledge [Park, 2008], which makes adding in flowfield characteristics such as air-water interfaces difficult. His method involves creating the Cartesian volume mesh and using Shewchuck's adaptive-precision arithmetic to create the cut elements. He then uses Riemannian Metric Tensors to do adaptation, but only after the original mesh has been created.

Wang is able to generate meshes on non-watertight geometries [Wang et al., 2004], but only after they have been shrink-wrapped by CFD-Viscart [CFD-VISCART, 2010]. This allows him to create the volume mesh and use projection to fill the cut area with impunity

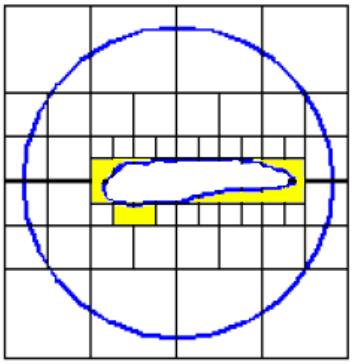


Figure 1.7: **Space Needing Filled Between Voxel Front and Geometry** – The portion of the volume shaded in yellow must be discretized to make a body conforming mesh.

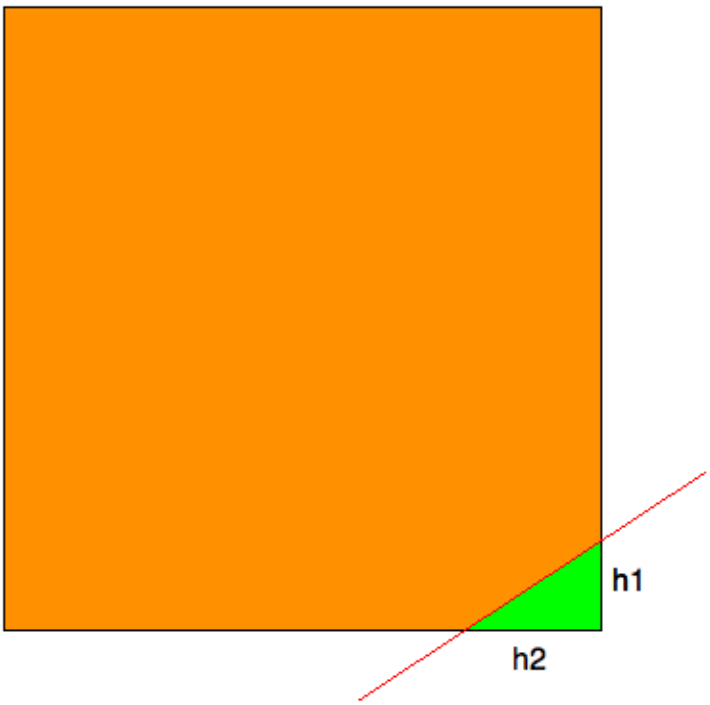


Figure 1.8: **Sliver Cells from General Cutting** – Sliver cells are portions of the cut cell inside the computational domain which often have very small areas. In the above figure, the orange area is outside the computational domain and the green inside. If h1 and h2 are very small, the area can approach machine epsilon.

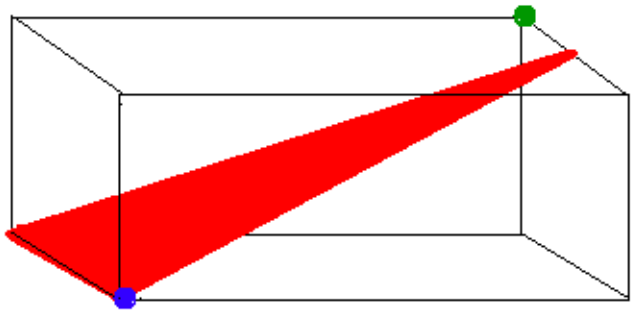


Figure 1.9: **An Extent Box** – This extent box for the red geometry facet was constructed by using the minimum x, y, and z values of the three geometry facet nodes to create the lower corner point (seen in blue) and the maximum x, y, and z values to create the upper corner point (seen in green).

due to modifications being made on the user geometry. However, due to the omni-tree nature of the generation scheme, the tree is not preserved coherently.

1.3 Riemannian Metric Tensors

Riemannian Metric Tensors allow the user to specify spacing parameters and a set of basis vectors that describe the shape of an ideal element. This results in a 3×3 tensor, which can be used by each of the three edge vectors to obtain a metric length. This determines whether an element should be refined or not and in which direction.

In the simplest case, one can choose spacing parameters in the x, y, and z directions (based on a coordinate system aligned with each element). Then, after creating a set of orthogonal basis vectors on each element (its coordinate system), a tensor is created by (1.1). This tensor is then applied within the extent box of the geometry facet, shown in Figure 1.9.

The constructs in (1.1), (1.2), and (1.3) are used to construct the uniform Riemannian Metric Tensor from the desired spacing parameters.

$$M = [R]\,[\lambda]\,[R]^{-1} \tag{1.1}$$

$$R = [\vec{e}_1\, \vec{e}_2\, \vec{e}_3] \tag{1.2}$$

$$\lambda = \begin{bmatrix} h_1^{-2} & 0 & 0 \\ 0 & h_2^{-2} & 0 \\ 0 & 0 & h_3^{-2} \end{bmatrix} \tag{1.3}$$

To apply this tensor to a given Cartesian element, the x, y, and z edge lengths of the current cell are determined ($\vec{AB}$, seen in Figure 1.10) and used in (1.4) to determine if the cell needs to be refined in the x, y, or z direction respectively (seen in Figure 1.11). The scalar metric length d that is obtained in each direction indicates a need for refinement if greater than 1.0 or acceptable refinement if less than or equal to 1.0.

$$d = \sqrt{\vec{AB}^T M \vec{AB}} \tag{1.4}$$

1.3.1 Use in Cartesian Mesh Generation

Riemannian Metric Tensors are a method of specifying a grid spacing in a given region. The use of both the metric tensor structure and metric length were defined clearly in a paper by Huang [Huang, 2004]. The ability to control refinement in specific regions of the mesh using the geometry as a harbinger [Shepherd, 2009] is the primary purpose of using Riemannian Metric Tensors.

However, often, the spacing needed by the geometry is only half the issue. There can be flowfield features that need to be resolved or multi-fluid interfaces that require something that resembles viscous packing in their vicinity. Since there might not be geometry facets in the area of these flowfield features and Riemannian Metric Tensors are defined only for a given area of the volume, spacing boxes are given in terms of metric tensors that shape regions of the mesh within their given extent box [Vyas et al., 2009]. Additionally, these metric tensors can be used for adaptation and smoothing, as seen in a paper by Karman [Karman et al., 2007]. One can compute an extent box for a given area of the volume mesh and supply an x, y, and z spacing to be applied within that extent.

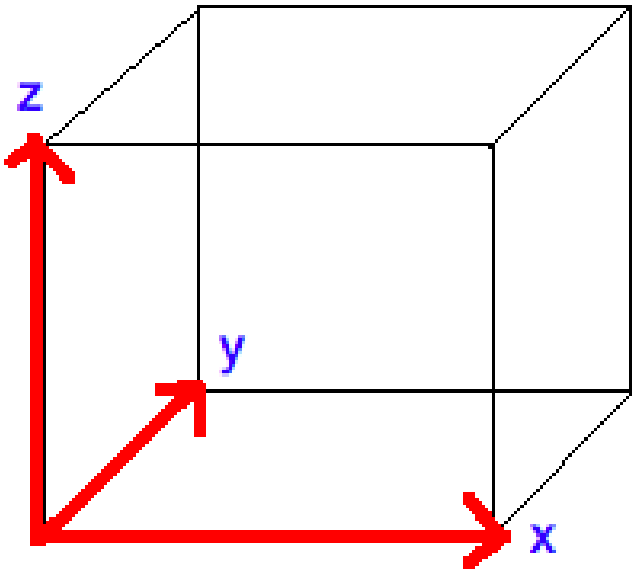


Figure 1.10: **Edges to Be Tested on a Cell** – In the case of Cartesian meshing, since all cells are lined up with the Cartesian axes, the components of $\vec{AB}$ are only non-zero in the direction of the edge in question. The edge vectors are denoted in red.

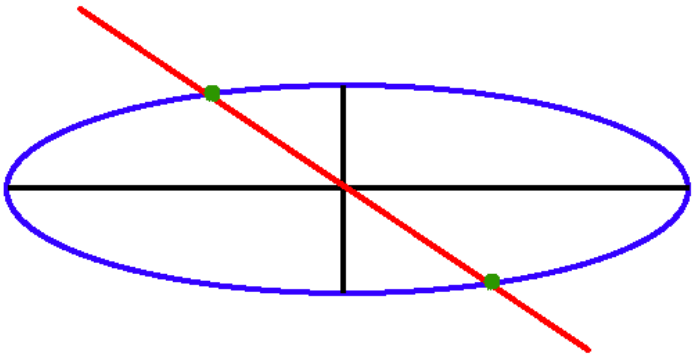


Figure 1.11: **Tensor Derived Ellipse Acting on an Edge** – Consider that the ellipse in the picture represents the spacing in x and y directions. As it is applied to an edge which intersects its extents, the green points represent the part of the edge that would satisfy the metric length of one, thus showing further refinement is needed.

The spacing box is especially useful in that it creates a very small file with only one entry, thus speeding the processing of the Riemannian Metric Tensor down the tree, since it only needs to be done once and will contain many cells. This is particularly useful in the hydrodynamics case presented in this dissertation due to the fact that at the air-water interface, which is not demarcated by geometry, needs to have refinement three orders of magnitude smaller than anywhere else in the mesh. This can be achieved in the initial generation, and the region can be very clearly defined. An example can be seen in Figure 1.12.

1.3.2 Use in Mesh Adaptation

In order to generate an accurate solution, one must have a sufficiently resolved mesh, specifically in the areas of high solution gradients [Bibb et al., 2006]. The process of adaptation is that an initial mesh can be generated and then refined and coarsened through use of Riemannian Metric Tensors and an element-based refinement/coarsening algorithm [Morrell et al., 2007]. However, this requires a mesh be generated on the geometry first, often consisting of tetrahedra and requiring many more nodes and elements than necessary. Since one goal of mesh generation is to reduce the number of points and elements necessary to obtain a solution, thereby speeding the process, the use of non-unit aspect ratio elements can be a boon to the user [Lahur et al., 2001] as can the fact that the initial generation is done with the adapted spacing in mind.

Riemannian Metric Tensors can be constructed through feature or adjoint-based adaptation, as well as through user-defined spacing needs, and are useful due to their construction using directional control.

1.4 Addressing Mesh Generation with `FASTAR`

Fully Anisotropic Split-Tree Adaptive Refinement (`FASTAR`) Hierarchical Cartesian Mesh Generator is a highly automated, highly recursive split-tree scheme and allows the user to control many facets of the mesh generation process. It also gives the user many different choices in the type of final product it generates to better cater to time constraints and

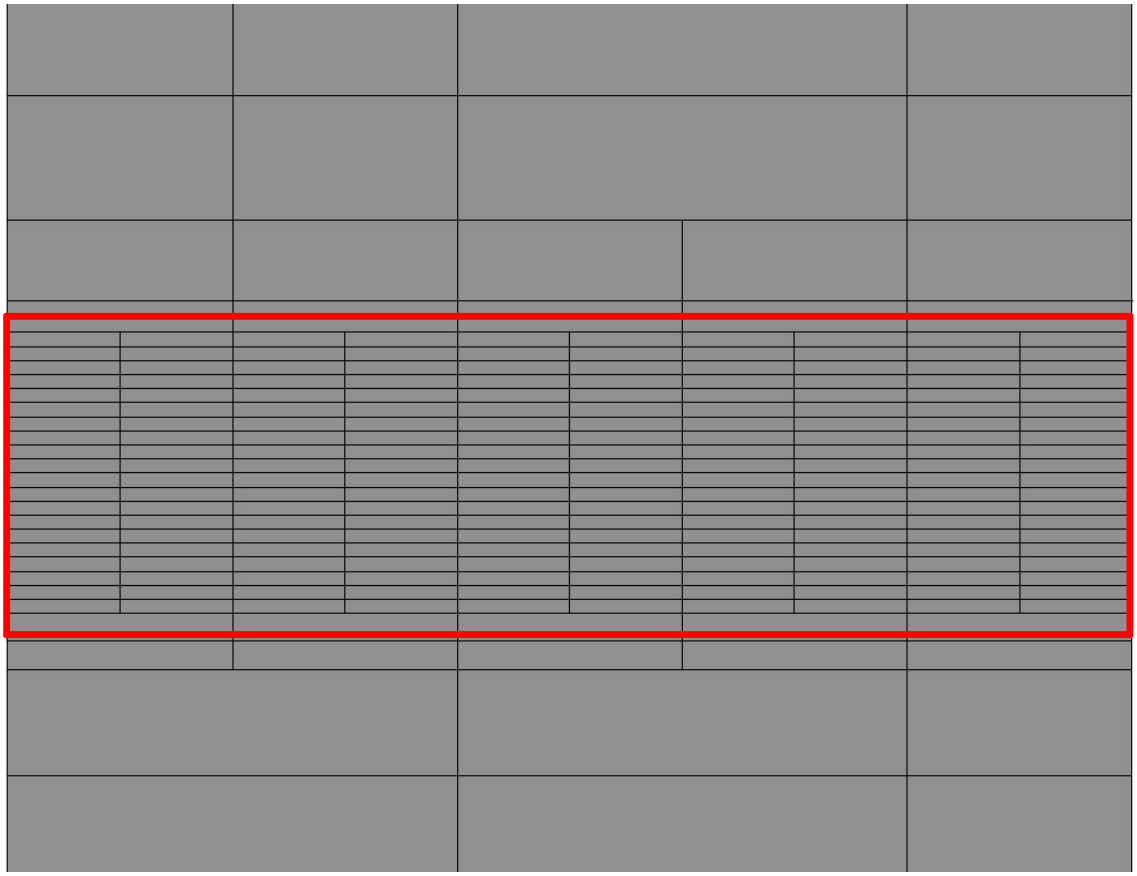


Figure 1.12: **Spacing Box for Air-Water Interface** – This section of a Seafighter mesh shows the usefulness of spacing boxes. The spacing of 10^{-3} on the air water interface (shown in the red box) was achieved simply by setting up an extent box in the area and specifying the appropriate spacing in the z direction.

solver needs. The basic mesh generation approach uses an anisotropic hierarchical Cartesian polyhedral mesh to discretize three-dimensional domains. While portions of this approach have been published in the past (see Section 2.4 for more detailed information) and are available in some commercial mesh generation packages, the additions of the unique cell deletion approach, tetrahedralization in lieu of sliver cells, the ability to generate viscous layers, the ability to generate overset meshes, and the flexibility of spacing make `FASTAR` unique and powerful. Additionally, when performing design optimization, the need for re-meshing automatically based on areas of high gradient between flow solver runs becomes paramount [Karman, 2004], and `FASTAR` is equipped to handle multiple types of mesh adaptation.

Using a fast split-tree approach in conjunction with advanced deletion and stitching algorithms, a large mesh may be generated on the most complex of geometries in a rapid fashion while still preserving the geometry as supplied by the user, and this capability allows those working on fluid simulations as well as fuel cell theory, plasmas, and electricity and magnetism simulations to more easily generate large meshes and adapt them to improve their solutions.

1.5 Chapter Summaries

In Chapter 2, the `FASTAR`-specific terminology used in this dissertation is explained and the rationale and concepts behind its implementation are defined. Also, the contributions of `FASTAR` to the field of mesh generation are detailed, as are the specific uses of Riemannian Metric Tensors.

In Chapter 3, the initial Cartesian mesh generation process with Riemanian Metric Tensors and implementation of quality constraints is detailed, as well as optional viscous layer insertion and overset capabilities. Additionally, this chapter discusses the use of cell marking and a tetrahedral mesh generator to merge the volume mesh front and the geometry. Issues of implementation are also detailed as well as more rationale as to `FASTAR`'s construction.

In Chapter 4, results are displayed that were obtained on several geometries with differing user inputs. Cases of adaptation and user defined spacing are examined, as are flow solver results on a prolate spheroid mesh. Some of these cases also utilize viscous insertion and give viscous solutions. Also, overset meshes and stitched meshes are presented. Most of the graphic images used in this section were created using the visualization software Fieldview [Fieldview, 2010].

In Chapter 5, conclusions about the uniqueness, usefulness and robustness of `FASTAR` are presented. Also, discussions of future work include the addition of a Lawson's algorithm tetrahedral mesh generator, parallelization, and dynamic meshing.

In the Appendix, one will find a partial derivation of the equations from Section 2.3.2, created using Maple 8 [Maple, 2002].

Chapter 2

Technical Approach of FASTAR

In order to understand from whence the FASTAR algorithm came, why it was chosen, and how it differs from and improves upon current technologies, this discussion of the benefits and drawbacks of the FASTAR approach was compiled.

2.1 Geometries

FASTAR is equipped to handle any geometry, usually a surface triangulation created on a database supplied by some Computer Aided Drafting (CAD) package or meshing program. The only stipulation is that the geometry must be watertight (no holes) and the facets must be stored such that their nodes are in counterclockwise order which yields a right hand rule surface normal that points into the computational domain.

One interesting feature of FASTAR is that it retains the user's original geometry exactly. This is unlike P_HUGG [Karman et al., 2008] where cells are generated around the geometry using a user-defined spacing and the intersections of the geometry facets and these cells create new boundary facets that will become part of the final mesh but are not identical to the original geometry. For instance, if the finest spacing is larger than that of the geometry facet and merging is enabled, boundary facets are merged within a cell, and the resolution of the geometry can be downgraded, as seen in Figure 2.1. Similarly, if a facet is much larger than a given cell, it will be broken apart and pieces of it will be used to create faces of the cut cells in the vicinity.

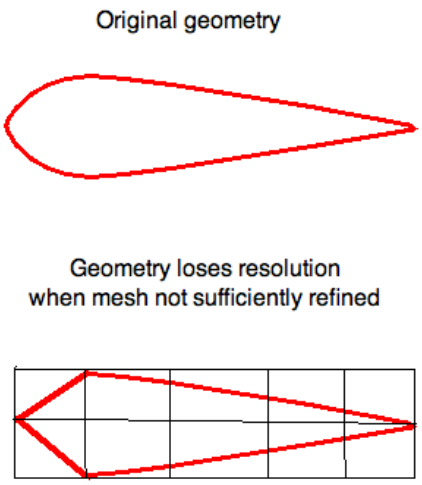


Figure 2.1: **Geometry Deterioration** – In `P_HUGG`, if the user does not sufficiently refine the mesh, the geometry facets that make up the curved leading edge of the NACA 0012 will be merged to make a poor quality airfoil. In `FASTAR`, this is ameliorated by coupling the exact original geometry with the Cartesian elements around it using a Delaunay tetrahedralization.

In this dissertation, a cube within a cube, a sphere within a cube, and a simple cube were used for testing various facets of the program and for explanatory purposes. All of these were convex shapes, which allowed for viscous insertion, and the size of the meshes generated allowed for shorter debugging times. Also, a 6:1 prolate spheroid (ellipsoid) was constructed and used for running test cases as well as determining viscous and overset capabilities. A NACA0015, a Drag Prediction Workshop (DPW) IV cargo plane, a high speed catamaran FSF-Seafighter, and a NASA SDT2-R4 rotor were meshed in order to show the robustness of `FASTAR` even in the case of concave, complex geometries.

It should also be noted that using `FASTAR` one can break apart geometries, mesh them with individual constraints, and then glue them back together as seen with the DPW IV and the Seafighter in Chapter 4. This empirical partitioning of a geometry can allow for vastly different meshes to be generated on portions of the geometry as well as help with issues of sharp corners or tight areas for refinement.

2.2 Principles Behind FASTAR

2.2.1 Cutting Technique Comparison

While the initial mesh generation phase of `FASTAR` is comparable to `P_HUGG` [Karman et al., 2008], the divergence between the two begins with allowing non-unit aspect ratio elements and the application of quality constraints. Also, the voxel deletion method used in `FASTAR` is very different from both the projection cutting method used in Dr. Steve Karman's original three-dimensional serial code `HUGG` [Karman, 2004] and the general cutting used in `P_HUGG` [Karman et al., 2008].

In `HUGG`, the tree is traversed to check for intersections between the geometry facets and each cell. Boundary cells are then created by projecting the exposed Cartesian faces to the geometry. Those new projected faces are matched up with the defined geometry parts to identify the mesh boundaries [Karman, 1995 (NASA)], as seen in Figure 2.2. While this is more rapid than general cutting, it presents the problem seen in Figure 2.3, which is intended to be the view from the top of seven geometry facets coming together at a vertex. The top face of the cell highlighted in yellow is at the vertex of the seven different geometry

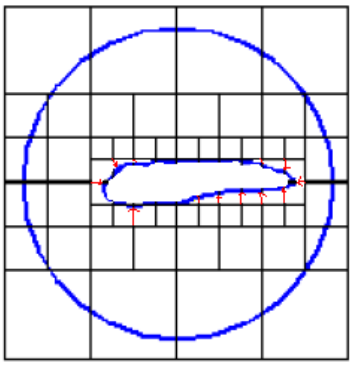


Figure 2.2: **The Projection Process** – As is done in `HUGG`, the cell front is projected onto the geometry, as seen by the red arrows.

parts, six of which have an equal projection area. Mesh edges in the vicinity should be moved to line up with the geometry edges. However, there is no point in the mesh that can have more than six edges emanating from itself. Another problem occurs if a cell is smaller than the spacing between two geometry segments and lies between them, it is not guaranteed to stretch and fill the gap.

Thus, while the projection cutting process can be very rapid compared with general cutting [Karman, 1995 (AIAA)], it is a process with definite limitations in three dimensions. While general cutting is more robust, the main limitation with general cutting is that often such small pieces of a voxel remain that there can be tolerance issues with determining if a node is distinct or not. Sliver cells are often created and require optimization to fix. The yellow cell portions seen in Figure 2.4 would become cut cells, and the red portions would be deleted in general cutting.

The manner in which `FASTAR` overcomes this cutting impediment is to simply remove all cells within a certain distance of the geometry and discretize the leftover space (see Figure 1.7). This can be accomplished by both tetrahedralization and viscous extrusion. The main downfall of using a Delaunay tetrahedralization to discretize the space is that it often cannot exactly recover the boundaries. In `FASTAR`, only meshes with unaltered surface

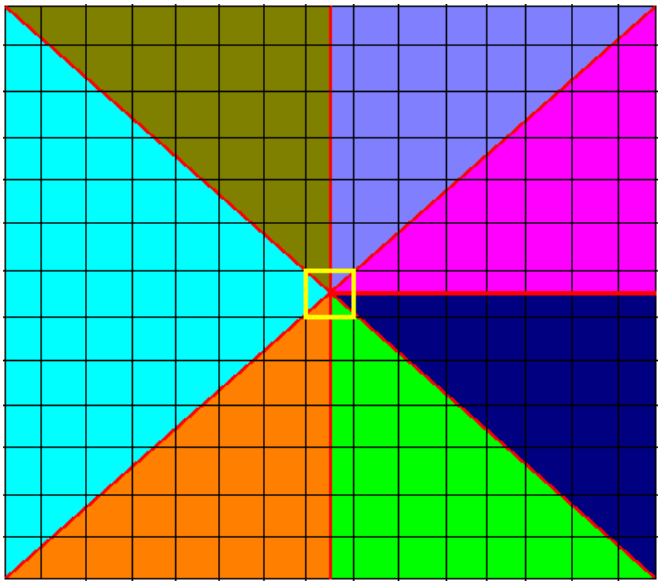


Figure 2.3: **Limitations of Projection Cutting** – The top face of the hexahedral cell to be distorted to fit into the scope of the light blue geometry facet is highlighted in yellow. However, the vertex at the center still cannot be modeled by Cartesian elements since a given point can have at most six edges emanating from itself.

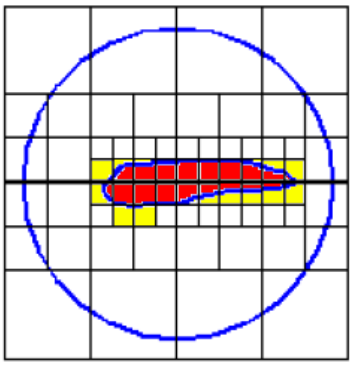


Figure 2.4: **The Cutting/Deletion Process** – Cell parts marked in yellow must either be made into their own cells (`P_HUGG`), deleted in order that the cell front may be projected to the geometry (`HUGG`), or have their space filled by another type of cell (`FASTAR`).

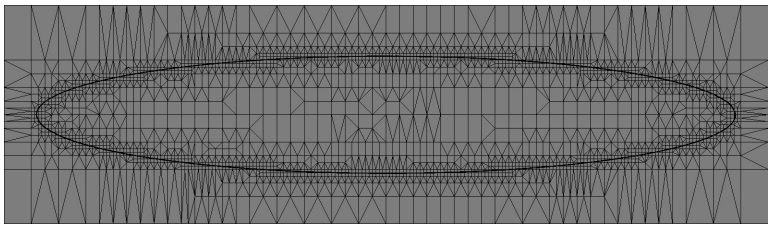


Figure 2.5: **Unstructured Overset Mesh** – This shows the final overlaid mesh, with the thick dark line being 35 viscous layers with a geometric progression factor of 1.15 and initial spacing of 1.0e-05.

triangulations can later be stitched together, so a Lawson's algorithm approach [Lawson, 1977] that fully respects boundaries by only flipping edges and not adding Steiner points is being explored to mitigate this issue.

2.2.2 Viscous Extrusion Ability

The technique of extrusion is employed in `FASTAR` to insert viscous layers directly from geometry facets on convex geometries. This results in prismatic elements which are excellent for modeling viscous flows. This method is only used for convex geometries, where intersections are rare (except with inviscid boundaries, which are discussed in Section 3.8). In order to make this meshing more robust, normal smoothing and intersection checking would need to be added to the algorithm.

2.2.3 Overset Ability

In `FASTAR`, a volume mesh is generated within the root cell constructed from the extents of the farfield boundaries, using Riemannian Metric Tensors to determine spacing to achieve parity with the geometry. In the case of an overset generation, this mesh is not cut and acts as the background mesh on which the geometry and viscous layers are overlaid. This approach can be seen in Figure 2.5, and this method is further discussed in Section 3.9.

2.3 Use of Riemannian Metric Tensors

Riemannian Metric Tensors are used by mesh generators to control the spacing of elements created based on parameters set by the user, the input geometry, and flow field characteristics. These tensors can be created in multiple fashions, including using Karman's Spacing Field code [Varghese, 2009], which computes the information discussed in Section 2.3.3. Additionally, `FASTAR` provides the user the opportunity to select the normal spacing by which the tetrahedra needed to construct three dimensional tensors are created from the two dimensional geometry facets.

The next subsections will go into detail describing how Riemannian Metric Tensors are used in `FASTAR` as well as the manner in which they are constructed. The condition number and rank of the matrices discussed in this section were generated by MATLAB2009b [MATLAB, 2009] as were the ellipsoid diagrams representing the manner in which the Riemannian Metric Tensors project onto the current elements to reshape them.

2.3.1 Uniform Tensor Construction

In the simplest case, the user can choose to let the minimum or maximum spacing parameters of the geometry facets (or any other desired spacing parameter) determine a uniform tensor. Using these spacing parameters and an orthogonal set of basis vectors for each element (its coordinate system), this tensor is then applied to all facets on the given boundary, within the extent box of each geometry facet.

Equations (1.1), (1.2), and (1.3) are used to construct the uniform Riemannian Metric Tensor from the desired geometry spacing parameters. Once the tensor is constructed, it is passed down the tree until the finest level cells within its extents are found. Effectively, this entails beginning at the root cell and comparing the centroid coordinate in the direction of the cell's split with the coordinates of the extent box. If the extent box is at all contained in the extent of the cell, the recursive algorithm continues to search down that branch, through all the cell's children and their children, until cells are found that have no children and contain all or some of the extent box. Figure 2.6 illustrates the method by which a tensor is passed down the tree.

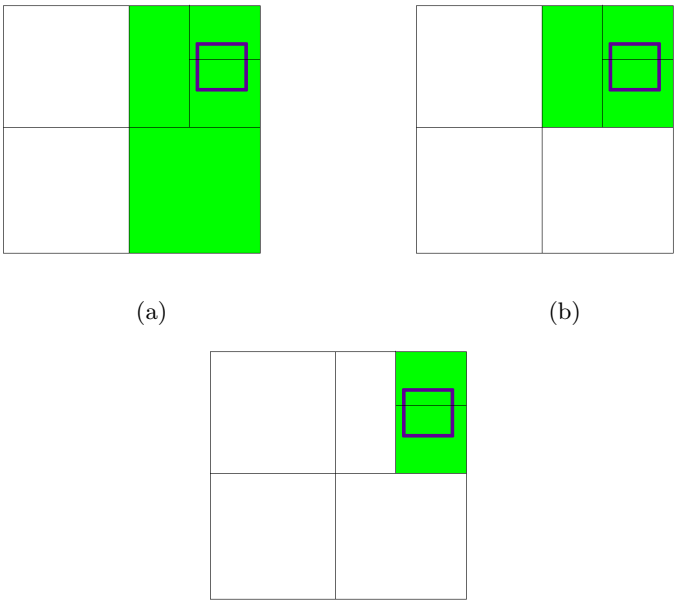


Figure 2.6: **Passing RMT Down the Tree** – A tensor is only applied to the finest element within its extents (purple box). (a) First, the root cell is checked for containment, and then the facet is passed down the green branch, in which it is contained. (b) Then, that child is checked for containment, and then the facet is again passed down the green branch, in which it is contained. (c) Finally, the extent box includes both cells marked in green and is applied to both.

Using (1.4) for the cases presented in this dissertation, a metric length greater than or equal to 1.01 is used as the demarcation of acceptable refinement for all tensor applications. Also, if all three directions require refinement, the order of precedence is x refinement, then y refinement, then z refinement. This could be reordered with no effect on the mesh created; the main area where order of precedence is noticeable is when quality constraints are applied.

2.3.2 Construction from Tetrahedron

In the case where the user wishes to refine in the extents of each geometry facet with a spacing commensurate with that facet, a tetrahedron is constructed by creating a surface normal from the centroid of a geometry facet, assigning it a length (either user defined, based on curvature or intersection of geometry, or based on attempting to create an equilateral tetrahedron), and using its endpoint as the fourth vertex of the tetrahedron, as seen in Figure 2.7.

Equation (2.1) is used to determine the unique Riemannian Metric Tensor (M) for each geometry facet-based tetrahedron that gives its edges a metric length of unity. The matrices that compose (2.1) are listed separately as (2.2), which is a symmetric positive definite matrix, thus only the upper triangular portion needs to be solved for, giving us a system of six equations. Matrix (2.3) is a combination of all the edge lengths of the given tetrahedron.

$$(P_j - P_i)^T [M] (P_j - P_i) = \vec{1} \tag{2.1}$$

for $1 \leq i < j \leq d$ where $d = 3$ since this process occurs in three dimensions.

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \tag{2.2}$$

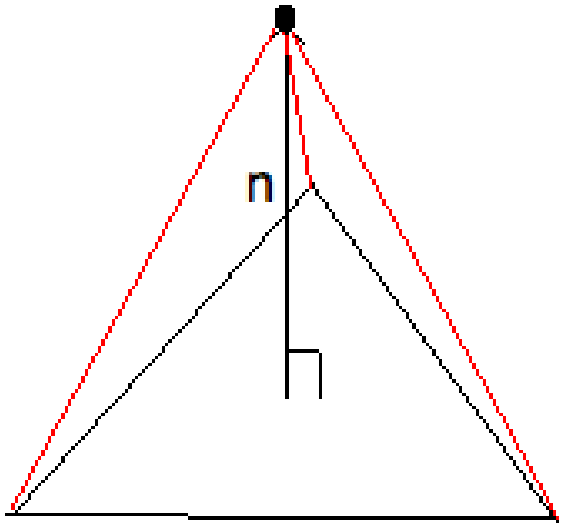


Figure 2.7: **Construction of a Tetrahedron From a Facet** – This facet has had a surface normal projected from its midpoint of length n. This will create a fourth vertex, from which a tetrahedron (in red) can be constructed.

$$P_j - P_i = \begin{bmatrix} (x_2 - x_1) & (x_3 - x_1) & (x_4 - x_1) & (x_3 - x_2) & (x_4 - x_2) & (x_4 - x_3) \\ (y_2 - y_1) & (y_3 - y_1) & (y_4 - y_1) & (y_3 - y_2) & (y_4 - y_2) & (y_4 - y_3) \\ (z_2 - z_1) & (z_3 - z_1) & (z_4 - z_1) & (z_3 - z_2) & (z_4 - z_2) & (z_4 - z_3) \end{bmatrix} \quad (2.3)$$

Using Maple [Maple, 2002], as seen in Appendix 6.1, (2.1) was reduced to an $A\vec{x} = \vec{b}$ system, with components shown below.

$$A =$$

$$\begin{bmatrix} (x_2 - x_1)^2 & 2(x_2 - x_1)(y_2 - y_1) & 2(x_2 - x_1)(z_2 - z_1) & (y_2 - y_1)^2 & 2(y_2 - y_1)(z_2 - z_1) & (z_2 - z_1)^2 \\ (x_3 - x_1)^2 & 2(x_3 - x_1)(y_3 - y_1) & 2(x_3 - x_1)(z_3 - z_1) & (y_3 - y_1)^2 & 2(y_3 - y_1)(z_3 - z_1) & (z_3 - z_1)^2 \\ (x_4 - x_1)^2 & 2(x_4 - x_1)(y_4 - y_1) & 2(x_4 - x_1)(z_4 - z_1) & (y_4 - y_1)^2 & 2(y_4 - y_1)(z_4 - z_1) & (z_4 - z_1)^2 \\ (x_3 - x_2)^2 & 2(x_3 - x_2)(y_3 - y_2) & 2(x_3 - x_2)(z_3 - z_2) & (y_3 - y_2)^2 & 2(y_3 - y_2)(z_3 - z_2) & (z_3 - z_2)^2 \\ (x_4 - x_2)^2 & 2(x_4 - x_2)(y_4 - y_2) & 2(x_4 - x_2)(z_4 - z_2) & (y_4 - y_2)^2 & 2(y_4 - y_2)(z_4 - z_2) & (z_4 - z_2)^2 \\ (x_4 - x_3)^2 & 2(x_4 - x_3)(y_4 - y_3) & 2(x_4 - x_3)(z_4 - z_3) & (y_4 - y_3)^2 & 2(y_4 - y_3)(z_4 - z_3) & (z_4 - z_3)^2 \end{bmatrix}$$

$$x = \begin{bmatrix} M_{11} \\ M_{12} \\ M_{13} \\ M_{22} \\ M_{23} \\ M_{33} \end{bmatrix}$$

$$b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Using singular value decomposition and back substitution, one solves the system to produce a metric tensor. This metric tensor gives a spacing field like those which are shown in Figures 2.8 and 2.9, which are represented by ellipsoids with the major axes having the

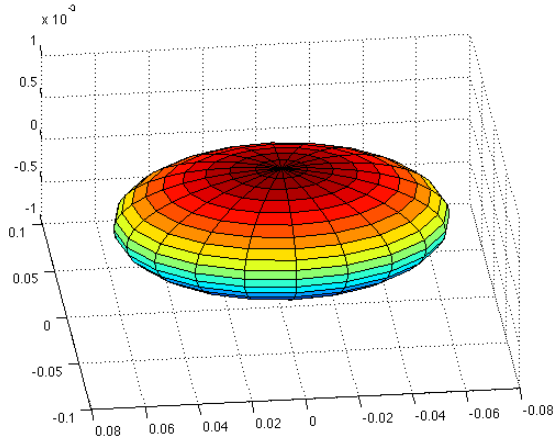


Figure 2.8: **Symmetric Ellipsoid Representation of Riemannian Tensor Scaling** – This ellipsoid represents the scaling applied by the Riemannian tensor with x and y scaling set at 0.0625 and z scaling set at 0.00625, as seen in the cube in Figure 4.2.

lengths of the eigenvalues of M. This field effectively scales the current element edge as a matrix scales a vector through multiplication.

2.3.3 Construction from Solution Information

In the case of feature-based adaptation, the user inputs a mesh with solution data and chooses to adapt based on one or more of several flowfield functions, such as density, velocity, pressure, helicity, and vorticity. Then, ∇f from (2.4) is determined by finding the gradient of that flowfield function between sets of two nodes. Also, p from (2.4) can be set to a parameter ≥ 1.0; setting it above 1.0 helps assure that the mesh will refine in regions other than shocks or other major discontinuities that would normally dominate the mesh.

C in (2.4) is a user-defined measure of equidistribution of error, such that the metric lengths for each edge in a control volume are re-determined in order to equally distribute the change in spacing over the whole mesh and achieve the desired resolution. Once these new lengths have been determined by (2.5), the new tensors are created using (1.1) for each element to drive the proper refinement in the adapted mesh to be generated. The λ_i values in M are replaced by $\frac{1}{\Delta d^2_{new}}$ and the basis vectors for R are constructed using the direction

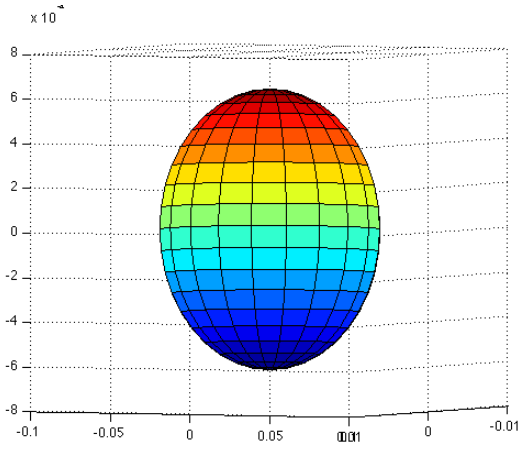


Figure 2.9: **Asymmetric Ellipsoid Representation of Riemannian Tensor Scaling** – This ellipsoid represents the scaling applied by the Riemannian tensor with x scaling set at 0.0625, y scaling set at 0.00625, and z scaling set at 0.00625. This can be rotated about in three space to match with the orientation of the tetrahedron.

of ∇f for the principal direction and creating the remaining two orthogonal directions using element edges and cross products.

$$\|\nabla f \cdot \hat{e}\| \Delta d^p = C \tag{2.4}$$

$$\Delta d_{new} = \sqrt[p]{\frac{C_{avg}}{\|\nabla f \cdot \hat{e}\|}} \tag{2.5}$$

In this way, a new set of Riemannian Metric Tensors can be stored at nodes or elements, with the extents determined by those of the elements or control volumes associated with the creation of the tensor. Thus, when creating the adapted mesh, these assure finer spacing in areas of flowfield phenomena regardless of their proximity to the geometry.

In creating a spacing box using empirically derived spacing, (1.1) is used, with the user specified spacings being substituted for each λ_i. The set of basis vectors is created from the three orthogonal corner vectors emanating from the extent box.

2.3.4 Rationale Behind use of Singular Value Decomposition

Singular value decomposition has been the method of choice for solving such a system due to the ill-conditioning of the A matrix [Karman et al., 2007]. However, it is instructive to see how this system becomes so ill-conditioned and near rank deficient.

The tetrahedron shown in Figure 2.10 is a nearly ideal element, with three sides of length 1 and three of length $\sqrt{2}$. Using (2.1), we obtain (2.6).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -2 & 0 & 1 & 0 & 0 \\ 1 & 0 & -2 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & -2 & 1 \end{bmatrix} \tag{2.6}$$

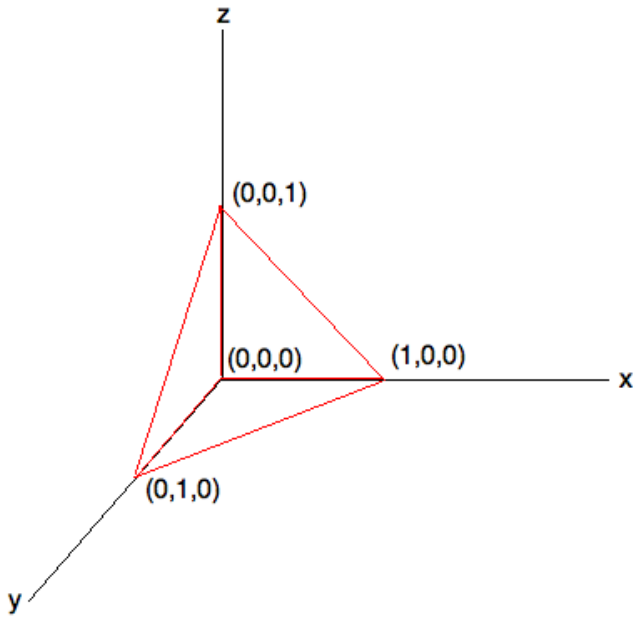


Figure 2.10: **Nearly Ideal Tetrahedron** – This tetrahedron is nearly ideal, with three edges of length 1 and three of length $\sqrt{2}$. It produces a well-conditioned A matrix.

Now, it is instructive to look at some of the metrics of (2.6), which will be referred to as matrix A, an $m \times n$ matrix. Consider the condition infinity norm and condition number of the matrix, given in (2.7) and (2.8).

$$\| \cdot \|_\infty = 1 \leq \overset{max}{j} \leq n \sum_{i=1}^{m} |a_{ij}| \tag{2.7}$$

$$\mathcal{K}(\mathcal{A}) = \|A\|_\infty \cdot \|A^{-1}\|_\infty \tag{2.8}$$

For this A matrix, one obtains $\mathcal{K}(\mathcal{A}) = 4 \cdot 1.5 = 6$, which is not so poorly conditioned. The rank of matrix A is 6 and it produces a well-conditioned system, which could be solved by a variety of methods.

Now, consider the same tetrahedron, but with the normal distance (z coordinate) reduced to 10^{-3}, as seen in Figure 2.11. Using (2.1), we obtain (2.9).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 10^{-6} \\ 1 & -2 & 0 & 1 & 0 & 0 \\ 1 & 0 & -2 \times 10^{-3} & 0 & 0 & 10^{-6} \\ 1 & 0 & 0 & 1 & -2 \times 10^{-3} & 10^{-6} \end{bmatrix} \tag{2.9}$$

Again, using (2.7) and (2.8), one finds that here $\mathcal{K}(\mathcal{A}) = 4 \cdot 1.0 \times 10^6 = 4 \times 10^6$. While the rank of matrix A is still 6 in exact arithmetic, it is nearing a rank deficient situation in the third, fifth, and sixth columns (the first, third and fifth rows are nearly identical), thus the conditioning becomes poor. This increases as the normal spacing shrinks, thus realizing the need to use singular value decomposition and back substitution to deal with solving these nearly rank deficient matrices.

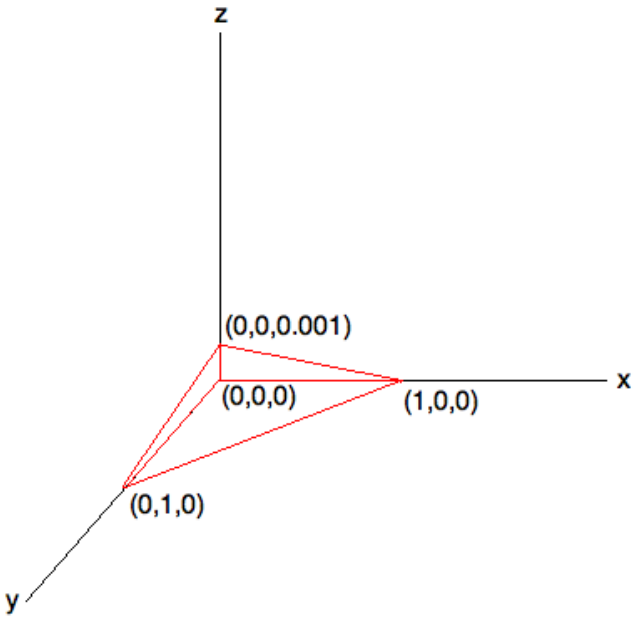


Figure 2.11: **Tetrahedron Created by 0.001 Normal Spacing** – This tetrahedron is representative of a spacing field wherein the geometry is preserved but the normal spacing is user-supplied and small. It produces an ill-conditioned A matrix.

2.4 Two-Dimensional Basis of FASTAR

`FASTAR` is loosely based on the two dimensional work of Luo and others [Luo et al., 2008], with some small exceptions. First, extra care must be taken with three-dimensional topologies due to crossbar scenarios, where neighboring voxels at the same level of refinement have been refined in different directions and their faces do not match, arising from the split tree. Also, Luo's generator uses octree and is thus only capable of generating unit aspect ratio meshes. Another difficulty in fully porting this approach to three dimensions was that while a Delaunay triangulation is guaranteed in two dimensions, it is not in three dimensions. Of course, since the geometry is retained as part of the final mesh (unlike general cutting schemes), the original surface mesh must be of good quality, and preferably unstructured [Beatty et al., 2008].

Chapter 3

Implementation of FASTAR

In Figure 3.1, a flow chart is shown giving the overall process of FASTAR in order to assist in the understanding of its implementation. The numbers beside each branch are referenced in the text as (n), where n is the step number shown.

3.1 Cartesian Hierarchical Terminology

In order to create a mesh around any given geometry a root cell must be constructed. The essence of Cartesian hierarchical meshing is simply subdividing this root cell until a desired number of child cells is reached, and this resolution varies in different areas of the mesh depending on local refinement criteria supplied by the user. Before it is sensible to discuss mesh quality issues, it is advisable to discuss the data structure that is used to create this root cell: a voxel.

While the concept of a voxel is most often used in discussing computer graphics rendering, it is an appropriate term for Cartesian hierarchical meshing as well. The word comes about by combining the words volumetric and pixel, which here contains all the information about one specific portion of the mesh and its surroundings, down to a given resolution, just like a pixel. More importantly, the term voxel is used since the voxel itself knows nothing of its physical coordinates; rather, it simply contains information regarding its relational position to other voxels [Voxel, 2007]. The specific pieces of information the VOXEL data structure contains are described next.

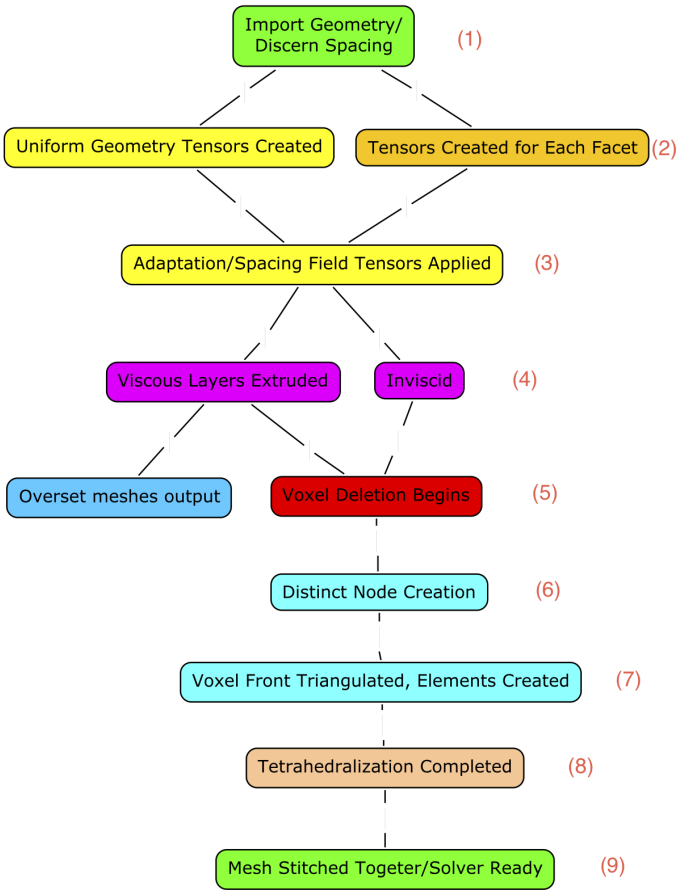


Figure 3.1: **Flow Chart of FASTAR Process**

Each voxel contains the index of its mother voxel, and the root voxel is the only voxel without an initialized mother. It also contains a list of its two children, if they exist. It contains a cut flag which is later used to determine whether the voxel is fully in the computational domain, fully outside the computational domain, or intersecting the geometry edges. A split variable is given in the structure, wherein the direction of refinement is held for ease of tree traversal, and the physical coordinates of the high and low corner points of the voxel are stored for construction of the physical points after the mesh is generated and for relative tree traversal purposes.

3.2 Benefits of Using the Tree Structure

Hierarchical Cartesian mesh generation has become very popular because of the ease with which connectivity can be developed using a tree structure as well as the fact that with some work and a priori knowledge cells can be aligned with flow field features. The tree is useful not only in applications like adaptive or dynamic meshing but also in traversing the space to be discretized with ease while generating the mesh.

Most Cartesian Hierarchical techniques discussed in Chapter 1 utilize the octree. The benefits of an octree are derived from both the simple nature in which all elements are subdivided equally and the fact that neighboring elements are always placed in a consistent manner. However, the inability to make a leaner, non-unit aspect ratio mesh made this the wrong tree choice for `FASTAR`, since it disallows anisotropic refinement in areas with drastically different directional spacing tensors.

As seen in the work of Karman and Domel [Domel et al., 2000] as well as Wang [Wang et al., 2004], another popular tree technique is to use an omni-tree. This type of tree allows for multiple directions of refinement, which makes non-unit aspect ratio mesh generation possible. However, when using omni-tree, the need to adopt grandchildren arises when the original children have been refined. This is the case when a voxel is not refined isotropically and its children are refined in such a way that the new stencil fits within one of the omni-tree types of refinement. At this point, the children of children become the actual children of the voxel and some or all of the original children are discarded. For a visual representation

of this phenomenon, see Figure 3.2, wherein the initial refinement is seen in red and the secondary refinement is seen in blue. This is effectively an isotropic refinement in two steps and would remain two levels in a split-tree. However, in an omni-tree, these blue refined voxels would be adopted as children of the original voxel and the red refinement would be discarded, making two levels of refinement into one level of refinement.

Additionally, as each tensor is passed down the tree, each voxel that has not been refined isotropically has to be traversed again since it may need to be refined in a different direction based on the current facet's metric tensor and have the grandchildren adopted. This fact made this approach particularly ill-suited for the algorithm used by `FASTAR`. For a visual representation of passing a facet down the tree, see Figure 3.3. The extent box of the facet is first passed to the root, and since it is not in the extent of the voxel with the yellow x, that branch is abandoned. Then, the same test occurs on the next level voxel, and the branch with the orange x is abandoned. Finally, when it reaches the next level voxel, the same test is performed, the branch with the purple x is abandoned and the finest level voxel in the facet's extent is located. Based on metric lengths, the voxel will be refined in the y direction (denoted by the red line). This process continues until the metric length computed is below 1.01. This is the same for split-tree and omni-tree, except that when the next facet is passed down in split tree, it will proceed directly to the finest level voxel in its extent, whereas in omni-tree, it will have to stop at the green voxel and determine if it needs refined in the x direction, do so, and then adopt the grandchildren.

Thus, the split-tree technique used in `FASTAR` has many advantages. First, since voxels can only be refined in one direction, there is no need to adopt grandchildren and reshuffle parentage as voxels are refined differently by different tensor specifications. Similarly, only one direction needs evaluated in passing a nodal location down the tree–that of the split of the voxel. Also, while one can create an non-unit aspect ratio mesh in this fashion, a unit aspect ratio mesh can also be obtained simply by limiting the aspect ratio to 1.0 and thus forcing each voxel to refine in all three directions through refinement of its children. Lastly, the tree can be traversed coherently, since the level of a voxel is always simply the ratio of its size in a given direction to that of the root voxel, and neighbors can still be found that are not on the same level.

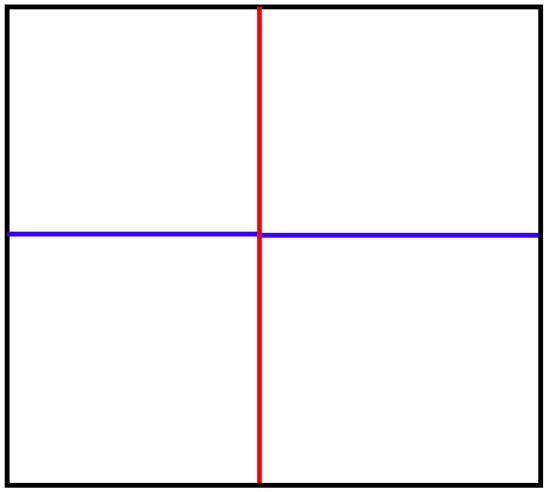


Figure 3.2: **Adoption of Grandchildren Using an Omni-Tree** – The initial refinement is seen in red and the secondary refinement is seen in blue. This isotropic refinement in two steps is left unchanged in split-tree, but these grandchildren are adopted in omni-tree and the red refinement is discarded.

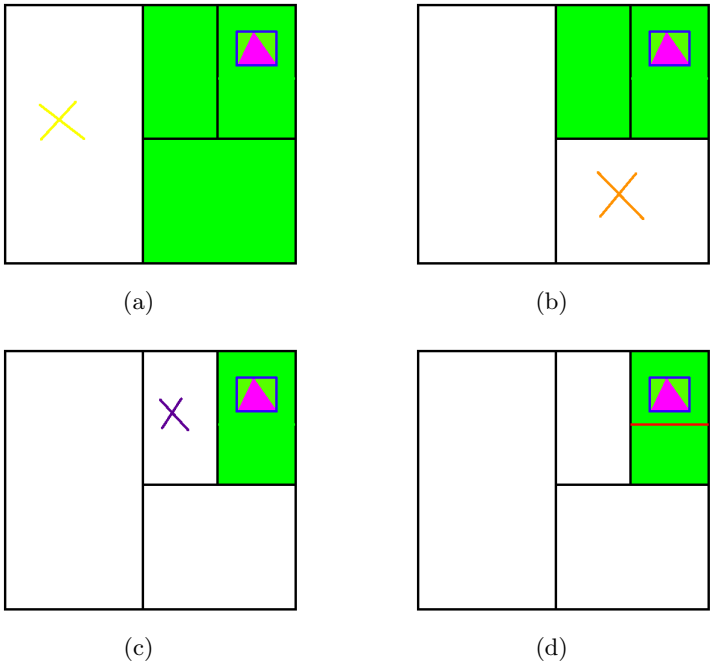


Figure 3.3: **Passing a Facet Down the Tree** – (a) The extent box of the facet is first passed to the root, and since it is not in the extent of the voxel with the yellow x, that branch is abandoned. (b) Then, the same test occurs on the next level voxel, and the branch with the orange x is abandoned. (c) When it reaches the next level voxel, the same test is performed, the branch with the purple x is abandoned and the finest level voxel in the facet's extent is located. (d) Finally, based on metric lengths, the voxel will be refined in the y direction (denoted by the red line). This process continues until the metric length computed is below 1.01.

Finally, the tree was adopted in order to store basic information about the discretization of the space so that, upon determining that the geometry must move or refinement needs to occur in a certain area, one can leave the majority of the mesh as it is and simply operate on that area in question without disturbing any of the other elements. This is a look ahead to dynamic meshing, which is not part of this dissertation, but was integral in development decisions while creating `FASTAR`.

3.3 Geometry Preparation

Geometry is supplied to this mesh generation process in the form of a triangulated surface mesh. This can be supplied in one or more files. The file format can either be a CART3D file [CART3D, 2010] or an ACAD facet file [ACAD, 2010]. Both of these file formats allow for triangles to be grouped together and tagged with a body number. This allows for describing various boundary pieces, such as wing upper surface, wing lower surface, symmetry plane, etc. The right-hand-rule normal vector of each triangle is assumed to be pointing in the direction of the interior of the computational domain. The collection of triangles is also assumed to be watertight with no gaps or overlaps.

The geometry facets are stored in an Octree data structure for rapid searches in geometry queries, such as projection to the closest point on the surface. This storage tree is used only by the geometry functions and is automatically constructed when the facets are input. It gives each facet an extent, and when the need arises to find facets in a given area, an extent test tree traversal is done, like that explained in Section 3.2, and the list of qualifying facets is returned as pointers to the actual facet data structure.

The meshing process refines Cartesian volume elements based on a Riemannian Metric Tensor created from spacing parameters associated with the geometry (as well as adaptation parameters discussed in Section 1.3.2). These spacing parameters are stored at the geometry facet level. Default values are automatically computed based on the overall size of a given body part by computing the approximate hydraulic diameter of that part. This is computed by dividing the surface area of the body part by 1/4th the circumference. In the case where a boundary does not have a curve defining its circumference, such as a single, closed

shell, the extent of the part is measured and the median dimension is used. In either case the computed value is then multiplied by 0.025 and specified for all facets contained in the boundary. This is only an initial value for the spacing parameter. Users have the opportunity to modify the initial values by editing one of the parameter files created by the program.

The user can also designate boundaries where these initialized spacing parameters are adjusted based on approximate surface curvature and proximity to other boundaries. The curvature modification examines the angle between a triangular facet and its neighbors. The four nodes of the two triangles can form a tetrahedron. These nodes are used to compute the radius of a circumscribing sphere, an approximate measure of the radius of curvature. This radius is reduced further by a factor based on the dot product of the surface normal vectors of the adjacent triangles. The spacing parameters for the two facets are then set to the minimum of the current value and this circumscribing radius-based value. If the two triangles were coplanar then the radius would be infinite, so reduction in the spacing parameters would not occur.

For user-selected boundaries an intersection test is also performed to detect gaps and confined regions in the geometry. The surface normal vector for each geometry facet is positioned with the tail at the facet center and the length scaled with the current spacing parameter. If this vector intersects any other geometry facet the spacing parameter is reduced to one-half the smallest intersection detected.

Using the minimum and maximum extents of the geometry, a Cartesian root cell is created, as seen in Figure 3.4. Any voxels created during the volume meshing process that are outside the computational domain will be deleted upon the voxel marking process commencing. This would be the case if the farfield was a sphere or other non-hexahedral shape. This root voxel is stored as having no mother and its high and low corner points are stored in its structure. This entire geometry reading and spacing checking process is seen in (1) in Figure 3.1.

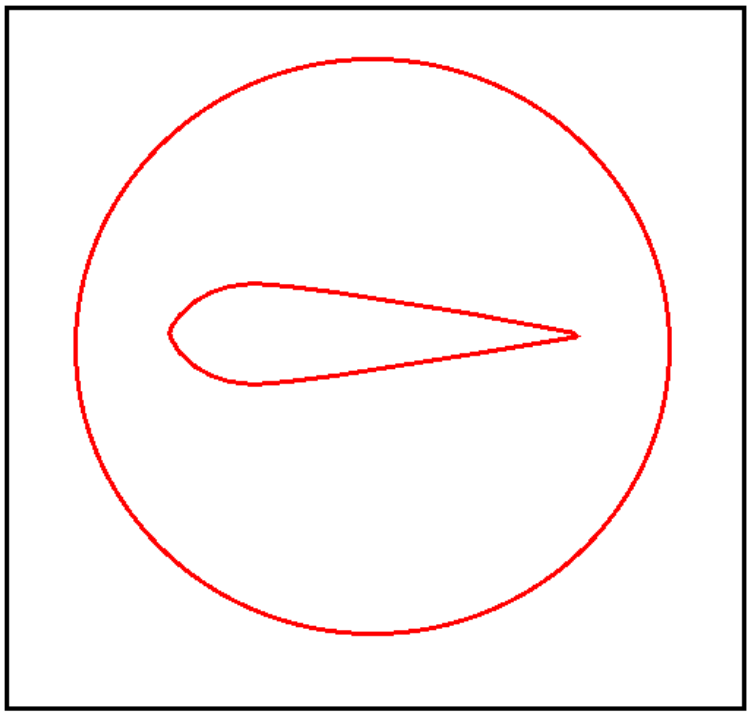


Figure 3.4: **Super Cell Creation About a Non-square Outer Boundary** – In order to begin recursive refinement, a Cartesian super cell is created around the existing geometry, unless the outer boundary is initially square, in which case the super cell and the outer boundary are coincident and there will be no external voxels to be turned off during cutting.

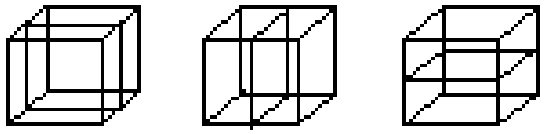


Figure 3.5: **Split-tree Refinement Options** – FASTAR employs split-tree refinement which allows for x, y, or z directional refinement.

3.4 Hierarchical Refinement

As seen in Figure 3.5, `FASTAR` uses a split-tree refinement of each voxel, which allows for rapid neighbor searching within the tree and tree coherency, which is defined here as the ability to discern which level of refinement one is at based solely on the number of children and parents along that branch of the tree. This will be important in the future implementation of a dynamic re-meshing scheme and parallel implementation. Split-tree was also chosen since the use of octree refinement disallows non-unit aspect ratio mesh creation, and an omni-tree approach did not provide for the tree coherency desired in `FASTAR` and also slowed the recursive neighbor search by needing more than one cardinal direction to test in order to continue searching down the tree.

Recursive refinement starts with the root voxel described in Section 3.3. A tetrahedron is constructed from each geometry facet with a user defined normal length creating the apex along a normal from the centroid, as seen in Section 2.3.2. Then, a symmetric, positive definite Riemannian Metric Tensor is created and applied to the each edge in the current voxel, generating a metric length that determines whether or not to refine and in which direction. The equations for the Riemannian Metric Tensor and for the metric length computation are also given in Chapters 1 and 2. Multiple tensor creation methods are used in `FASTAR`, and they are usually coupled with those created by the geometry tetrahedra.

The process of refining a root cell on a sphere can be seen in Figure 3.6. Once the root voxel has been refined, it may not be refined again. Rather, its children are refined in a direction commensurate with the largest Riemannian metric length. If multiple metric lengths are identical within a tolerance, there is a bias toward x-refinement, then y-refinement, then

z-refinement. However, for a symmetric geometry, a symmetric mesh will still be obtained, as seen in Figures 3.7 and 3.6(d).

In the way of an example, consider the series of images in Figure 3.8. The yellow triangle is an arbitrary geometry facet that has become the base of a tetrahedron, drawn in red. Then, the Riemannian Metric Tensor (a slice of which is seen in blue) is applied and metric lengths calculated. Since this one is spherical, the metric lengths are all the same, so x refinement occurs first. Next, the same Riemannian Metric Tensor information is passed to the two children of the root voxel, both of which are within its extents. The metric length of x is now smaller since the spacing in that direction has been cut in half, and y and z are now tied for longest metric length. Thus, the refinement occurs in y. Finally, this process is repeated in the four lowest level children, which are all within the extent of the tetrahedron, and z is the longest metric length, causing refinement in z.

The above tensor creation and application process is seen in (2) and (3) in Figure 3.1.

3.5 Neighbor Searching and Quality Constraints

3.5.1 Neighbor Search Algorithm

Once the initial volume mesh has been created, the need to apply quality constraints arises. While the specific quality constraints are explained below, it is important to distinguish that one type of constraint assures a valid mesh, where all the connectivities exist, and the other type of constraint makes the mesh more solver friendly by assuring that neighboring volumes are not drastically different.

In order to effectively apply these quality constraints, neighbors must be determined. In order to save memory, deal with a constantly changing element count, and utilize the tree, a highly efficient neighbor algorithm was developed. As seen in Figure 3.9, a point is projected in the direction of the desired neighbor, and then a tree traversal determines in which voxel the point lies. This is greatly simplified by the fact that since the split of the voxel is known, only that cardinal direction must be compared with the point to determine which direction to travel in the tree. Also, since we have a user defined minimum spacing,

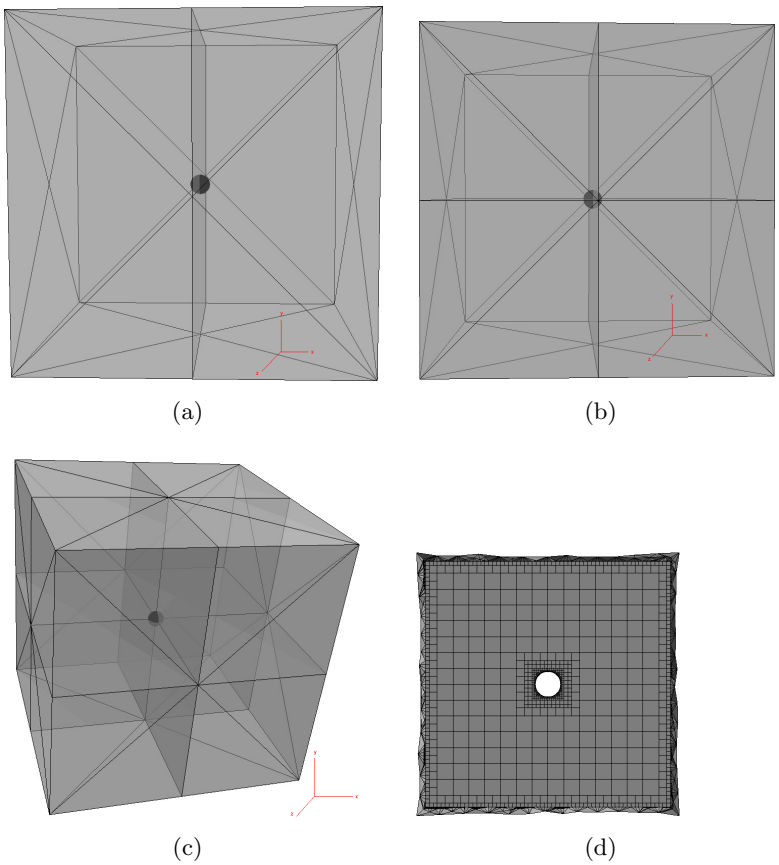


Figure 3.6: **Refining a Root Cell** – (a) Root voxel refined in x direction. (b) Root voxel's children refined in y direction. (c) Root voxel's childrens children refined in z direction. (d) Final mesh after all refinement complete.

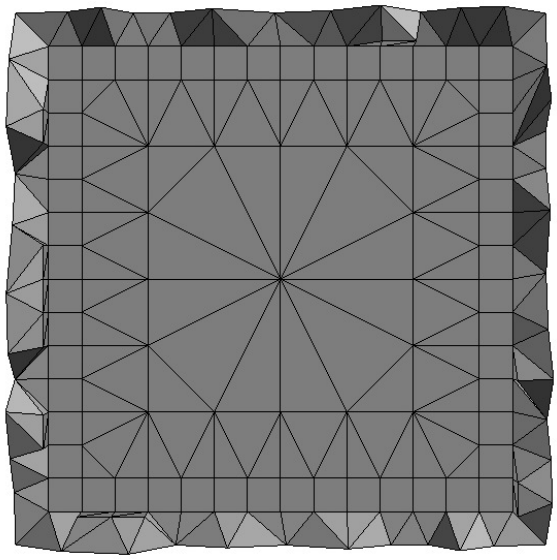
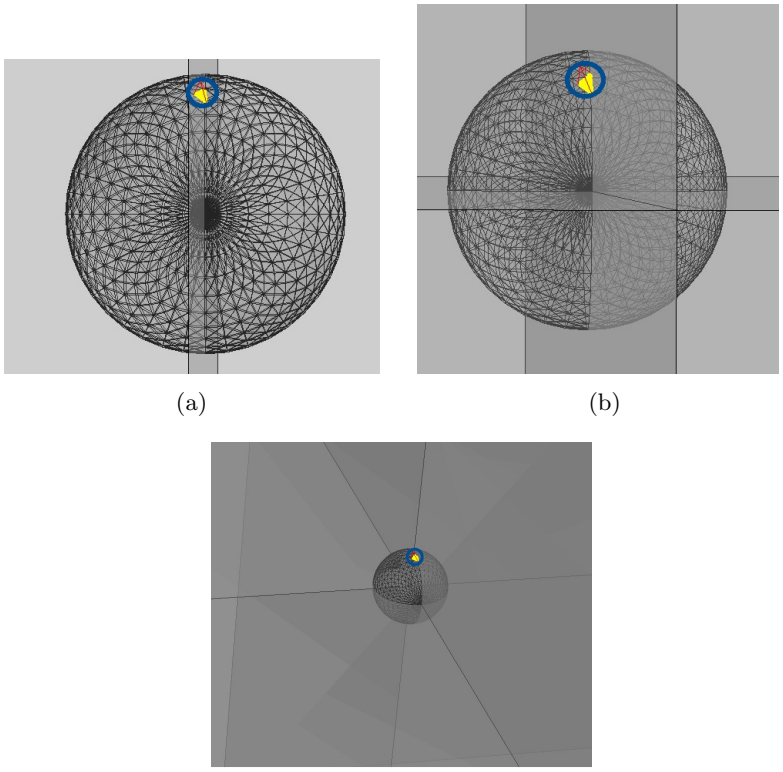


Figure 3.7: **Symmetric Refinement** – Symmetric refinement of a symmetric cube geometry using Riemannian Metric Tensors, converted to the four basic element types.



(a) (b)

(c)

Figure 3.8: **Refining a Root Cell–RMT View** – (a) Root voxel refined in x direction. (b) Root voxel's children refined in y direction. (c) Root voxel's childrens children refined in z direction.

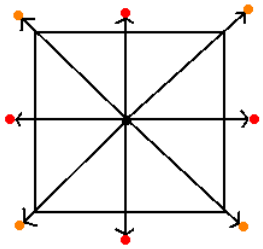


Figure 3.9: **Neighbor Searching** – Neighbor search directions and points for a face of a given voxel.

the tolerance applied to this rare floating point calculation is 1/4th of the minimum spacing, assuring that we always find the voxel we are looking for exactly.

While this algorithm is versatile and will find neighbors despite differing sizes and levels of refinement since it is simply looking for containment, the drawback is that in cases like that seen in Figure 3.10, one may find a neighbor with children. If this is not desired, the algorithm will create four nodes, denoted in red in Figure 3.10, and filter the results of these searches to discern how many children the neighbor has and in what positions. The nodes are created to be 1/4th of the minimum spacing into the neighbor and 1/4th of the minimum spacing in the given cardinal direction away from the face center node. This results in a vector that is $\frac{\sqrt{2}}{4}$ times the minimum spacing, which is larger than the 1/4th that the neighbor routine uses as a tolerance, thus assuring that we get into the right voxel and get the appropriate neighbor.

3.5.2 Desired and Required Quality Constraints

Before discussing which quality constraints are desired and which are required as well as their processes, it is instructive to define each quality constraint. The first quality constraint is aspect ratio, which is simply the maximum edge length divided by the minimum edge length. This is tested by performing this simple calculation and refining in the direction which is longer than desired. If the root cell is a perfect cube, its aspect ratio is exactly 1.0.

The next quality constraint is neighbor refinement. If a voxel has neighbors on opposite sides that are both refined in the same fashion, the voxel in the middle is also refined in that fashion (see Figure 3.11). This is done to make more consistent volumes for the flow

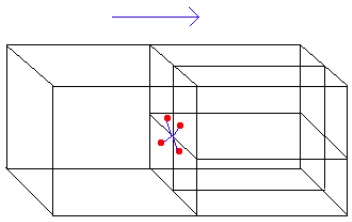


Figure 3.10: **Finding Children of Neighbors** – The algorithm creates four nodes, denoted in red in the above figure, and filters the results of these searches to discern how many children the neighbor has and in what positions.

solver, but it is only done if the two voxels are refined in the same direction. If not, the refinement is not performed due to the propensity to create a crossbar situation, which will be discussed shortly.

Another constraint is gradation. While `P_HUGG` is able to slowly grow the element size as levels change, at the current time, `FASTAR` will only check that the neighboring element is no greater than twice as long in the tangential direction. This is also done to make the mesh more consistent for the flow solver (see Figure 3.12).

In order to assure that the connectivity is correct, the algorithm must identify crossbar neighbors. This is the phenomenon seen in Figure 3.13, and the issue it presents is that neighboring faces do not match up at the nodes. This is found by checking the edge lengths of neighboring voxels and comparing them to those of the voxel in question. If the edge in the first tangential direction is twice as long and the edge in the second tangential direction is one half as long, then this situation is present and both voxels' children are refined in the opposite direction to rectify the situation. This problem is unique to three dimensions.

Finally, there is the quality constraint of too many voxels sharing a single edge seen in Figure 3.14. The stencil for a voxel only contains 27 nodes, of which 18 are mid-face or mid-edge nodes. The issue presented in Figure 3.14 is that there are no quarter-edge nodes, and so the neighboring voxel has no place to reference such a node. Thus, the neighbor must be refined in a direction commensurate with the quarter-edge node.

In order to speed the process to a quality mesh, the desired mesh quality constraints (aspect ratio, neighbor refinement, and gradation) are enforced during element creation by artificially changing the metric length results obtained from the given tensor to garner

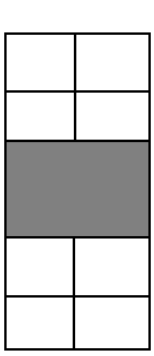


Figure 3.11: **Assuring Mesh Quality–Neighbor Refinement** – A voxel may not have two opposite neighbors that are at a higher level of refinement.

refinement or prevent it. While these quality metrics can be checked recursively along with the required quality constraints, this adds a great deal of time to the longest part of the process of generating the mesh and does little more than over-refine the mesh.

On the other hand, the required quality constraints (crossbar and too many nodes on an edge) are enforced within a recursive refinement loop. This assures that when changes are made to rectify one quality issue, they do not cause another. Once all constraints have been satisfied, aspect ratio is checked one last time, as are the required constraints, and the mesh is ready for voxel deletion and/or viscous extrusion.

The above inviscid quality constraint process is seen in (4) in Figure 3.1, and the option to do viscous extrusion will be explored in Section 3.8.

3.6 Voxel Marking and Distinct Node Creation

3.6.1 Voxel Marking and Flood Fill

The departure from traditional Cartesian Hierarchical mesh generation occurs with the handling of the physical geometry. As posited by Lahur, et. al. in a two-dimensional case [Lahur et al., 2001], the voxels near the boundary are stripped away leaving a voxel front and a geometry or viscous layer front, as seen in Figure 3.15. The area between the two fronts is then tetrahedralized to conform to the geometry supplied, as seen in Figure 3.16, where the inner and outer stitching boundaries around the ellipsoid are green and the tetrahedra are red. Thus, at this point the voxel mesh can be divided into three distinct areas: voxels that are completely inside the computational domain, voxels that are completely outside

Figure 3.12: **Assuring Mesh Quality–Gradation** – Spacing transitions that are too rapid are prevented.

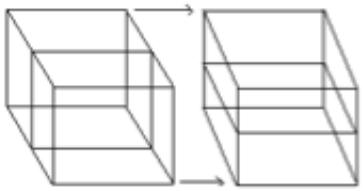


Figure 3.13: **Assuring Mesh Quality–Crossbar** – Crossbar situations lead to neighboring faces not matching up and a mid-face node being left unaccounted for.

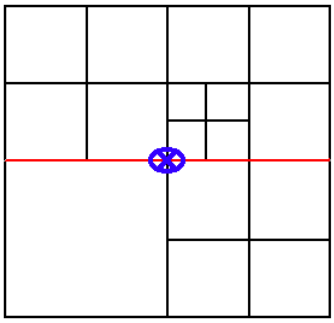


Figure 3.14: **Assuring Mesh Quality–4-to-1 on an Edge** – One mesh level difference is enforced along all edges. The large voxel in the lower left is forced to refine, as the edge marked with the x coming out of the paper is in violation.

the computational domain, and voxels that are intersected by the geometry. Defining these three areas begins with determining the intersected voxels.

In the case of an inviscid mesh, the extents of the geometry facets are expanded by a user-defined spacing parameter (which assures that there is always space between the two fronts) and are passed down the split-tree to the finest levels of the mesh. In the case of a viscous mesh, the nodes of the geometry facets are projected to the tops of the viscous prisms, then the resulting extents are expanded by a user-defined spacing parameter, and they are passed down the split-tree to the finest levels of the mesh.

Because the tree is used, the facet only visits voxels that could contain the facet or be intersected by the facet. First, a test is done to see if either the voxel or the facet are completely within the extents of the other. If this is not the case, it continues by testing for intersection of the geometry facet edges (again, extended by the user-defined spacing parameter) with voxel edges and vice versa. Since all extents have been expanded, this also alleviates the need to test for coplanar facets, since that case is covered by the extension, as seen in Figure 3.17. If any of these tests holds true, the voxel is marked for deletion.

Once the intersection tests are complete, there is a watertight domain consisting of "intersected" voxels that divides the voxels inside the computational domain from those outside of it. Using a flood-fill algorithm, the remaining non-intersected voxels at the finest mesh level can be marked as in or out of the computational domain. First, a non-intersected voxel is chosen at random, and a ray is sent out in each cardinal direction. If the ray passes through an even number of geometric boundaries, the facet is outside the computational domain, and if it passes through an odd number, then it is inside the computational domain. If there is a mix of results, this voxel is disregarded for the moment and another is chosen. Then, the non-intersected voxels that have this marked neighbor and their neighbors down the line can be properly marked using an iterative flood-filling algorithm, and when no more neighbors can be reached with this information, another voxel is chosen and the process repeats until all voxels are properly marked.

The above voxel marking and deletion process is seen in (5) in Figure 3.1, and the option to do overset meshing will be explored in Section 3.9.

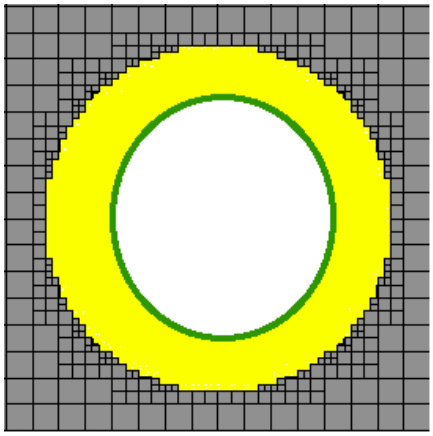


Figure 3.15: **Tetrahedralizable Region** – The green represents the geometry facets or viscous layers, and the yellow represents the area to be tetrahedralized.

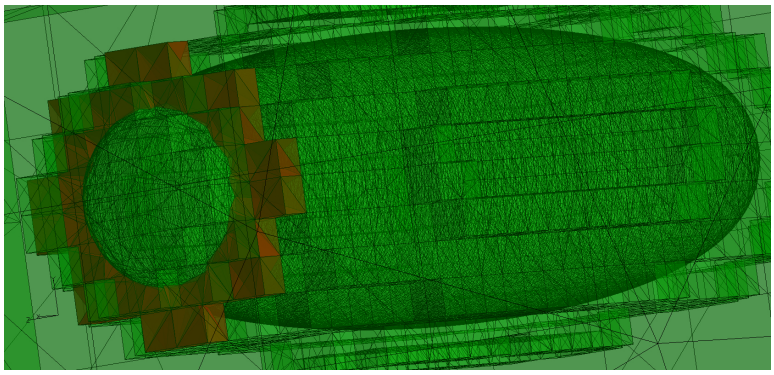


Figure 3.16: **Slice of Tetrahedralized Region from a Prolate Spheroid** – The tetrahedra shown here in red are from a slice of those used to fill in the region between the geometry and the polyhedral mesh shown in green. Notice that on one side the triangles correspond to the voxel or viscous layer front and the other triangles correspond to the prolate spheroid boundary.

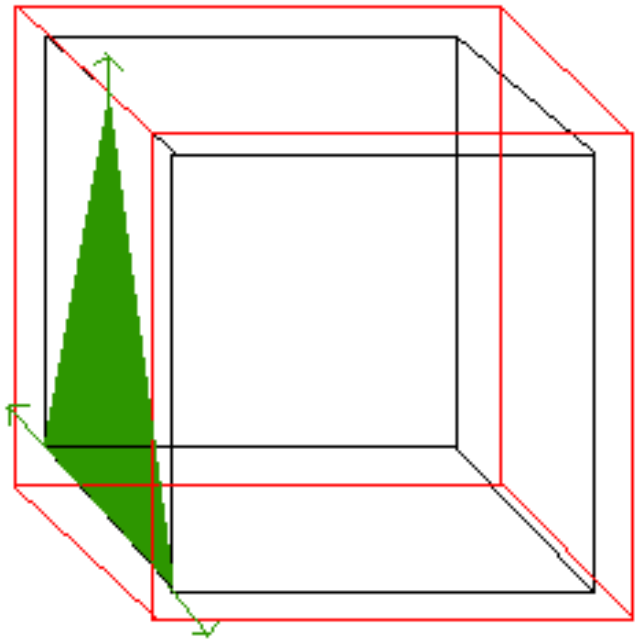


Figure 3.17: **Detecting a Coplanar Facet by Extending the Extent Box** – The green facet has its extents grown by a user-defined parameter, as does the voxel (new extents in red). This assures coplanar and near coplanar facets (as well as any facet within the extension distance) will trigger the voxel to be deleted.

3.6.2 Distinct Node Creation

At this point, distinct nodes are created by using the extents of each voxel and comparing node numbering with neighbors by moving recursively down the tree. This process requires that all quality constraints have been applied so that all faces have a match and there are no nodes outside the 27 node voxel stencil seen in Figure 3.18.

The distinct nodes are created using the values of the corner points of the voxel. Node numbering is assured to be distinct by comparing the matching the stencil from the side of one voxel with the corresponding stencil on the neighbor voxel. This is shown in (6) in Figure 3.1.

3.7 Tetrahedralization, Stitching, and Post-Processing

There is now a polyhedral mesh and an area to be filled with tetrahedra, as seen in Figure 3.19. Before this area can be filled in, one must triangulate the polyhedral faces that on the voxel front, as shown in (7) in Figure 3.1. Using both Pointwise [Pointwise, 2010], a commercially available grid generation package, and TetGen [Si, 2006], a tetrahedralization is attempted on the region between the polyhedral mesh and the geometry.

As both meshers use Delaunay tetrahedralization with a Bowyer-Watson algorithm and Steiner points, there are occasions where the original boundary cannot be recovered. If this occurs, there is no manner to stitch the mesh back together, since the connectivity is not retained by the third-party code. One work around for this is to box cut the geometry (seen in Figure 3.20), which uses the full geometry extents to do voxel deletion instead of the extents of each facet independently. Thus, many meshes that could be made with a Lawson's algorithm tetrahedral mesh generator must now be box cut to allow TetGen or Pointwise adequate room to work. As discussed in Section 1.2.3, the reason that the Lawson's algorithm is so crucial is that it will do edge swapping to recover the boundary facets in lieu of deleting all tetrahedra and trying to remesh the convex hull or insert Steiner points. These tetrahedralization techniques are used in (8) in Figure 3.1.

Once the tetrahedral mesh has been created, the mesh is imported into `FASTAR` and using an octree-sorted facet list of both the original geometry facets and the stitching boundary

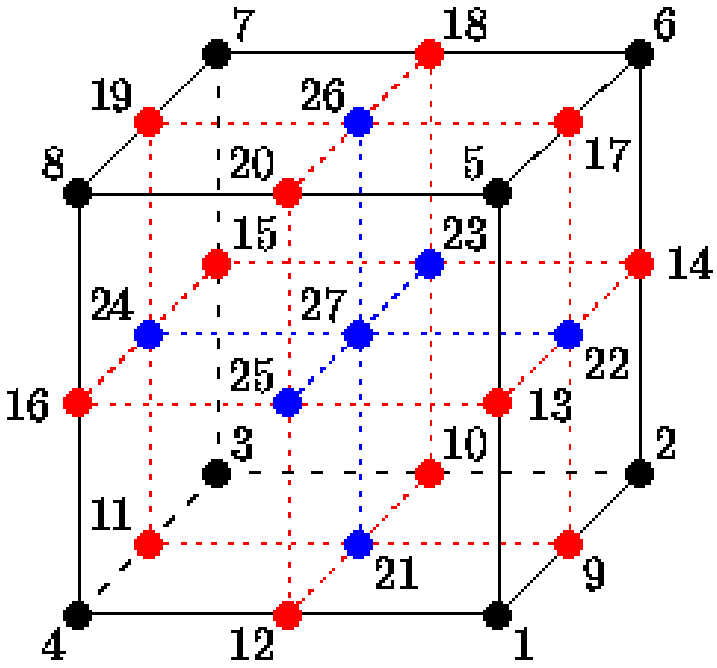
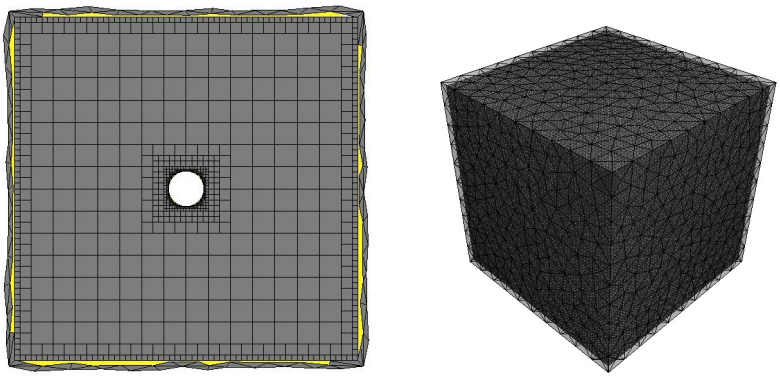
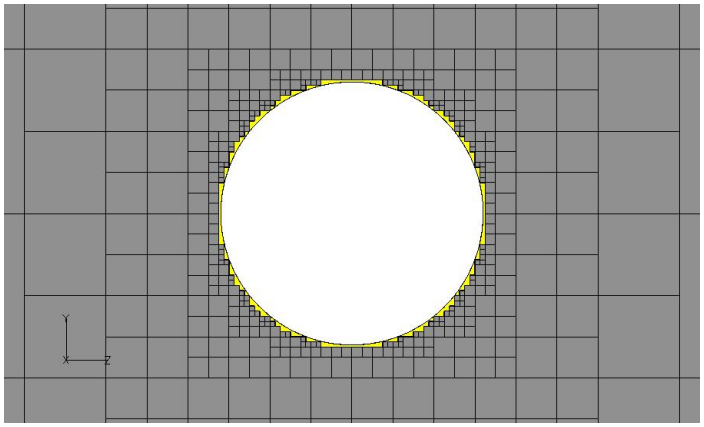


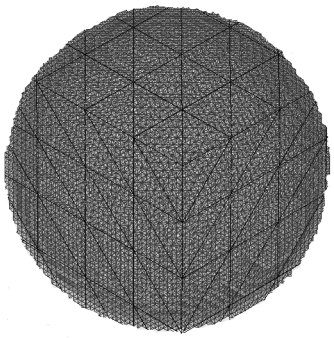
Figure 3.18: **CGNS Voxel 27 Node Stencil** – This stencil [CGNS, 2010] includes corner nodes, mid-edge nodes, mid-face nodes, and a centroid, which is unused in `FASTAR`.



(a) (b)

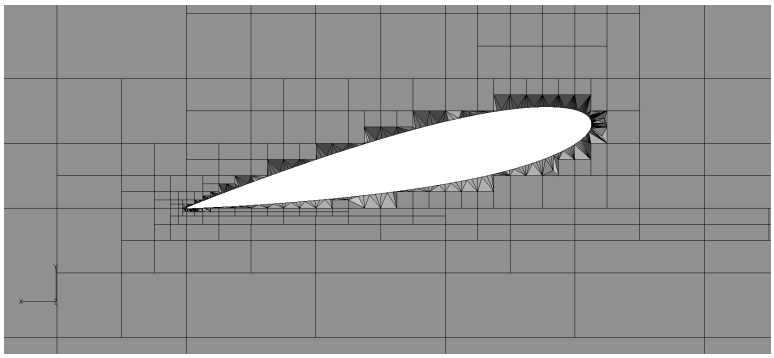


(c)

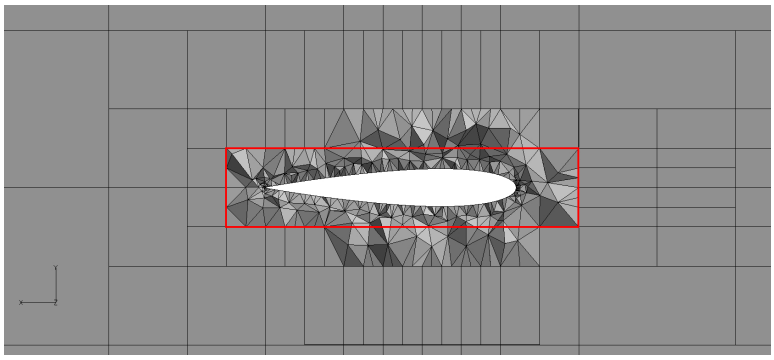


(d)

Figure 3.19: **Tetrahedralization Region** – (a) Yellow shaded region between voxel front and geometry where tetrahedralization will occur. (b) 3D picture of the outer geometric boundary and the inner stitching boundary/voxel front. (c) Yellow shaded region near to sphere where tetrahedralization will occur. (d) 3D picture of the inner spherical geometric boundary and the inner stitching boundary/voxel front.



(a)



(b)

Figure 3.20: **Box Cut Mesh on NACA 0015** – (a) This mesh has not been box cut, and the voxel front conforms to the geometry. However, this proximity can cause issues with the tetrahedral mesh generator being able to recover the boundaries, and if it can recover the boundaries, it may make very high aspect ratio tetrahedra. (b) Notice that in this mesh, the voxel front does not closely follow the geometry as it does in Figure 3.20(a). Rather, the entire extent box of the geometry (and some voxels within a user-specified spacing of the box) has been cut out of the polyhedral mesh and been filled with tetrahedra.

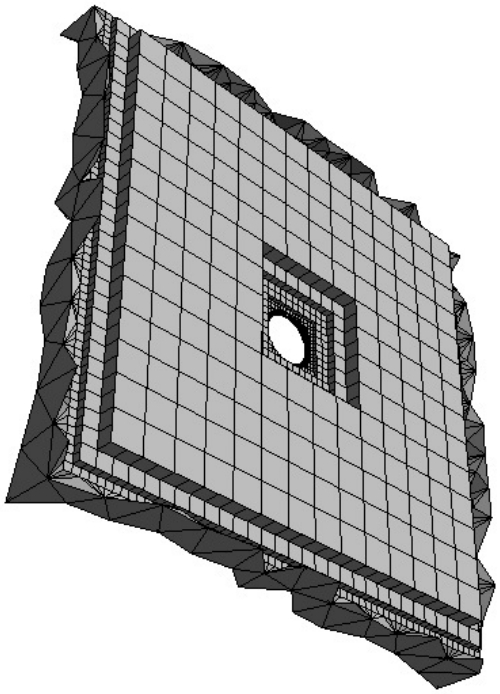
facets, the nodes are remapped to the original mesh nodes and the new nodes added by the tetrahedral mesh generator are added in, as are the tetrahedra. Finally, the stitching boundary is deleted and the final mesh is ready for the flow solver, as seen in (9) in Figure 3.1. A final polyhedral mesh on the sphere is shown in Figure 3.21.

If desired, all `FASTAR` meshes can be converted to a standard hybrid unstructured mesh comprised of a combination of tetrahedral, pyramid, prism and hexahedral elements, as seen in Figure 3.22. If a polyhedral element can be directly mapped using the element to node connectivity to one of the standard element types the conversion is made. Otherwise, averaging the nodes creates a centroid for the polyhedral element and sub-elements are constructed by connecting the centroid to the faces. Prior to this step any polygonal faces with more than 4 sides are converted to quadrilaterals and triangles. The resulting collection of sub-elements will consist of tetrahedra and pyramids. This is possible because all elements are convex, and thus the centroid is always contained within an element.

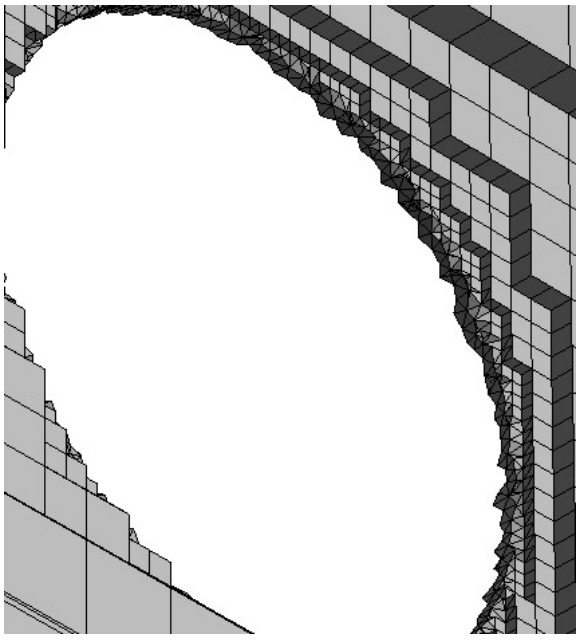
3.8 Viscous Layer Extrusion

Viscous layers may be extruded from any shape of geometry facet, although `FASTAR` uses the triangular facets to create prism element. Currently, this can be accomplished for convex geometry only using normals from each geometry node and a geometric progression factor and initial spacing supplied by the user. First, for all facets on viscous boundaries, surface normals based on the facet are created from each node, except in the case where two viscous boundaries meet or a viscous and inviscid boundary meet. In the former case, a normal is created in the average direction of the normal off each facet containing the node. In the latter case, no normal is created, and fan elements are created as seen in Figure 3.23 to assure that no crossing of boundaries occurs. The drawback to these fan elements is that in the case of very large numbers of viscous layers, the size of each element is so small that often the flow solver cannot obtain an adequate volume and will fail.

Once the normals have been created and given the magnitude of $\sum_{n=0}^{l-1} f^n s$, where l is the number of layers, f is the geometric progression factor, and s is the initial viscous spacing, nodes are created at the location of each term as the former sum is computed. Then,



(a)



(b)

Figure 3.21: **Final Polyhedral Mesh on Sphere** – (a) Polyhedral mesh on the sphere within a cube. (b) Close up of the polyhedral mesh on the inner sphere.

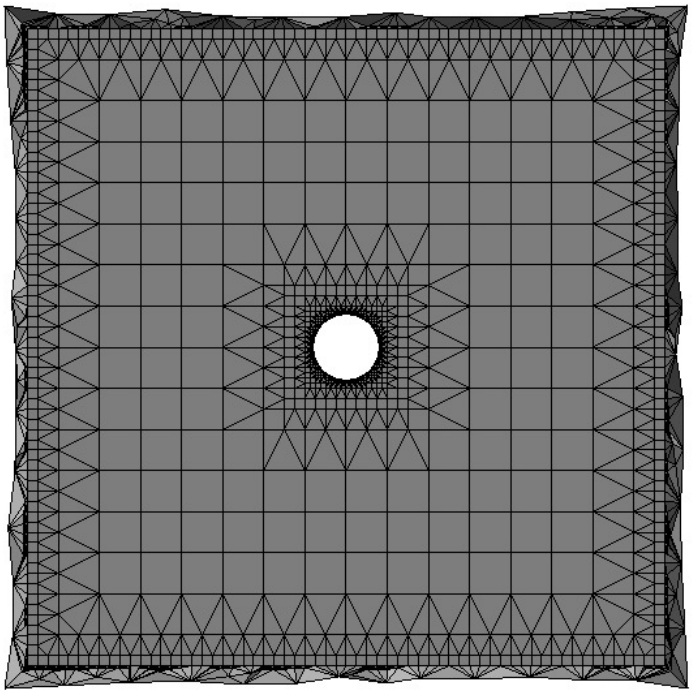


Figure 3.22: **Sphere Mesh Converted to Standard Types** – Stitched mesh on sphere geometry, that has been converted to four basic types.

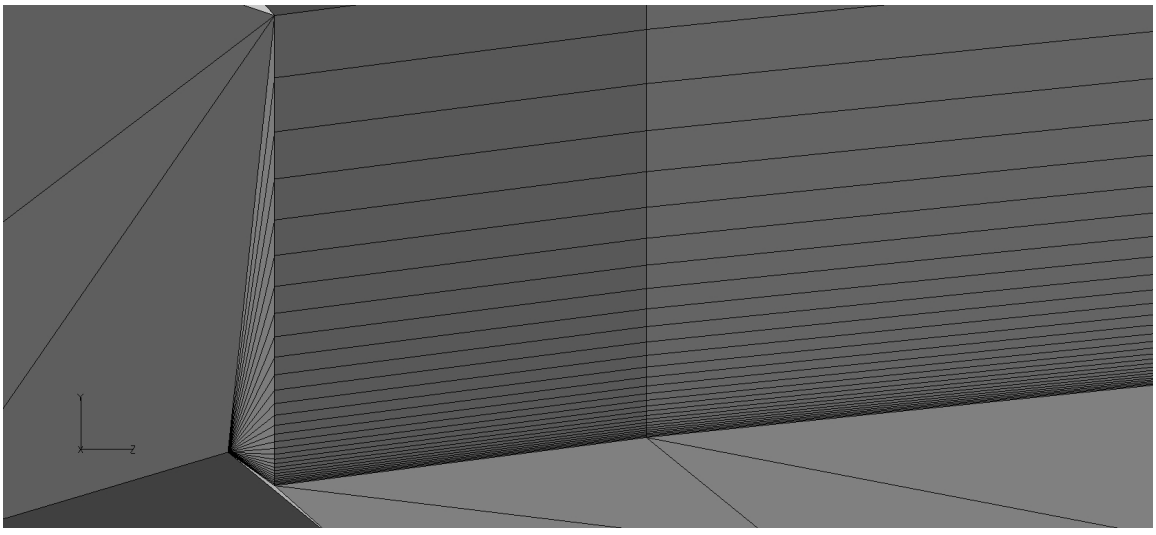


Figure 3.23: **Fan Element Creation** – Where the viscous boundary meets the inviscid symmetry plane, a fan element is created.

prismatic elements are created using the given nodes, and the top surface of each prism is added to the stitching boundary. This is the branch option seen in (4) in Figure 3.1.

Note that in Figure 3.24, the thirty-five viscous layers are extruded for a user specified initial viscous spacing of 1.0e-05 with a geometric progression factor of 1.15. Viscous layers are always composed of prismatic elements in `FASTAR`, and the reason they are so convenient to create is that they are extruded from the geometry as opposed to inserted into the mesh, so there is no mesh to push back–only an area to extrude into.

3.9 Overset

In some circumstances, an unstructured overset mesh is desired for rapid generation. This is made simple by the fact that there is no voxel deletion; rather, there is a volume mesh on the entire farfield, constructed with the geometry in mind, and a separate mesh of viscous layers on the geometry. Thus, viscous layers can be created and kept as part of an inset mesh on the geometry's viscous components, creating an overset mesh, as is posited in (5) of Figure 3.1.

The entire outer computational domain can be rendered to be easily overset by this inset mesh by continuing to use the Riemannian Metric Tensors from the geometry to drive refinement. Then, when deletion would occur, it is skipped and the viscous layer mesh is output to a separate file. This makes for a seamless interface, as can be seen in Figure 3.25. Now, the component grids are processed with an overset grid processing algorithm to classify points and to compute inter-grid interpolation coefficients which are used by the flow solver to yield flow solutions. This can all be done without the need for the time consuming deletion and tetrahedralization.

3.10 Manual Parallelization

While `FASTAR` has yet to be parallelized, the user can determine that a mesh might be too large to generate on one machine or that there need to be drastically different spacings on different parts of the mesh and not desire to use spacing boxes. In this case, the geometry

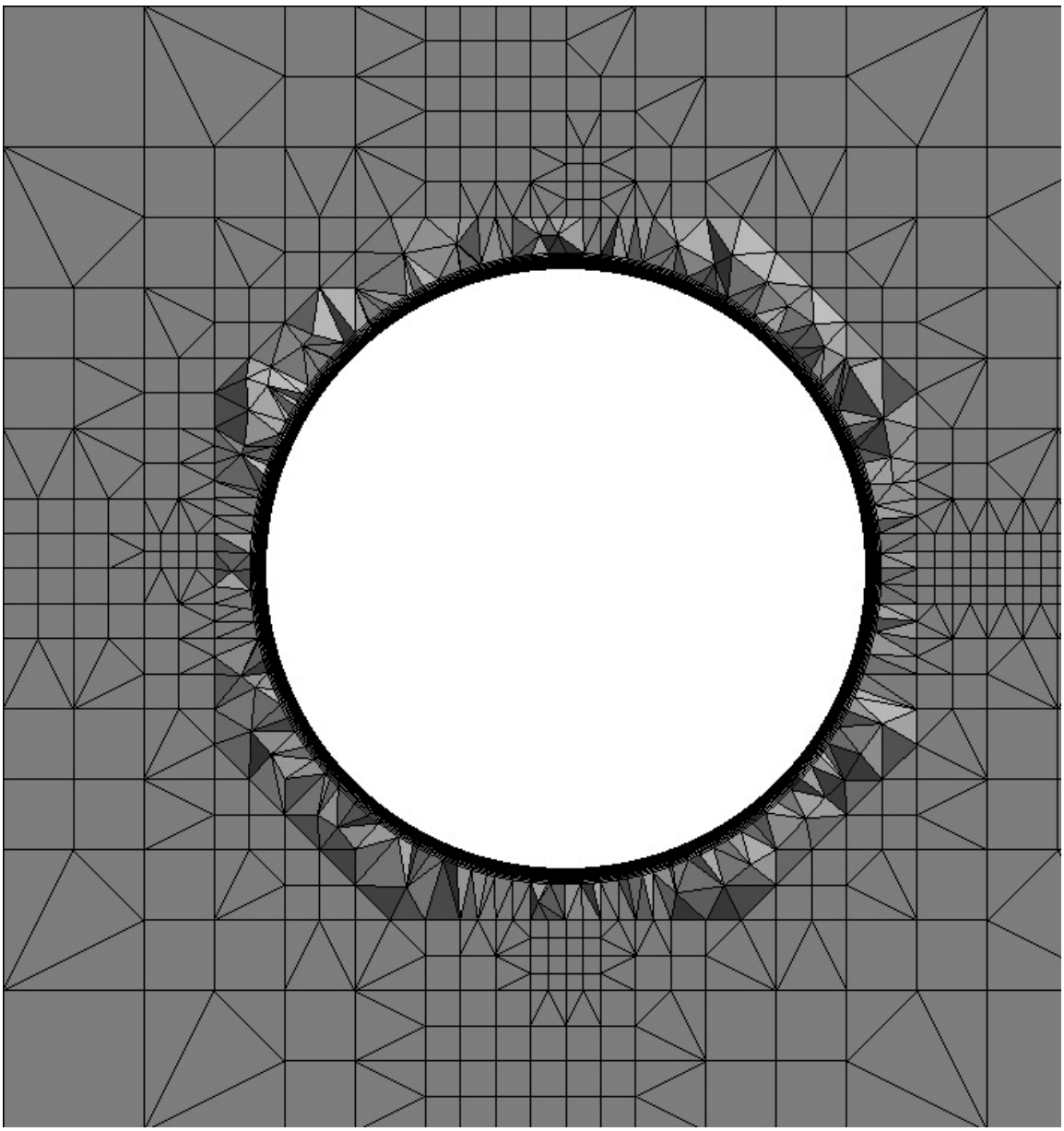


Figure 3.24: **Viscous Layer Extrusion** – This is an ellipsoid which has had viscous layers extruded. These are the prismatic elements (seen here from the side as quadrilaterals) around the surface of the ellipsoid.

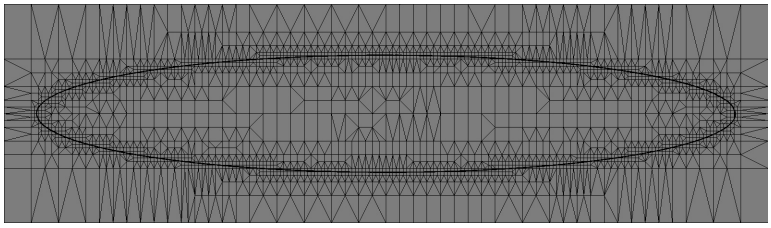


Figure 3.25: **Overset Viscous Extrusion** – A mesh composed of the ellipsoid geometry and twenty viscous layers of initial height 1.0e-15 and with geometric progression factor of 1.15 is overlaid on a mesh of the outer computational domain that was generated with consideration of the geometry through the influence of Riemannian Metric Tensors.

can be broken apart, with stitching boundaries being added to each piece where they come together.

Since the boundaries are preserved for `FASTAR` to be able to stitch a tetrahedral mesh onto the polyhedral mesh, a geometry can be arbitrarily divided, meshed at appropriate spacing for each part, and then stitched back together in the same fashion. This lends itself to an embarrassingly "parallel" algorithm, although the user must manually (and cleverly) divide the geometry to get the best quality mesh possible and be accepting of the lack of Cartesian elements at interfaces between sections. This is also analogous to multi-block meshing with a simple stitching interface derived from the natural state of the algorithm itself.

The mesh seen in Figure 3.26 was created on the DPW IV geometry in two sections. A Pointwise mesh was created on the fuselage and outer boundary, and a viscous `FASTAR` mesh was created on the wing, the area of interest. The two meshes were stitched together to create one complete volume mesh.

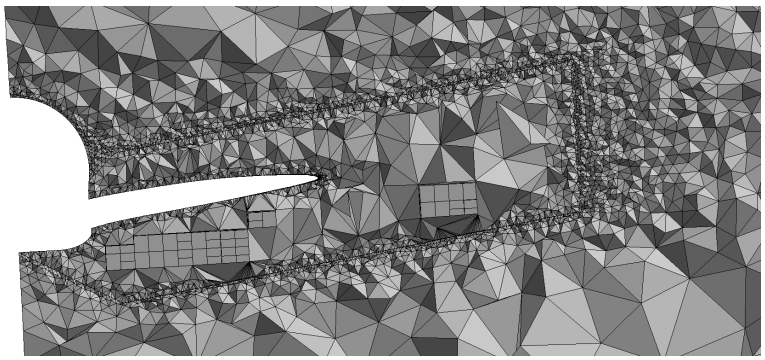


Figure 3.26: **Multiblock Mesh Can be Generated in Parallel** – This mesh on the DPW IV geometry has been created in two sections. A Pointwise mesh was created on the fuselage and outer boundary and a viscous FASTAR mesh was created on the wing, the area of interest.

Chapter 4

Computational Results with Data Comparisons

4.1 Basic Results

To better understand the functionality of `FASTAR`, some simple geometries will be introduced to display the various styles of mesh `FASTAR` can generate.

First, `FASTAR` can generate both unit aspect ratio and non-unit aspect ratio meshes. In Figures 4.1 and 4.2, a cube is shown that has been refined both with non-unit aspect ratio and unit aspect ratio parameters. The elements seen are polyhedral. This cube had a geometry spacing of 0.0625, which was observed for the unit aspect ratio case. In the non-unit aspect ratio case, the normal spacing at the geometry facets was changed to 0.00625 to make the normally symmetric tensors generate a non-unit aspect ratio mesh.

`FASTAR` utilizes spacing fields to assure proper refinement in a given region. In Figure 4.3, a cube is shown for which a spacing field was applied forcing refinement between -0.1 and 0.1 in the z direction to make elements of height 1.0e-03. The rest of the mesh was generated using Riemannian Metric Tensors with all spacings set at 0.0625.

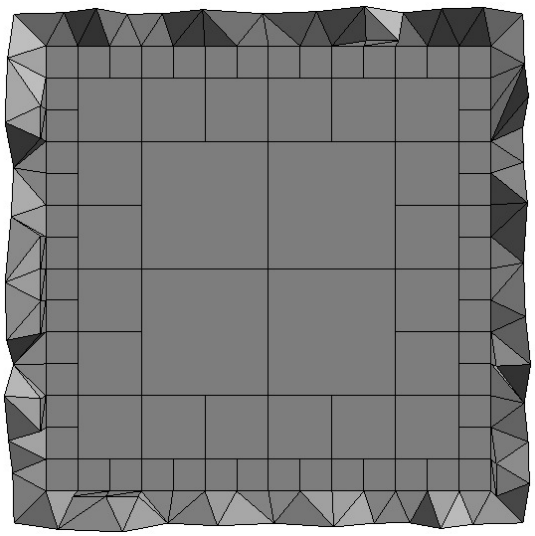


Figure 4.1: **Unit Aspect Ratio Cube Mesh** – A unit aspect ratio mesh on a simple cube with 0.0625 spacing at the boundaries.

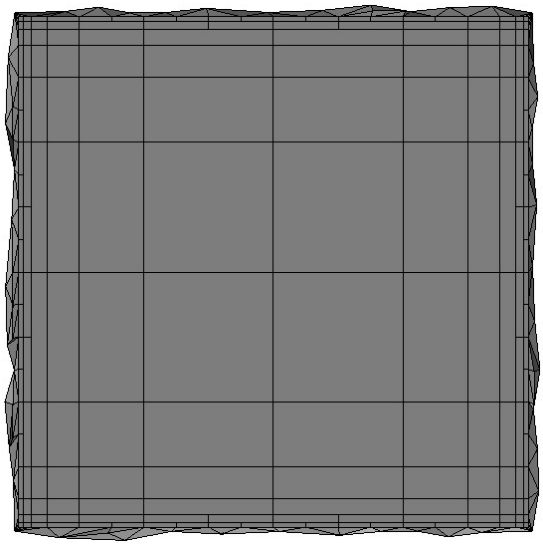


Figure 4.2: **Non-unit Aspect Ratio Cube Mesh** – A non-unit aspect ratio mesh on a simple cube with 0.0625 tangential spacing and 0.00625 normal spacing at the boundaries.

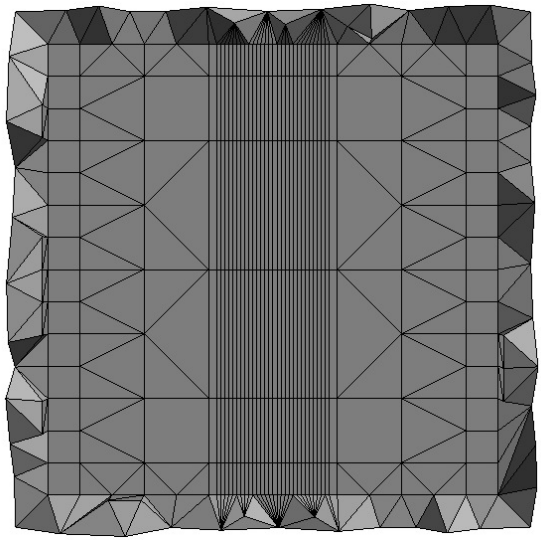


Figure 4.3: **Cube Mesh with Spacing Field Applied** – A non-unit aspect ratio mesh on a simple cube with a spacing field defined for a portion of the mesh.

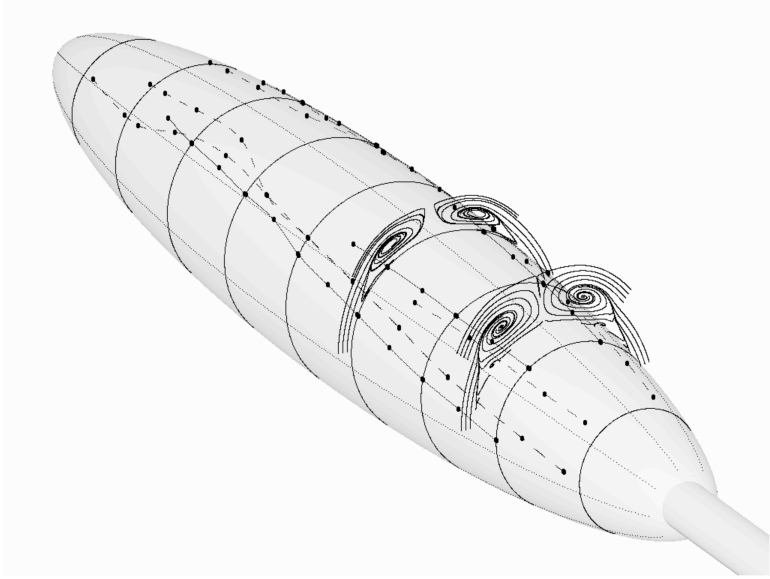


Figure 4.4: **Experimental Data Showing Vortices Along the Major Axis of the Prolate Spheroid** – Simpson's experimental solution data showing vortices along the major axis of the prolate spheroid at x/L = 0.600 and x/L = 0.772 as well as velocity streamlines.

4.2 6:1 Prolate Spheroid

The following 6:1 prolate spheroid case is used to demonstrate that the meshes generated with `FASTAR` and the solutions run on them with Tenasi, the SimCenter in-house flow solver, compare well with experimental results generated by Simpson at Virginia Tech [Simpson, 2000]. Figure 4.4 shows Simpson's results, and Figures 4.5 and 4.6 show those generated by `FASTAR` and Tenasi after three rounds of adaptation. Additionally, meshes were generated with commercial and in-house mesh generation packages (Pointwise and `P_HUGG`, respectively), and the results from each type of mesh coupled with Tenasi and using adaptation are examined.

The case of a prolate spheroid, which can approximate the body of a submarine and will subsequently be referred to as an ellipsoid, has a major axis of magnitude 1.0 in the x direction and minor axes of $0.1\bar{6}$ in the y and z directions. This case is not only reasonably rapid to generate, but it also allows for the use of adaptation to resolve vortices along the major axis.

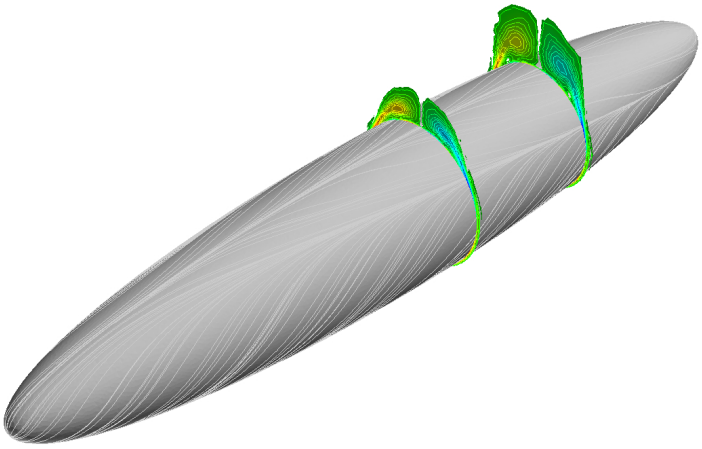


Figure 4.5: **Tenasi Solution Showing Vortices Along the Major Axis of the Prolate Spheroid** – This solution was generated with `FASTAR` and Tenasi after three rounds of adaptation, and it compares well with that of Simpson. One can see the secondary vortices beginning to develop.

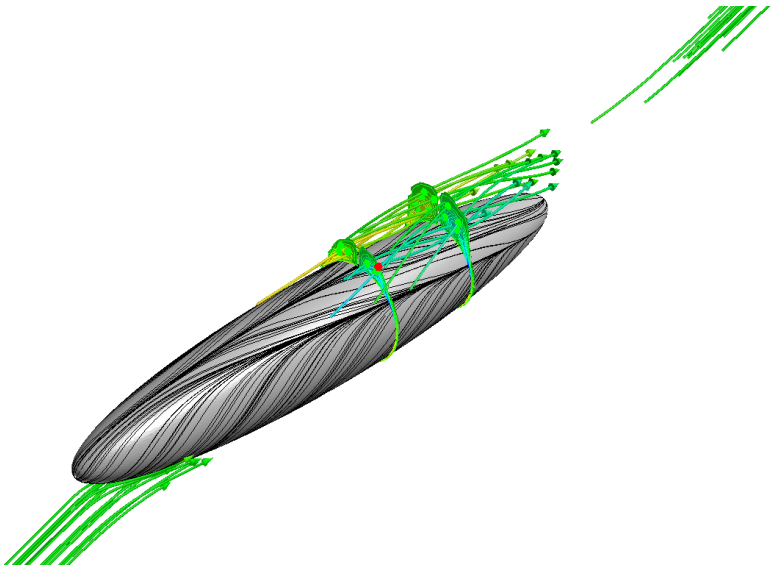


Figure 4.6: **Tenasi Solution Showing Streamlines** – The stream lines show the flow being explored, and separation can be seen on the surface of the ellipsoid.

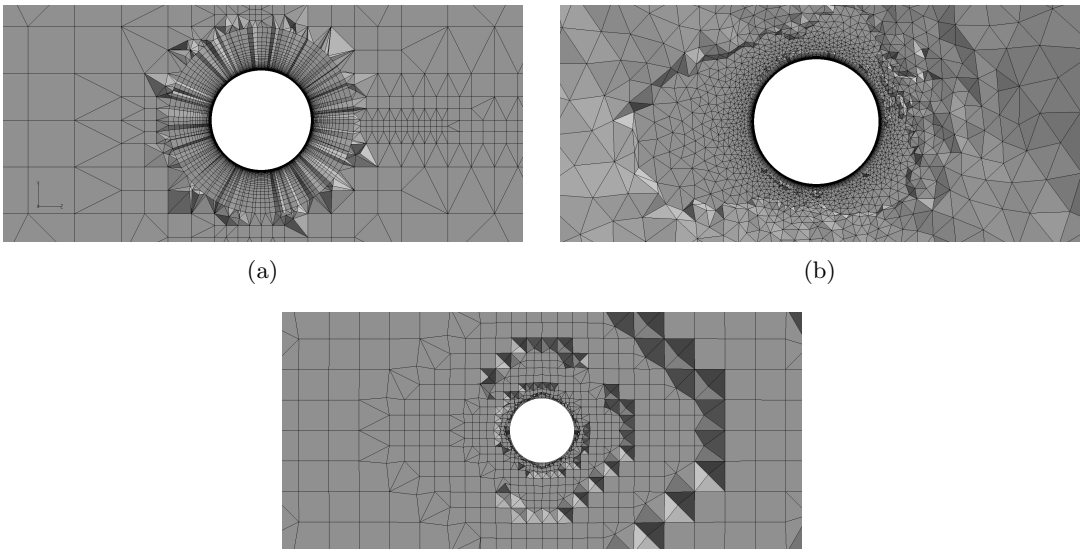
The initial meshes for this case were generated with a minimum spacing of 0.01 and a maximum spacing of 5.0, and the final meshes were generated with a minimum spacing of 0.001 and a maximum spacing of 5.0. Also, viscous layers were inserted on all cases. The initial `FASTAR` mesh has 50 viscous layers with an initial height of 1.0e-05 and a geometric progression factor of 1.15. In order to allow the volume mesh to resolve vortices and utilize adaptation instead of only viscous packing, these numbers were reduced by 10 viscous layers at each adaptation. Additionally, the spacing generated by the tetrahedralization had to be observed in assuring the transition between viscous layers, tetrahedral region, and polyhedral mesh was sufficiently smooth. The `P_HUGG` and Pointwise meshes used between 10 and 30 viscous layers in order to best match the viscous spacing with that of the first layer of volume mesh.

In Figure 4.7, a unit aspect ratio initial mesh was generated by `FASTAR`. Also, a Pointwise generated mesh on the same ellipse is shown, with viscous layers inserted by `P_VLI`, and a `P_HUGG` mesh was generated with the same spacing parameters, optimized with `P_OPT` [Karman et al., 2008] to remove sliver cells, and then viscous layers were added with `P_VLI` [Karman, 2007].

In Figure 4.8, the initial mesh solutions are presented with helicity being the flow field function of interest. Helicity is computed using (4.1), where $\nabla\times$ is the curl of the vector and $\vec{v}$ is the velocity vector with components u, v, and w. These solutions were obtained by running Tenasi to 5,000 iterations, with an incompressible flow regime, a 20° angle of attack, a mach number of 0.2, a Reynolds number of 4.2e6, and the Menter SST $k-\omega$ two-equation eddy viscosity turbulence model. All plots are oriented in the positive x direction, all cases were decomposed and run in parallel, and the cut seen here is $\frac{x}{L} = 0.600$.

$$H = (\nabla \times \vec{v}) \cdot \vec{v} \tag{4.1}$$

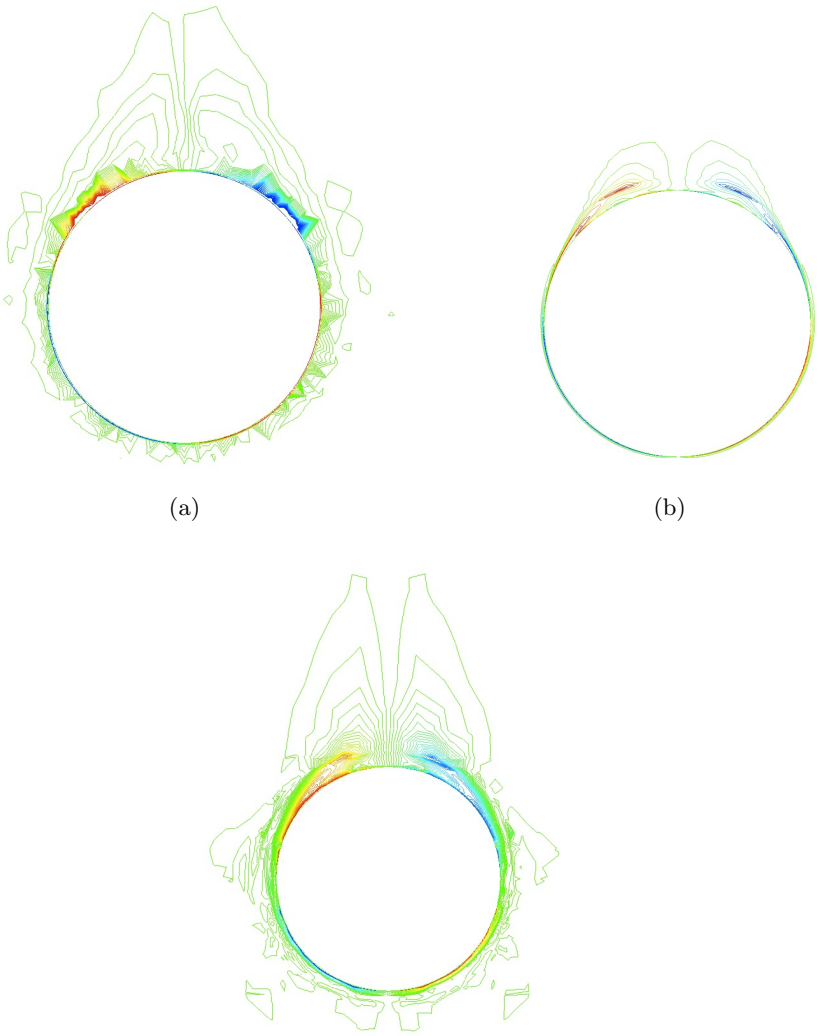
Additionally, in Figure 4.9, contour plots are shown of the x, y, and z spacing field parameters generated by Karman's Spacing Field code using helicity as the adaptation function on the unit aspect ratio `FASTAR` mesh. Also, when running the Spacing Field code, prism nodes were considered for adaptation on only the first cycle, since the flow occurs



(a) (b)

(c)

Figure 4.7: **Initial Meshes on 6:1 Ellipsoid** – (a) `FASTAR`: A unit aspect ratio mesh was generated on a 6:1 ellipsoid and 50 viscous layers were added to make the spacing transition less rapid from viscous layers to volume mesh. (b) A tetrahedral Pointwise mesh was generated on a 6:1 ellipsoid with boundary decay of 0.9. Thirty viscous layers were inserted by `P_VLI`. (c) A `P_HUGG` mesh was generated on a 6:1 ellipsoid, then smoothed by `P_OPT`. Finally, 25 viscous layers were inserted with `P_VLI`.



(a) (b)

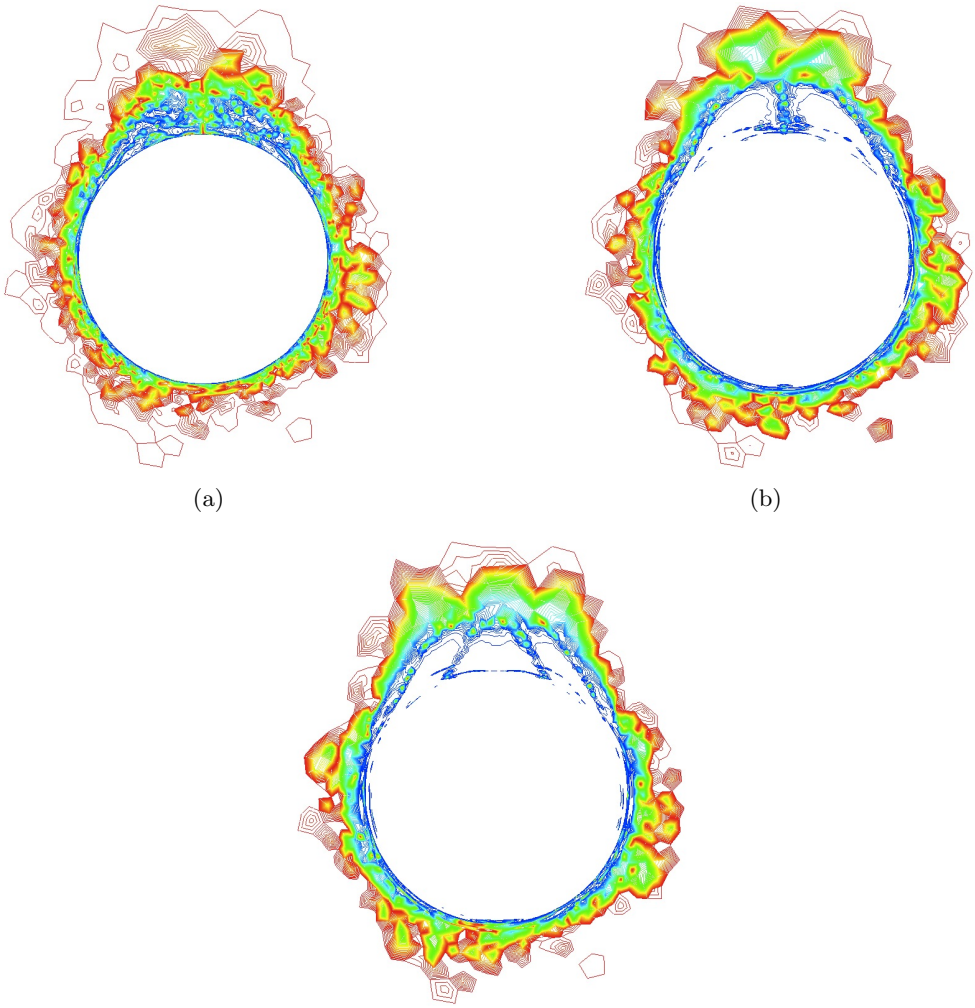
(c)

Figure 4.8: **Initial Scalar Solution on Ellipsoid with Helicity** – (a) `FASTAR`: While the vortices were captured on this first run, the resolution is not perfect and the secondary vortices are yet to be resolved. (b) The Pointwise mesh gave the best initial results; however, it also had the most elements. (c) `P_HUGG` generated the smallest mesh due to the gradation parameter that can be supplied in the initial generation phase.

mostly in the viscous region, the length scale exponent was unity, elemental gradients were weighted by control volume, and the error parameter used to generated the new metric lengths was one standard deviation from the mean. While FASTAR and P_HUGG are able to generate meshes using the adaptation tensors directly, the Pointwise meshes were refined with subdivision refinement found in Karman's P_REFINE [Karman et al., 2009]. The drawback of using P_REFINE is that refinement must be done to the original mesh and it must be subsequently smoothed; alternately, in P_HUGG and FASTAR, the mesh is regenerated using the proper spacing parameters from the outset, allowing for a smooth transition. Alternately, in P_REFINE and P_HUGG, the viscous elements are refined either directly or by the fact that the surface mesh has been changed; in FASTAR, the surface mesh does not change, so the only control that can be exerted on the viscous layers is stack height.

This process continued over three adaptation cycles, yielding the meshes in Figure 4.10, and the final solutions are found in Figure 4.11. The choice of three adaptation cycles was arbitrary; in a dynamic environment, adaptation would occur until an error threshold was reached or until the solution stopped changing within a tolerance. The solutions compare well to Simpson's results, seen in Figures 4.4, 4.12, 4.13, and 4.14, thus showing the robust nature of meshes generated by all three codes. While the adaptation cycles were only run to 5,000 iterations and a residual value of 1.0e-08 to conserve time and still get the adaptation desired, the final solutions were run for 7,500 cycles. Final residuals were shown for all three meshes in Figure 4.15. The mesh was run through 1,000 first order iterations to smooth out the large initial gradients, and it was then switched to second order, thus the spike in residual at 1,000 iterations. Also, it is believed that the reason the P_HUGG residual was so much lower than the FASTAR and Pointwise residuals was due to the optimization of the P_HUGG mesh with P_OPT, yielding fewer poor quality elements.

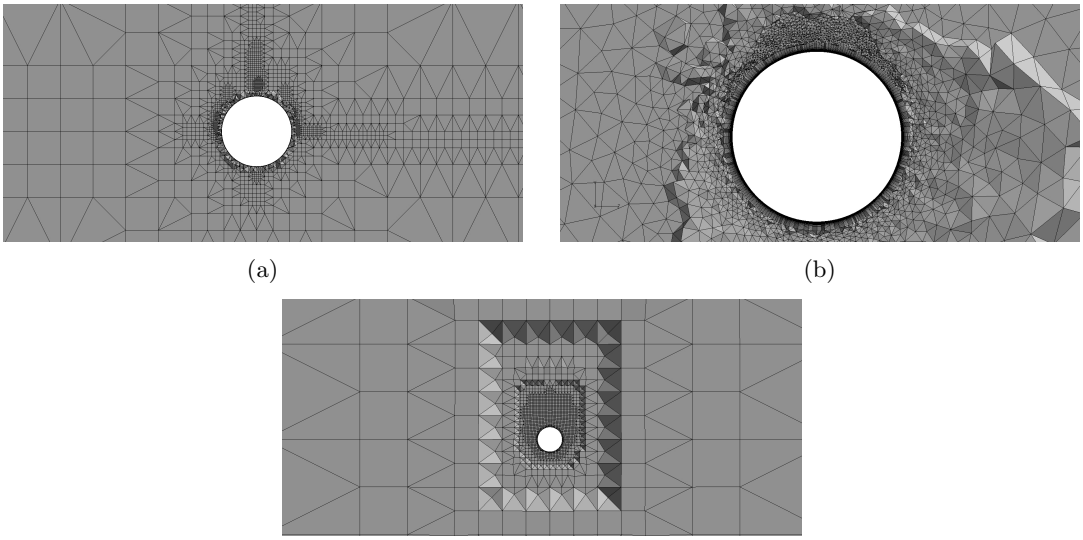
The relative error between the experimental results and the solutions obtained after three cycles for each type of mesh are shown in Table 4.1. These were obtained by using (4.2) at twelve equally spaced points along the velocity profile, and taking the average of each of these calculations. It should be noted that the geometry that is optimal for Pointwise and FASTAR is not the geometry that is ideal for P_HUGG. For both Pointwise and FASTAR, an unstructured Delaunay tetrahedral surface mesh is utilized to assist the



(a) (b)

(c)

Figure 4.9: **Contour Plot of Spacing Field Based on Helicity** – These contour plots show regions of refinement that will be promulgated through the next mesh generated in each cardinal direction, respectively. The contours are based on the new metric lengths for each element in the mesh.



(a) (b)

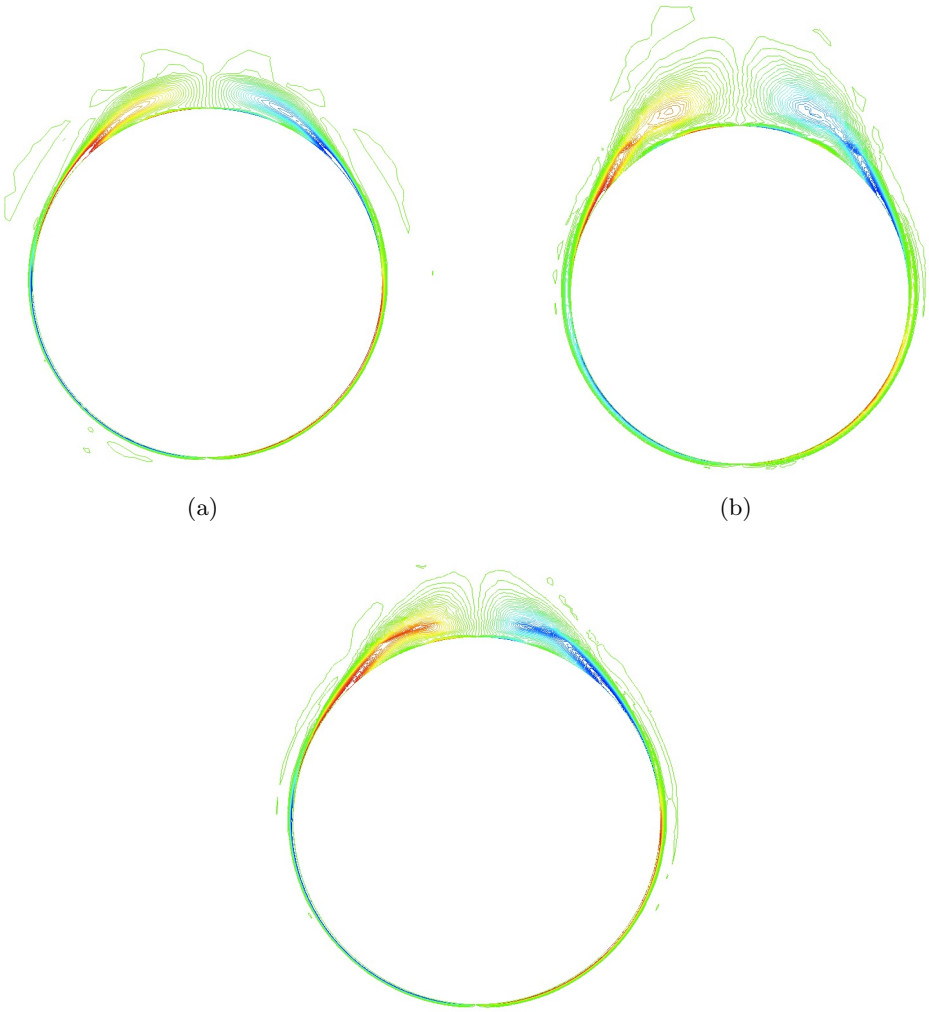
(c)

Figure 4.10: **Adapted Meshes on 6:1 Ellipsoid** – The refinement seen above the ellipsoid is based on the helicity adaptation, and the results are for `FASTAR`, Pointwise, and `P_HUGG`, respectively.

Delaunay tetrahedral mesh generator in recovering boundaries. However, due to the nature of general cutting with a hierarchical cartesian mesh, as is done in `P_HUGG`, a diagonalized unstructured surface mesh would have been better suited, due to the Cartesian alignment of facets with voxels. The lack of Cartesian aligned boundary facets caused thousands of negative volume elements (sliver cells or other cut elements) to be generated, and `P_OPT` could not untangle the mesh in a reasonable amount of time. Additionally, this problem is again exacerbated in `P_VLI`, where these elements must have prisms inserted between them and the Cartesian mesh, causing even further difficulty with smoothing the rest of the non-aligned mesh between layer insertions.

$$err = \frac{|exp - comp|}{exp} \tag{4.2}$$

Since all the mesh generators were able to resolve the flow features well, it is instructive to look at element counts and times to generate the mesh for each code. The times to generate each type of mesh (does not include solution times, which were all nearly equal,



(a) (b)

(c)

Figure 4.11: **Final Scalar Solution on Ellipsoid with Helicity** – Contour plots of helicity are displayed on `FASTAR`, Pointwise, and `P_HUGG` adapted meshes. These are viewed looking in the negative x direction.

Mesh Generator	Relative Error (percent)
FASTAR	2.02
P_HUGG	3.18
Pointwise	1.14

Table 4.1: **Relative Error Between Experimental Results and Three Types of Computational Results** – This is the relative error from twelve sample points between the experimental results of Simpson and `P_HUGG`, Pointwise, and `FASTAR` adapted meshes. These numbers come from the cut $\frac{x}{L} = 0.600$.

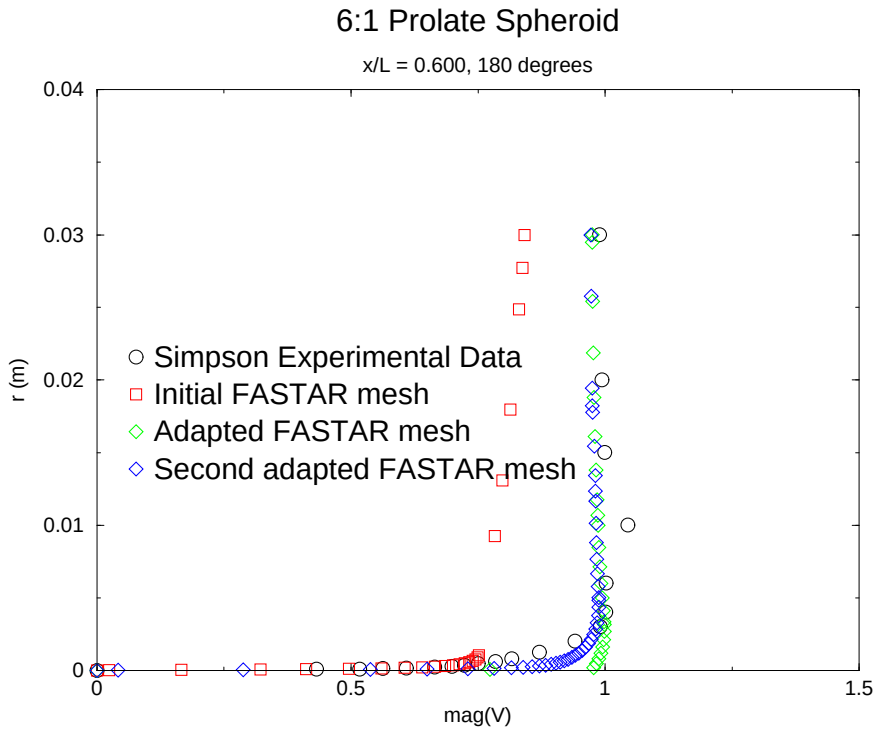


Figure 4.12: **Comparison of Computed Solution and Experimental Data at x/L = 0.600** – Simpson's experimental data is compared with the solution obtained by `FASTAR` and Tenasi at three adaptation steps. Note that as adaptation progresses, the matching of the computed and experimental solutions improves.

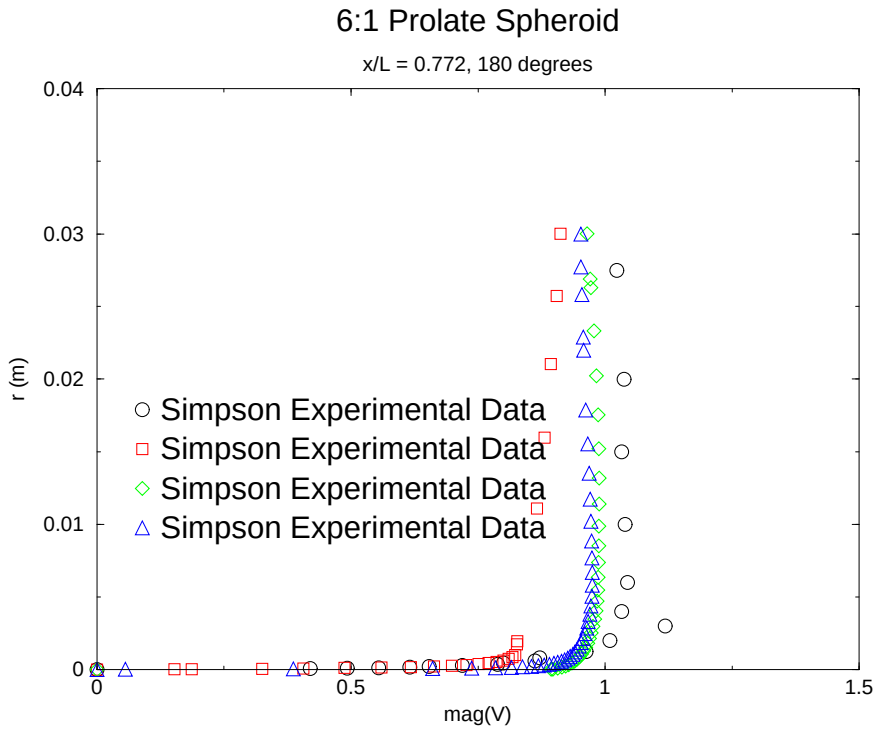


Figure 4.13: **Comparison of Computed Solution and Experimental Data at x/L = 0.772** – Simpson's experimental data is compared with the solution obtained by `FASTAR` and Tenasi at three adaptation steps.

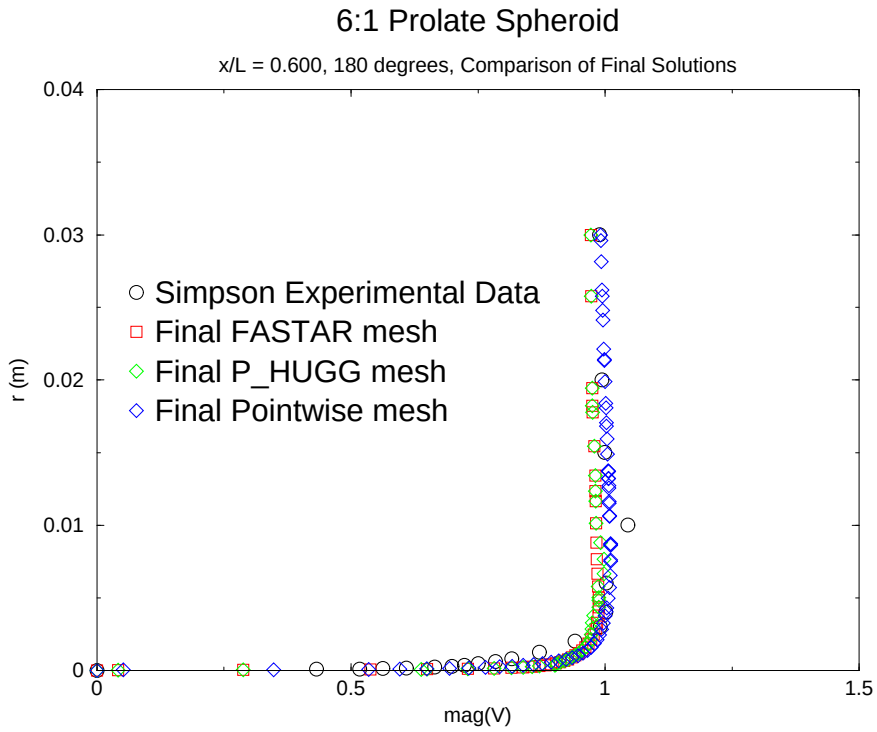


Figure 4.14: **Comparison of Computed Solution and Experimental Data at x/L = 0.600 for Three Types of Meshes** – Simpson's experimental data is compared with the solution obtained by `FASTAR`, Pointwise, and `P_HUGG` with Tenasi at the third adaptation step.

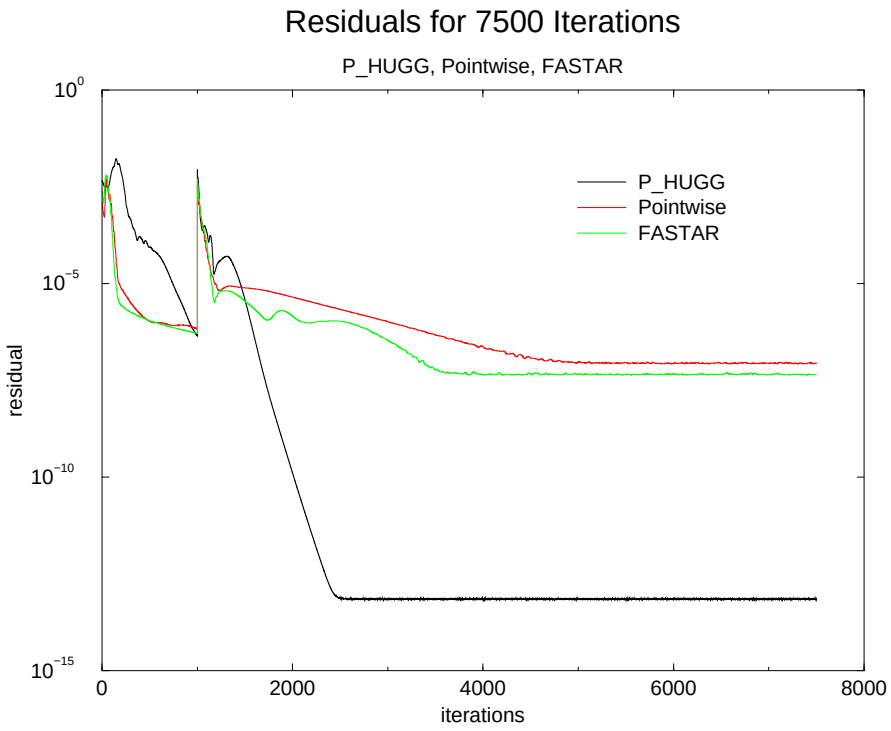


Figure 4.15: **Residuals from Tenasi Runs** – The residual plots show the convergence behavior of the solution runs on each type of mesh.

although decomposed to a different number of processors) are given in tables 4.2, 4.3, and 4.4. The times displayed are for a single FASTAR run, a Pointwise run coupled with P_VLI for viscous layer insertion, and for P_HUGG coupled with P_OPT to remove sliver cells and P_VLI for viscous layer insertion. The number of elements is listed based on all polyhedra being converted to the four basic types. Pointwise only makes tetrahedral meshes in this case, and the boundary decay factor used to generate the given mesh was 0.9. While it might seem odd that P_HUGG has such small element counts, this is due to the differing treatment of gradation.

Mesh	**FASTAR**		
–	Number of Nodes	Number of Elements	Time (min)
Unit aspect ratio	572,031	1,511,369	17
Helicity Adapted unit aspect ratio	618,742	1,730,074	61
Helicity Twice Adapted unit aspect ratio	950,463	2,241,587	83

Table 4.2: **Times to Generate FASTAR Meshes of Various Sizes with Various Inputs**

Mesh	**Pointwise/P_VLI**		
–	Number of Nodes	Number of Elements	Time (min)
Initial mesh	809,885	2,032,978	95
Helicity Adapted	2,384,622	5,644,139	432
Helicity Twice Adapted	3,809,390	8,512,123	814

Table 4.3: **Times to Generate Pointwise / P_VLI Meshes of Various Sizes with Various Inputs**

4.3 Seafighter

The Seafighter is a perfect example of a mesh that benefits from the ability to utilize spacing fields in FASTAR. Since this case involves an air-water interface, the cells in that region must be refined to 1.0e-03 in the y direction in order to capture the flow physics. These meshes were run with a minimum spacing of 1.0e-03 and a maximum spacing of 100.0.

Mesh	P_HUGG/P_OPT/P_VLI		
–	Number of Nodes	Number of Elements	Time (min)
Unit aspect ratio	181,438	485,265	110
Helicity Adapted unit aspect ratio	256,632	624,303	293
Helicity Twice Adapted unit aspect ratio	351,307	629,828	313

Table 4.4: **Times to Generate `P_HUGG` / `P_OPT` / `P_VLI` Meshes of Various Sizes with Various Inputs**

In Figures 4.16, 4.17, and 4.18, the Seafighter geometry is shown from both starboard and port sides, as well as a close up of the aft nozzles. In Figure 4.19, the internal nozzle structure is shown.

The main issue experienced in generating a truly form-fitting Seafighter mesh is the lack of a good tetrahedralization algorithm. Due to the complexity of the geometry and the internal flow regions, the tetrahedral mesh generator was unable to recover the boundaries using a Bowyer-Watson scheme. As such, the only way a mesh was able to be generated was through the use of box cutting, since this expands the void between the voxel front and the geometry facets. Since this mesh has internal geometry (shown in Figure 4.19), the entirety of this region was meshed with tetrahedra.

In Figure 4.20, the final volume mesh is shown with highlights on the meshing of the hull and nozzles. Note that the results use a box cutting approach that deletes all voxels within the extent box of the geometry. This can make for easier tetrahedralization but makes for a less polyhedral mesh.

In Figures 4.21(a) and 4.21(b), one can see close ups of a Seafighter mesh that has a small area of deleted voxels to be tetrahedralized and polyhedra throughout the mesh; this mesh was not box cut in order to show how the voxel boundaries would conform to the geometry in an ideal mesh. While the current tetrahedralization algorithm could not fill this small void, the quality of the mesh is good, and it could be used for an overset grid or with the Lawson's algorithm tetrahedralization routine in the future.

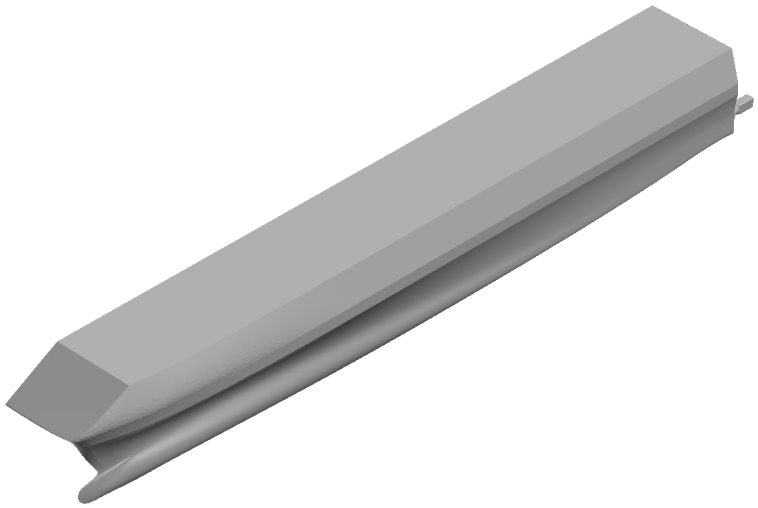


Figure 4.16: **Seafighter Geometry–Port View** – The geometry of the seafighter case viewed from the port side.

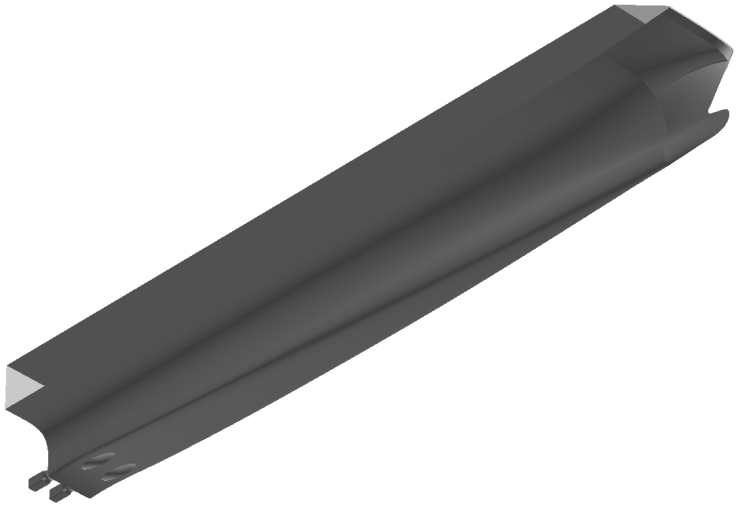


Figure 4.17: **Seafighter Geometry–Starboard View** – The geometry of the seafighter case viewed from the starboard side.

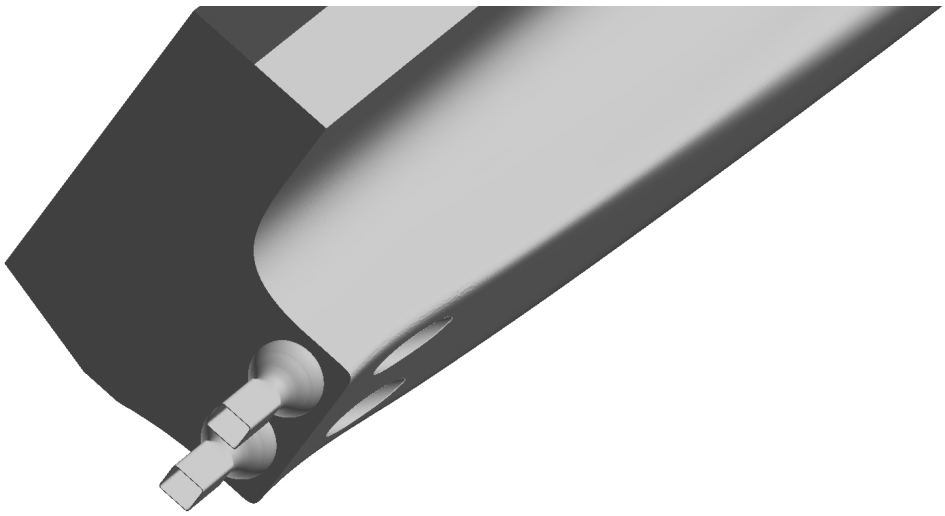


Figure 4.18: **Seafighter Geometry–Aft View** – The geometry of the seafighter case viewed from the rear, showing the nozzles.

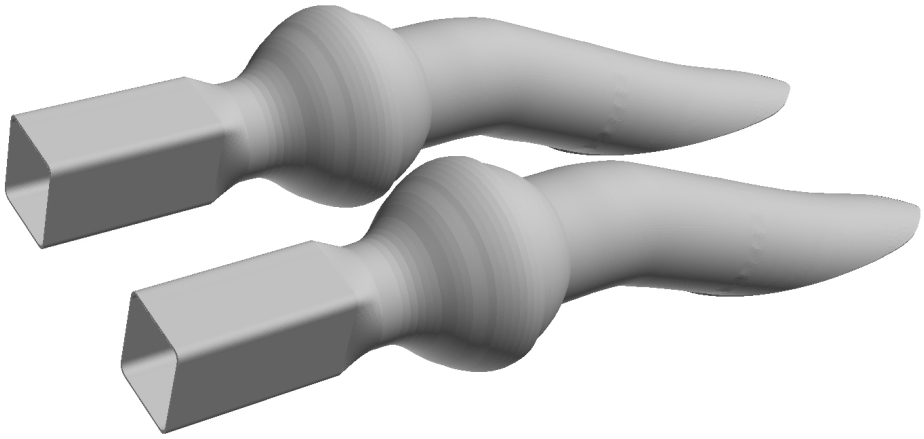
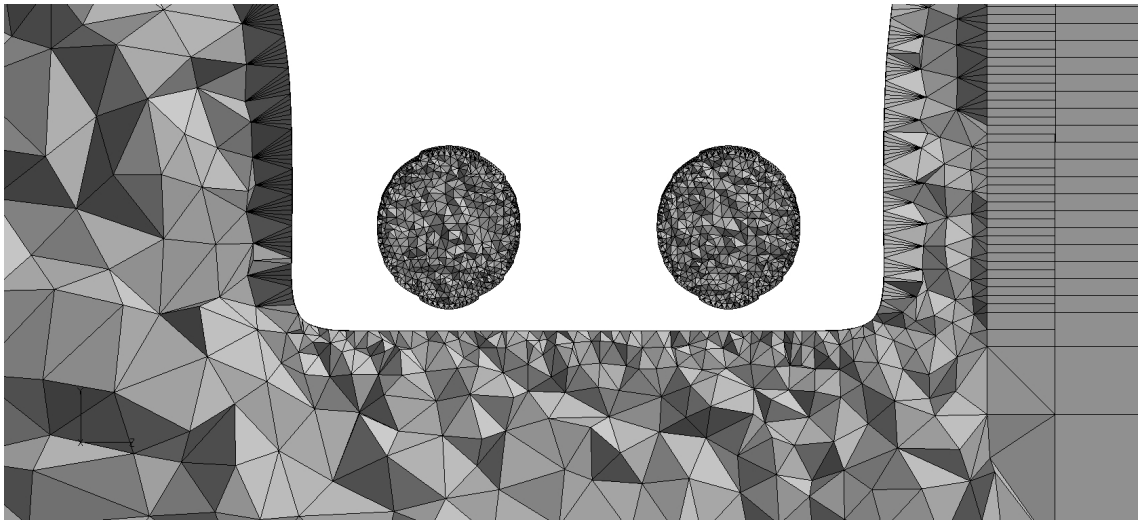
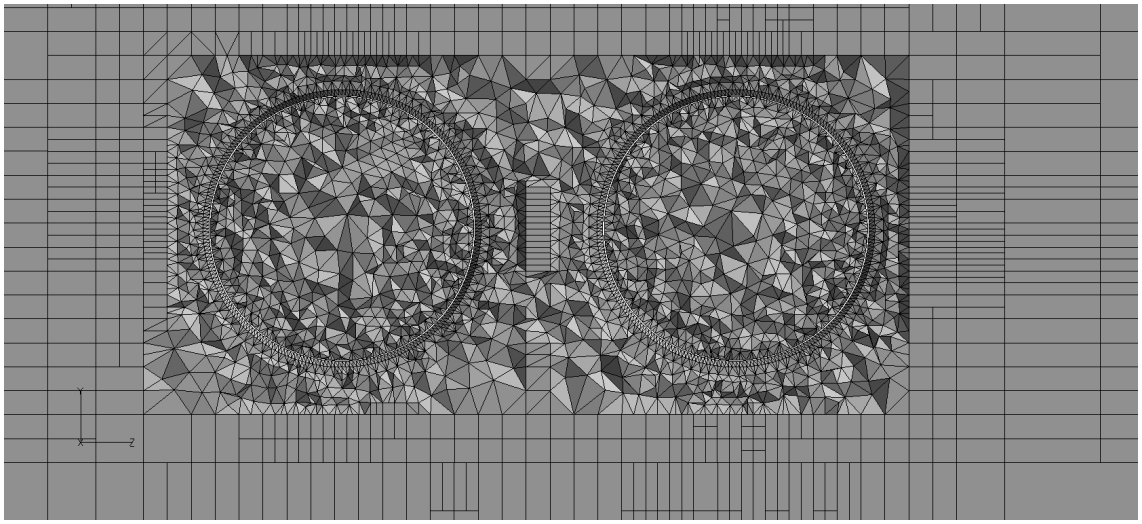


Figure 4.19: **Seafighter Geometry–Internal Nozzle View** – The internal intake and outflow nozzle geometry of the seafighter case.

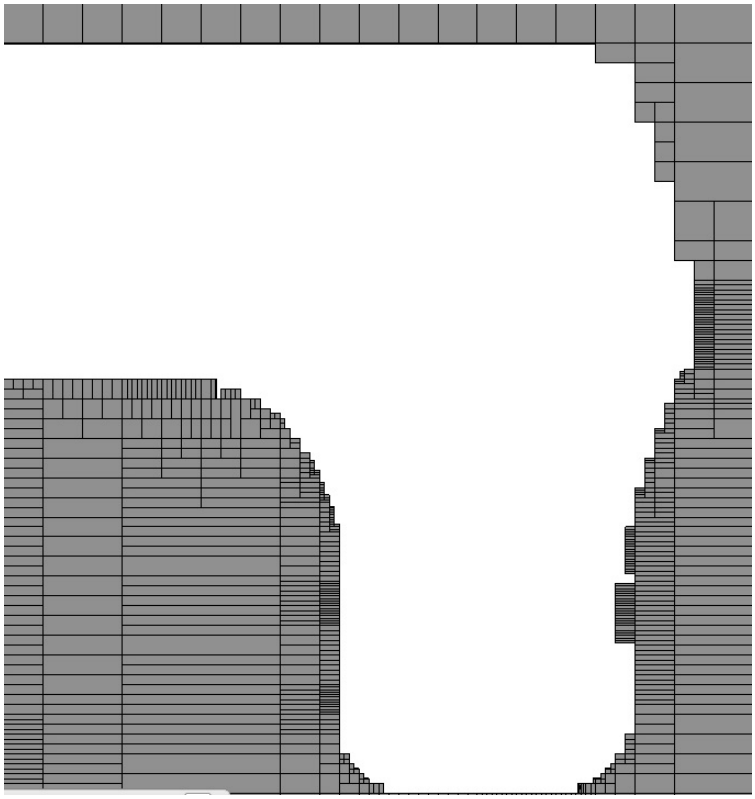


(a)

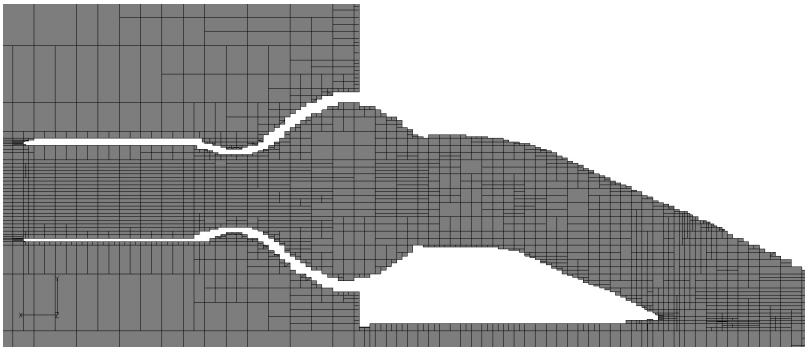


(b)

Figure 4.20: **Seafighter Volume Mesh from Various Viewpoints–Box Cut** – (a) Close up of the intakes and box cut mesh on the hull. (b) Close up of the nozzles, which have been fully tetrahedralized due to box cutting.



(a)



(b)

Figure 4.21: **Seafighter Volume Mesh from Various Viewpoints–Polyhedral** – (a) Close up of the hull. (b) Close up of the nozzles from the z-plane, including intake and outflow.

4.4 Other meshes on practical geometries

In Figures 4.22 and 4.23, a non-unit aspect ratio inviscid and unit aspect ratio viscous mesh are shown for a NACA 0015 airfoil. This mesh is used for demonstration only, since while there are a reasonable number of layers, the fan elements with 35 layers can cause the solver to fail. This issue with generating viscous meshes on symmetry plane geometries can be resolved by a more robust extrusion/insertion hybrid algorithm. The initial spacing on the layers was 1.0e-06 and the geometric progression factor was 1.15.

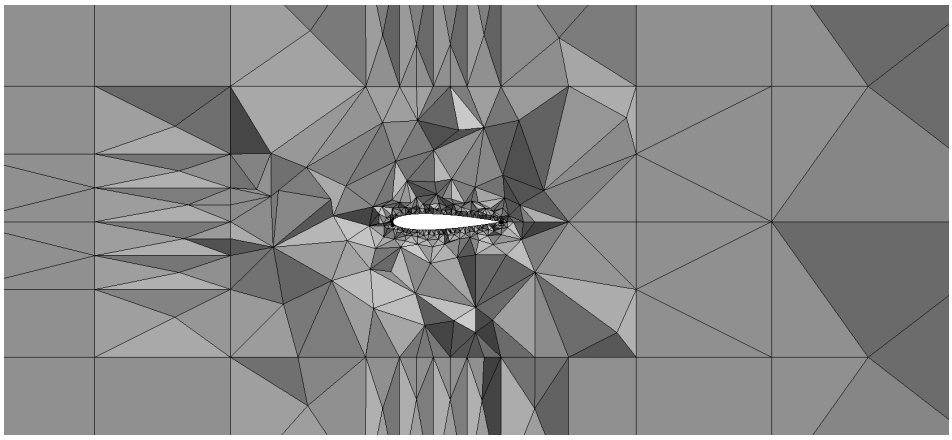


Figure 4.22: **NACA 0015–Close Up with Box Cutting** – A non-unit aspect ratio mesh on a NACA 0015 airfoil, where box cutting was employed.

In Figure 4.24, a multiblock overset mesh is generated on the Drag Prediction Workshop IV cargo plane geometry by stitching a Pointwise generated tetrahedral mesh on the fuselage and outer boundary to an overset `FASTAR` mesh on the wing. This mesh was generated in order to show that `FASTAR` can generate meshes on individual pieces of a geometry and stitch them back together, as was seen in Figure 3.26 in Section 3.10. The image is notable since along the boundaries where stitching occurs, there are layers of tetrahedra from the voxel deletion construct, and there are polyhedra created around the wing, where the flow of interest would occur.

The final real-world meshing case is that of the NASA SDT2-R4 rotor geometry. This geometry is particularly difficult due to the spacing changes between the blades and the camber of the blades. However, `FASTAR` was able to generate a form fitting mesh seen in Figure 4.25, with a minimum spacing of 0.2 and a maximum spacing of 4.0.

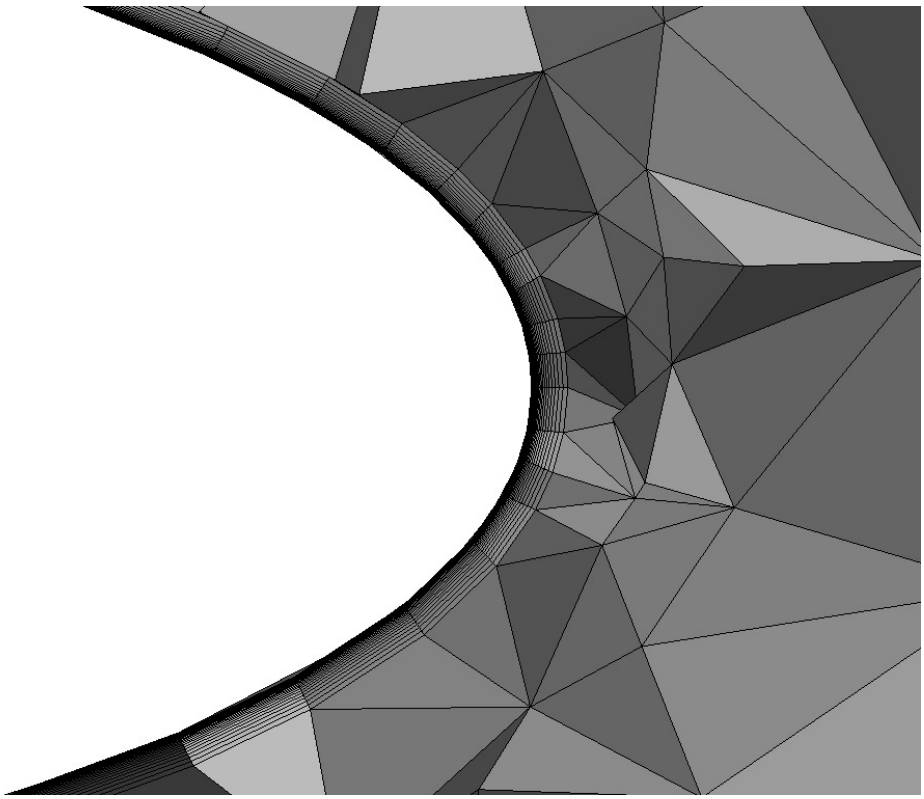


Figure 4.23: **NACA 0015–Close Up with Viscous Layers** – A close up of the viscous layers at the leading edge of a unit aspect ratio mesh on a NACA 0015 airfoil.

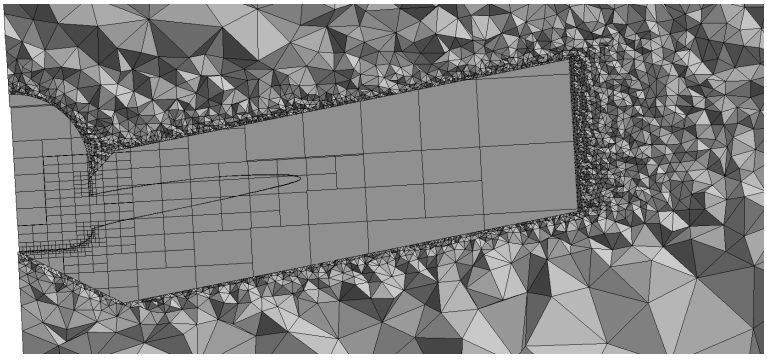


Figure 4.24: **Multiblock Overset DPW IV Mesh: Stitched Together** – This geometry was broken apart, meshed separately in Pointwise and FASTAR, and stitched back together.

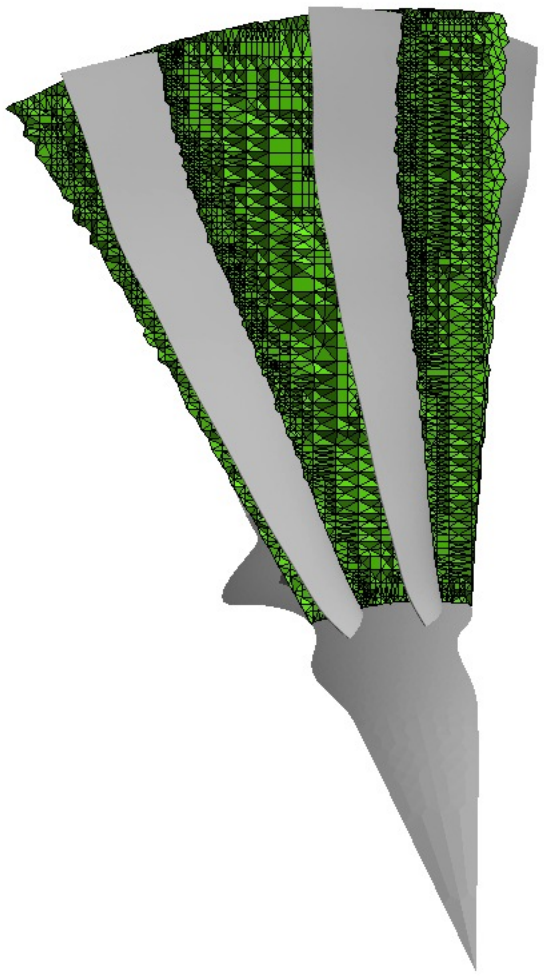


Figure 4.25: **NASA SDT2-R4 Rotor Mesh** – This mesh was generated on a two-blade section of the NASA SDT2-R4 rotor geometry.

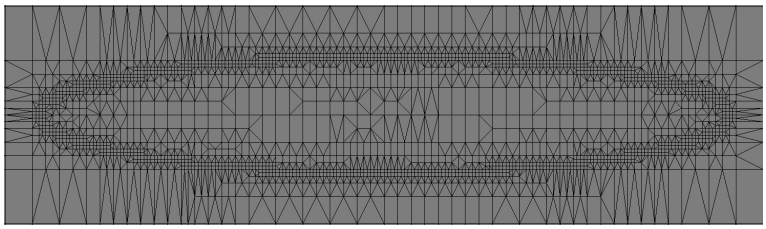


Figure 4.26: **Volume Mesh on the Geometry Boundaries Using Tensors From the Geometry** – The volume mesh which has been generated given the spacing parameters of the internal geometry of a 6:1 ellipsoid.

4.5 Overset Mesh Generation

Often, a solution algorithm does not have a need for a single discretized volume. In this case, it can solve the set of equations on multiple meshes and overlay the results, interpolating between "junction" nodes. This is called using an overset solution technique and overset meshes, as discussed in Chapter 1.2.2.

Fortunately, due to the decoupled nature of generation used in `FASTAR`, unstructured overset meshes can be very simple to make. First, the viscous layers are extruded from the viscous portions of the geometry and saved as a separate mesh with boundaries determined by the original viscous marked facets and the tops of the prisms on the outermost viscous layer. Then, a volume mesh is generated on the outer inviscid boundaries using Riemannian Metric Tensors based on the sizes of both viscous and inviscid marked facets.

This approach is excellent due to the fact that the region in which the interpolation will occur is properly sized to meet with the small viscous layers and geometry and make interpolation more accurate. In Figure 4.26, one can observe the volume mesh which has been generated given the spacing parameters of the internal geometry of a 6:1 ellipsoid.

In Figure 4.27, one can see the viscous layers that are extruded from the geometry by generating prisms at heights from the initial 1.0e-05 to a height of $\sum_{n=0}^{l-1} f^n s$, where l is the number of layers, f is the geometric progression factor, and s is the initial viscous spacing (here 1.0e-05). In Figure 4.28, a slice one third of the way through the ellipse shows the viscous prism layers cut away at different heights along the slice.

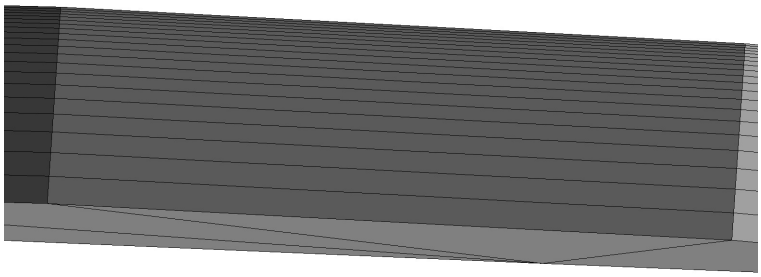


Figure 4.27: **Viscous Layers Generated on Geometry: Close up** – Close up of the viscous layers that are extruded from the geometry by generating prisms at heights from the initial 1.0e-15 to a height of $1.0e-15(g)^l$, where g is the geometric progression factor and l is the number of the layer.

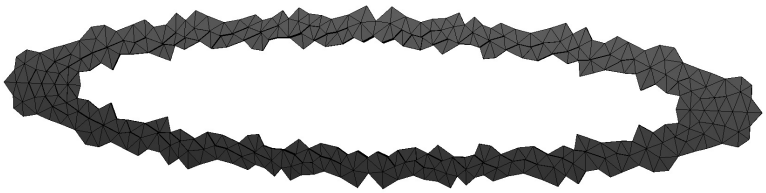


Figure 4.28: **Viscous Layers Generated on Geometry** – The viscous layers that are extruded from the geometry by generating prisms at heights from the initial 1.0e-05 to a height of $\sum_{n=0}^{l-1} f^n s$, where l is the number of layers, f is the geometric progression factor, and s is the initial viscous spacing (here 1.0e-05).

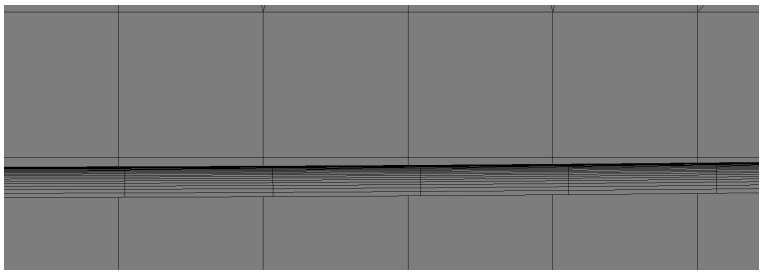


Figure 4.29: **Viscous Layers Overset on Volume Mesh: Close up** – The region of the viscous layers overlaying the central volume mesh.

Finally, one can see the region of the viscous layers overlaying the central volume mesh. Results would be interpolated between these two sections in evaluating a solution on the mesh.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

FASTAR has proven to be capable of generating large meshes on complex geometries, while allowing for non-unit aspect ratio mesh creation, which is important in a diverse set of arenas, including the hydrodynamics and aerodynamics cases presented in this dissertation. It is also exceptionally useful for generally keeping the node and element counts as low as possible, making rapid solution generation easier. The code uses a split-tree data structure to rapidly generate the volume mesh. Quality constraints are applied, voxels are deleted from the area cut by the geometry and outside the computational domain, and tetrahedralization of the remaining space occurs, followed by stitching meshes to create a grid that can be run through a flow solver.

FASTAR retains the exact tessellation supplied in the users geometry file, allowing for the final mesh to conform to the exact shape and still be decoupled from the flow solver being used. It also allows for overset mesh capability and stitching together of pieces of one large mesh (since the boundaries are recovered exactly). The code allows adaptive refinement to occur, helping the user capture the exact physics of their flow phenomena with repeated, automated generation in between solver runs. This is in contrast to P_REFINE which requires the user to coarsen and refine within the confines of the previously generated mesh. Also, FASTAR allows for extrusion of viscous layers without the need to insert and then smooth

the mesh to insert them after the initial generation, possibly losing Cartesian alignment with the flowfield solution.

Multiple real-world cases test the robustness of the algorithm, the ability to adapt to flow field features, as well as the ability to obtain solutions on the given grid, including a prolate sphere, a fully delineated Seafighter, a DPW IV cargo plane, a NACA 0015, and a rotor. `FASTAR` was able to generate solutions that were as accurate as those generated by commercial meshes, all while using less nodes and elements. Additionally, the use of the Cartesian elements prevents smearing of flowfield features. While all this may be enough to set `FASTAR` apart from other Cartesian mesh generators, the addition of a robust neighbor search algorithm which allows the tree to be traversed with only one criterion per level, the ability to generate easily coupled overset grids with appropriate spacing at the interpolation interface, and the speed with which tree traversal and octree-sorted lists can be traversed give it the potential to revolutionize mesh generation.

5.2 Future Work

5.2.1 Tetrahedral Mesh Generator

In order for `FASTAR` to generate a mesh for any size space between the stitching boundary and the geometry boundary (even for very complex, concave geometries), a robust tetrahedral mesh generator is being developed by Bruce Hilbert at the SimCenter. Unlike TetGen [Si, 2006], which uses a Delaunay criterion and Bowyer-Watson algorithm to generate tetrahedra and recover boundaries, the algorithm being developed is based on the edge swapping techniques of Lawson's algorithm. This should mitigate the need for box cutting, which can detriment the ability to get correct boundary layer calculations due to the random sizes of tetrahedra generated in a large region and somewhat defeats the purpose of using Cartesian grids around the geometry.

While this does not guarantee a perfect mesh, it does generate a valid mesh each time on these surfaces while preserving the boundaries. Also, the ability to add Steiner points to the boundaries to increase quality will be afforded as this library can be called from `FASTAR`, using its data structures. While currently boundaries must be preserved due to the

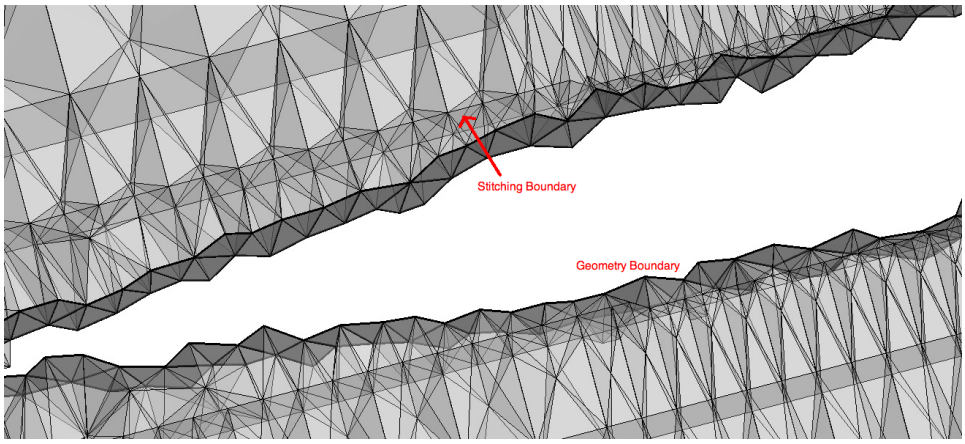


Figure 5.1: **Same Size Triangles on Stitching Boundaries** – On this NACA 0015 mesh, the triangles on the geometry and the boundary are nearly the same size ± 0.00025 units, thus giving TetGen a reasonable surface mesh and spacing to work with.

need to map the results back to the original surface meshes, the new library will be able to dynamically subdivide boundary triangles and remap the current configuration of both boundary elements and volume elements to which they are attached while generating the mesh. This also saves time on remapping nodes from the tetrahedral mesh to those in the full volume mesh.

The sizes of the inner and outer stitching boundary triangles are important in TetGen's ability to create a tetrahedralization of a difficult region successfully. As seen in Figures 5.1 and 5.2, the relative size of the stitching boundary triangles to the geometry facets improves TetGen's ability to generate a mesh when the outer boundary facets are larger or the same size. This is also partially due to the nature of a hierarchical Cartesian mesh, since larger triangles mean larger elements and thus they are further from the geometry and TetGen has more room to work; this situation also arises when many viscous layers are inserted and the projected triangles taking the geometry facets' place have grown in area significantly. While efforts were made to see if the reverse of this is true, TetGen was unable to generate a mesh on an outer boundary of triangles one fifth the size of the geometry facets, as seen in Figure 5.3 as the lightly shaded region between the Pointwise mesh and the polyhedral mesh.

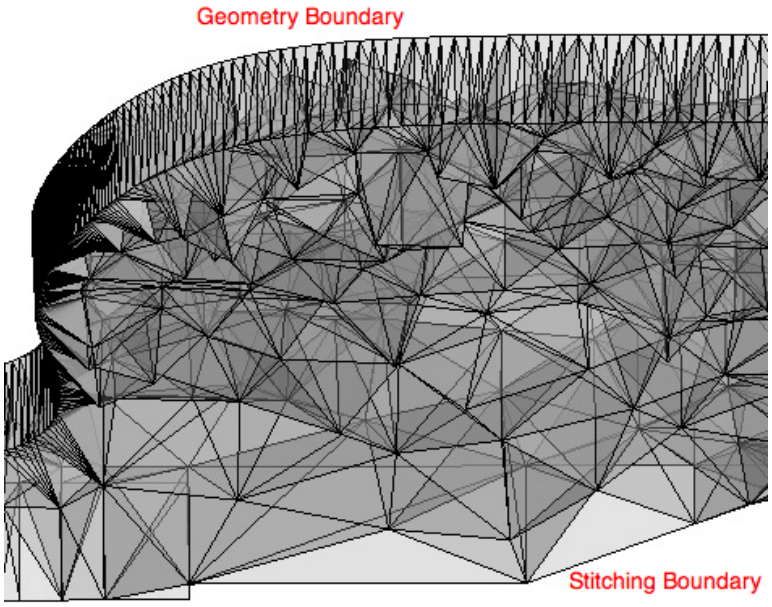


Figure 5.2: **Larger Triangles on Stitching Boundaries** – On this Seafighter mesh that was box cut, the triangles on the stitching boundary are much larger than the geometry facets ~ 5 times, thus giving TetGen enough space to insert points and gradate the tetrahedral mesh.

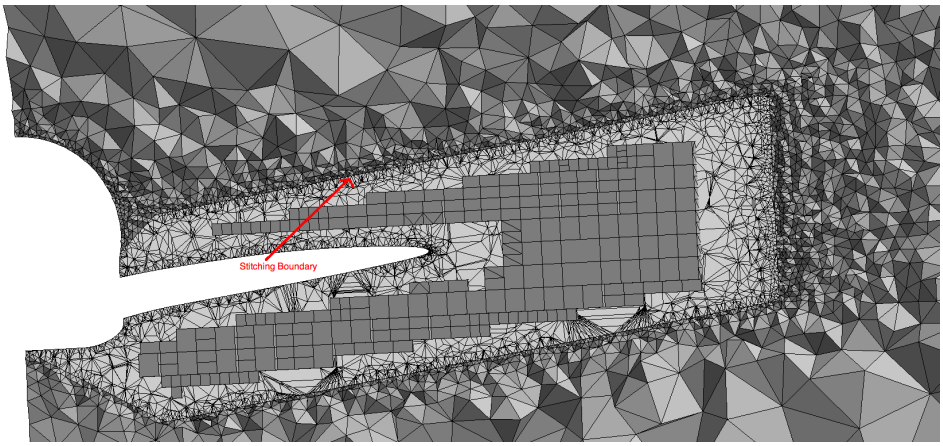


Figure 5.3: **Smaller Triangles on Stitching Boundaries** – On this DPW IV mesh, the triangles on the stitching boundary are much smaller than the viscous facets ~ 5 times, and TetGen was unable to generate a tetrahedral mesh and preserve the boundaries in this case.

5.2.2 Parallelization

As is posited by Dawes, parallelization is becoming required for all mesh generators due to the size and complexity of modern CAD geometries [Dawes, 2006]. For instance, on a machine with two dual-core Intel Xeon processors and 16 GB of RAM, the largest mesh that can be generated by `FASTAR` is approximately 20 million elements. While parallelization is necessary, it is very difficult to partition and load balance when the area being partitioned has yet to be discretized. Thus, speed to a good quality mesh is often not improved by parallelization of the mesh generation algorithm. However, the ability to spawn off processors as the discretization grows to allow for more room to generate large meshes is the major goal of parallelization of `FASTAR` [Betro, 2007].

While many considerations for eventual parallelization were implemented when designing the serial FASTAR, there are always obstacles to scalability when partitioning space without a predetermined discretization. Using lessons learned from `P_HUGG`, many modification have been made to mitigate tolerance issues. For instance, in `P_HUGG`, nodes are created in voxels on processor boundaries, and information about which processor and voxel should name and own the node is exchanged. However, issues arise in determining, within a tolerance, that the node is at the same location on both processors. Thus, in `FASTAR`, the fact that distinct nodes are not generated until the entire mesh has been created opens doors to re-partitioning on the fly and may alleviate some issues with tolerance across processor boundaries. Additionally, the use of simple data structures and the fact that portions of the mesh can be generated independently and then stitched allow for quick communication and some embarrassingly parallel behavior, meaning that the work can be divided with little or no communication necessary.

One issue that exists in `FASTAR` is a lack of symmetry in symmetric tensor generated meshes due to the order of operations during the quality control process. This order has been refined greatly to reduce speed and aid symmetry by performing desired quality constraint checks during the initial generation only, and then recursively checking for required quality constraints once the mesh was complete. However, this resulted in non-unit aspect ratio meshes being generated exclusively, due to the fact that there was no re-check for aspect

ratio after some cells had been refined in only one direction to rectify four-node-on-an-edge and crossbar issues. Implementing the aspect ratio check within this recursive loop required quality constraints check rectified the issue; however, due to the nature of propagation in unstructured hierarchical mesh generation, each time a four-node-on-an-edge fix was performed, an aspect ratio check then caused the situation to recur repeatedly. This was further underscored by the fact that the behavior occurred in the directions that are not refinement priority, y and z, and the fact that it would eventually gradate out in levels of hundreds of same level voxels until the issue was self-mitigated (see Figure 5.4).

By changing the order of operations to require an aspect ratio check after any type of refinement, this can be mitigated. However, the mesh may come out to be much larger than desired if unit aspect ratio is a hard requirement. Another option to fix this issue was used in the parallelization of `HUGG`–pass all the geometry facets and spacing tensors through in order, generate the appropriate refinement, perform quality constraints, and then repeat cycle level by level until minimum spacing is reached. While neither is a guaranteed fix and in fact a slightly higher than unit aspect ratio in the far field is not far from optimal, these options will be explored for speed and eventual parallelization.

5.2.3 Dynamic Meshing

The ability to do local remeshing on a geometry given adaptation parameters is requisite for a robust dynamic mesh generation algorithm [Loseille et al., 2009]. The retention of a coherent tree by `FASTAR` makes this type of local remeshing possible and saves a great deal of time between flow solutions and adaptation steps by avoiding the need to coarsen and optimize the entire mesh.

If one generates a mesh and then needs to adapt a certain area or move the geometry, the tree can be traversed to determine only the affected voxels and let the work be done there only. Also, the use of voxel deletion instead of cutting opens doors to moving within a pre-determined box and only needed to re-tetrahedralize the region between the geometry or viscous layers and the borders of the box, as seen in Figure 5.5.

This could be accomplished by generating a mesh, evolving a solution at a given time step through adaptation, and then moving the geometry and repeating the process. There

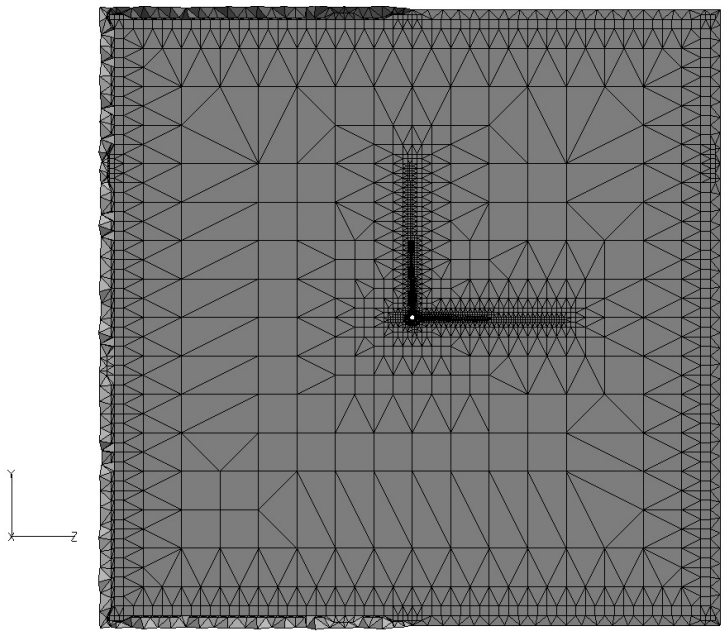


Figure 5.4: **Order of Operations in Quality Constraints** – On this ellipsoid mesh, the spikes one sees propagating from the ellipsoid to the far field in the y and z directions are artifacts of the order of operations causing over-refinement to balance four-to-one-on-an-edge and crossbar situations being fixed with aspect ratio constraints undoing the operation over and over again.

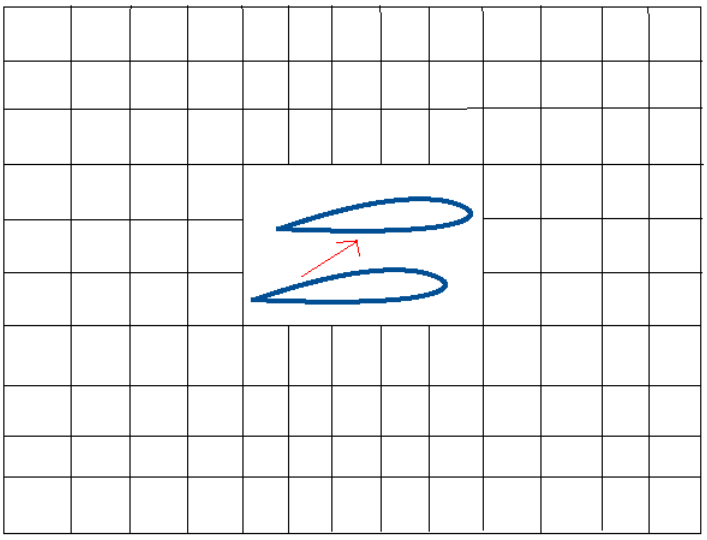


Figure 5.5: **Dynamic Meshing by Moving Geometry Within a Box** – If desired, a bounding box could be put around the region in which the geometry is moving, viscous layers could be added to the geometry, and the area left undiscretized would simply be meshed using tetrahedra.

would be library calls to `FASTAR` and Tenasi from a wrapper code that would handle determining when the adaptation was no longer changing the solution, moving the geometry, and interpolating results as the mesh changes.

Bibliography

Bibliography

[ACAD, 2010] Lockheed Martin Aerospace Company. Proprietary CAD program. Fort Worth, TX. 2010.

[Aftosmis et al., 1997] Aftosmis, M.J., Berger, M.J., Melton, J.E. (1997). Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry. In *National Aeronautics and Space Administration Technical Reports (NASA, Moffet Field, CA 1997).*

[Beatty et al., 2008] Beatty, K. and Mukherjee, N. (2008). Flattening 3D Triangulations for Quality Surface Mesh Generation. In *Proceedings of the 17th International Meshing Roundtable (Springer, Berlin).*

[Berger et al., 1998] Berger, M.J., and Aftosmis, M.J. (1998). Aspects (and Aspect Ratios) of Cartesian Mesh Methods. In *Proceedings of the 16th International Conference on Numerical Methods in Fluid Dynamics (Springer, Heidelberg).*

[Betro, 2007] Betro, Vincent C. (2007). Parallel Hierarchical 2D Unstructured Mesh Generation with General Cutting. Master's Thesis, University of Tennessee at Chattanooga, August, 2007.

[Bibb et al., 2006] Bibb, K.L., Gnoffo, P.A., Park, M.A., Jones, W.T. (2006). Parallel, Gradient-Based Anisotropic Mesh Adaptation for Re-entry Vehicle Configurations. At *9th AIAA/ASME Joint Thermophysics and Heat Transfer Conference (AIAA/ASME, San Francisco, CA June 5-8, 2006).*

[Blacker et al., 1991] Blacker, T.D. and Stephenson, M.B. (1991). Paving: A New Approach to Automated Quadrilateral Mesh Generation. In *International Journal for Numerical Methods in Engineering, Volume 32, pp. 811-847.*

[CART3D, 2010] Aftosmis, M.J. (2010). Retrieved May 7, 2010, from http://people.nas.nasa.gov/ aftosmis/cart3d.

[CEI, 2010] Sharc Limited. (2010). Harpoon brief. Retrieved May 7, 2010, from http://www.sharc.co.uk/index.html.

[CFD-VISCART, 2010] Retrieved May 7, 2010, from http://www.cfdrc.com/serv_prod/cfd_multiphysics/software/ace/viscart.html.

[CGNS, 2010] CGNS Documentation. (2010). "The CFD General Notation System Standard Interface Data Structures." Retrieved May 7, 2010, from http://www.grc.nasa.gov/WWW/cgns/beta/sids/sids.pdf.

[Chalasani et al., 2005] Chalasani, S., Luke, E.A., Senguttuvan, V., Thompson, D.S. (2005). Assessing Generalized Mesh Quality via CFD Solution Validation. At *43rd Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 10-13, 2005).*

[Coirier et al., 1996] Coirier, W.J., and Powell, K.G. (1996). Solution-Adaptive Cartesian Cell Approach for Viscous and Inviscid Flows. *AIAA American Institute of Aeronautics and Astronautics Journal, Volume 34, Number 5 (AIAA, Reston, VA 1996)*, pp. 938-945.

[Dawes, 2006] Dawes, W.N. (2006). Towards a fully parallel integrated geometry kernel, mesh generator, flow solver, and post-processor. At *44th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 9-12, 2006).*

[Dawes et al., 2007] Dawes, W.N., Harvey, S.A., Fellows, S., Favaretto, C.F., and Velivelli, A. (2007). Viscous Layer Meshes from Level Sets on Cartesian Meshes. At *45th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 8-11, 2007).*

[Dawes et al., 2009] Dawes, W.N., Harvey, S.A., Fellows, S., Eccles, N., Jaeggi, D., and Kellar, W.P. (2009). A practical demonstration of scalable, parallel mesh generation. At *47th Aerospace Sciences Meeting and Exhibit (AIAA, Orlando, FL January 5-8, 2009).*

[Dawes, Kellar, et al., 2009] Dawes, W.N., Kellar, W.P., and Harvey, S.A. (2009). Using Level Sets as the basis for a scalable, parallel geometry engine and mesh generation system. At *47th Aerospace Sciences Meeting and Exhibit (AIAA, Orlando, FL January 5-8, 2009).*

[Delaunay, 1934] Delaunay, N. (1934). Sur la Sphere Vide. In *Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk, Volume 7, pp.793-800.*

[Djomehri et al., 2003] Djomehri, M., Biswas, R., and Lopez-Benitez, N. (2003). Load Balancing Strategies for Multi-Block Overset Grid Applications. In *National Aeronautics and Space Administration Advanced Supercomputing Technical Reports, Number 7 (NASA, Moffet Field, CA 2003).*

[Domel et al., 2000] Domel, N.D., Karman, Jr., S.L. (2000). Splitflow: Progress in 3D CFD with Cartesian Omni-tree Grids for Complex Geometries (2000-1006). At *38th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 10-13, 2000).*

[Fieldview, 2010] Intelligent Light, Inc. Retrieved May 7, 2010. http://www.ilight.com.

[George et al., 1991] George, P.L., Hecht, F., and Saltel, E. (1991). Automatic Mesh Generator with Specified Boundary. In *Computer Methods in Applied Mechanics and Engineering, Volume 92 (North-Holland 1991), pp. 269-288.*

[Huang, 2004] Huang, Weizhang. (2004). Metric tensors for anisotropic mesh generation. In *Journal of Computational Physics, Volume 204 (Academic Press, Amsterdam 2004)*, pp. 633-655.

[Ishida et al., 2008] Ishida, T., Takahashi, S., and Nakahashi, K. (2008). Fast Cartesian Mesh Generation for Building-Cube Method using Multi-Core PC. At *46th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 7-10, 2008).*

[Ishida et al., 2009] Ishida, T., Takahashi, S., and Nakahashi, K. (2009). Flow Computations Around Moving and Deforming Bodies Using Cartesian Mesh. *AIAA American Institute of Aeronautics and Astronautics Journal (AIAA, Reston, VA 2009).*

[Ito et al., 2004] Ito, Y. and Nakahashi, K. (2004). Improvements in the reliability and quality of unstructured hybrid mesh generation. In *International Journal For Numerical Methods in Fluids, Volume 45 (John Wiley and Sons, Ltd. Hoboken, NJ 2004)*, pp. 79-108.

[Joe, 1992] Joe, B. (1992). Three-dimensional boundary-constrained triangulations. In *Artificial Intelligence, Expert Systems, and Symbolic Computing – Proceedings of the 13th IMACS World Congress, ed. E. N. Houstis and J. R. Rice, (Elsevier Science Publishers 1992), pp. 215-222.*

[Karman, 1995 (AIAA)] Karman, Jr., Steve L. (1995). SPLITFLOW: A 3D Unstructured Cartesian/Prismatic Grid CFD Code for Complex Geometries (1995-0343). At *33rd Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 9-12, 1995).*

[Karman, 1995 (NASA)] Karman, Jr., Steve L. (1995). Unstructured Cartesian/Prismatic Grid Generation for Complex Geometries. In *National Aeronautics and Space Administration Conference Publication 3291: Surface Modeling, Grid Generation, and Related Issues in Computational Fluid Dynamic (CFD) Solutions (NASA, Moffet Field, CA 1995).*

[Karman, 2004] Karman, Jr., Steve L. (2004). Hierarchical Unstructured Mesh Generation (2004-0613). At *42nd Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 4-7, 2004).*

[Karman et al., 2007] Karman, Jr., S.L., Anderson, W.K., and Sahasrabudhe, M. (2007). Mesh Generation Using Unstructured Computational Meshes and Elliptic Partial Differential Equation Smoothing (2007-0559). *AIAA American Institute of Aeronautics and Astronautics Journal, Volume 45, Number 6 (AIAA, Reston, VA 2007)*, pp. 1277-1286.

[Karman et al., 2007 (AIAA-ASM)] Karman, Jr., S.L., and Sahasrabudhe, M. (2007). Unstructured Adaptive Elliptic Smoothing (2007-0559). *45th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV 2007)January 8-11, 2007).*

[Karman, 2007] Karman, Jr., S.L. (2007). Unstructured Viscous Layer Insertion Using Linear-Elastic Smoothing. *AIAA American Institute of Aeronautics and Astronautics Journal, Volume 45, Number 1 (AIAA, Reston, VA 2007)*, pp. 168-180.

[Karman et al., 2008] Karman, Jr., S. L. and Betro, V. C. (2008). Parallel Hierarchical Unstructured Mesh Generation with General Cutting (2008-0918). At *46th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 7-10, 2008).*

[Karman et al., 2009] Karman, Jr., S. L. and Wooden, P. (2009). CFD Modeling of F-35 Using Hybrid Unstructured Meshes (2009-3662). At *47th Aerospace Sciences Meeting and Exhibit (AIAA, Orlando, FL January 5-8, 2009).*

[Lahur et al., 2001] Lahur, P.R. and Nakamura, Y. (2001). Anisotropic Grid Adaptation. In *The Japan Society for Aeronautical and Space Sciences Journal, Volume 44, Number 143 (JSASS, Tokyo 2001)*, pp. 31-39.

[Lawson, 1977] Lawson, C.L. (1977). Software for C1 Surface Interpolation. In *Mathematical Software III, pp.161-194.*

[Liu et al., 2000] Liu, C.Y. and Hwang, C.J. (2000). A New Strategy for Unstructured Mesh Generation. At *Fluids 2000 (AIAA, Denver, CO June 19-22, 2000).*

[Lohner et al., 1988] Lohner, R., Parikh, P., and Gumbert, C. (1988). Interactive Generation of Unstructured Grid for Three Dimensional Problems. In *Numerical Grid Generation in Computational Fluid Mechanics 88, (Pineridge Press 1988), pp.687-697.*

[Lohner, 1996] Lohner, R. (1996). Progress in Grid Generation via the Advancing Front Technique. In *Engineering with Computers, Volume 12, pp.186-210.*

[Loseille et al., 2009] Loseille, A. and Lohner, R. (2009). On 3D Anisotropic Local Remeshing for Surface, Volume, and Boundary Layers. In *Proceedings of the 18th International Meshing Roundtable (Springer, Berlin).*

[Luo et al., 2008] Luo, H., Chen, G., and Lohner, R. (2008). A Hybrid Grid Generation Method for Complex Geometries. At *46th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 7-10, 2008).*

[Maple, 2002] Waterloo Maple, Inc. (Maplesoft). Maple 8.0. Retrieved June 11, 2010. http://www.maplesoft.com.

[MATLAB, 2009] The MathWorks, Inc. MATLAB 2009b. Retrieved June 11, 2010. http://www.mathworks.com.

[Morrell et al., 2007] Morrell, J.M., Sweby, P.K., and Barlow, A. (2007). A cell by cell anisotropic adaptive mesh ALE scheme for the numerical solution of the Euler equations. In *Journal of Computational Physics, Volume 2226 (Academic Press, Amsterdam 2007)*, pp. 1152-1180.

[O'Shea, et al., 2008] O'Shea, T., Brucker, K., Dommermuth, D. and Wyatt, D. (2008). A Numerical Formulation for Simulating Free-Surface Hydrodynamics. At *27th Symposium on Naval Hydrodynamics, Seoul, Korea, 2008.*

[Owen, 2007] Owen, S.J. (2007). A Survey of Unstructured Mesh Generation Technology. Carnegie Mellon University, November 4, 2007.

[Park, 2008] Park, Michael A. (2008). Anisotropic Output-Based Adaptation with Tetrahedral Cut Cells for Compressible Flows. PhD Dissertation, Massachusetts Institute of Technology, September, 2008.

[Pointwise, 2010] Pointwise, Inc. (2010). Pointwise User Manual, version 16. Retrieved May 7, 2010, from http://www.pointwise.com.

[Ruppert, 1995] Ruppert, Jim. (1995). A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. In *Journal of Algorithms, Volume 18, Number 3 (Academic Press, New York, NY 1995)*, pp. 548-585.

[Sahasrabudhe, 2006] Sahasrabudhe, Mandar S., Karman, Jr., S.L., Anderson, W.K. (2006). Grid Control of Viscous Unstructured Meshes Using Optimization (2006-0532). At *44th Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 9-12, 2006).*

[Shephard, 1991] Shephard, M. and Georges, M.K. (1991). Three-Dimensional Mesh Generation by Finite Octree Technique. In *International Journal for Numerical Methods in Engineering, Volume 32, pp. 709-749.*

[Shepherd, 2009] Shepherd, J.F. (2009). Conforming Hexahedral Mesh Generation via Geometric Capture Methods. In *Proceedings of the 18th International Meshing Roundtable (Springer, Berlin).*

[Shewchuk (Adaptive), 1996] Shewchuk, Jonathan Richard. (1996). Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Carnegie Mellon University, May 17, 1996.

[Shewchuk (Triangle), 1996] Shewchuk, Jonathan Richard. (1996). Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. Carnegie Mellon University, May 17, 1996.

[Si, 2006] Si, Hang. (2006). TetGen: A Quality Tetrahedral Mesh Generator and Three Dimensional Delaunay Triangulator (Version 1.4 Users Manual). http://tetgen.berlios.de. January 18, 2006.

[Simpson, 2000] Simpson, Roger L. (2000). Steady Flow Over a 6:1 Prolate Spheroid. http://www.aoe.vt.edu/ simpson/prolatespheroid/.

[Staten et al., 2008] Staten, M., Shepherd, J., Shimada, K. (2008). Mesh Matching– Creating Conforming Interfaces Between Hexahedral Meshes. In *Proceedings of the 17th International Meshing Roundtable (Springer, Berlin).*

[Taylor et al., 1995] Taylor, L. K., Arabshahi, A., and Whitfield, D. L. (1995). Unsteady Three-Dimensional Incompressible Navier-Stokes Computations for a 6:1 Prolate Spheroid Undergoing Time-Dependent Maneuvers (1995-0313). At *33rd Aerospace Sciences Meeting and Exhibit (AIAA, Reno, NV January 9-12, 1995).*

[Thompson et al., 1999] Thompson, J.F., Soni, B., and Weatherhill, N., editors. Aftosmis, M.J., Berger, M.J., and Melton, J.E., chapter authors. (1999). Handbook of Grid Generation. CRC Press LLC, Boca Raton, FL, pages 22-1 to 22-26.

[Varghese, 2009] Varghese, Jacob C. (2009). Sonic Boom Prediction Methods Using Feature-Based Adaptation of Unstructured Meshes. Master's Thesis, University of Tennessee at Chattanooga, December, 2009.

[Voxel, 2007] Voxel. (2007). Retrieved May 7, 2010, from http://en.wikipedia.org/wiki/Voxel.

[Vyas et al., 2009] Vyas, V. and Shimada, K. (2009). Tensor-Guided Hex-Dominant Mesh Generation with Targeted All-Hex Regions. In *Proceedings of the 18th International Meshing Roundtable (Springer, Berlin).*

[Wang et al., 2004] Wang, Z.J., Srinivasan, K., Sun, R., and Yuan, W. (2004). Vehicle Thermal Management Simulation Using a Rapid Omni-tree Based Adaptive Cartesian Mesh Generation Methodology. At *2004 ASME Heat Transfer/Fluids Engineering Summer Conference (ASME, Charlotte, NC July 11-15, 2004).*

[Watson, 1981] Watson, D.F. (1981). Computing the Delaunay Tesselation with Application to Voronoi Polytopes. In *The ComputerJournal, Volume 24, Number 2, pp.167-172.*

[Weatherill, 1994] Weatherill, N.P. and Hassan, O. (1994). Efficient Three-dimensional Delaunay Triangulation with Automatic Point Creation and Imposed Boundary Constraints In *International Journal for Numerical Methods in Engineering, Volume 37, pp. 2005-2039.*

[Welterlen, 2000] Welterlen, Tracy J. (2000). Weapons Bay Flow Field Simulation (2000-3926). At *18th Applied Aerodynamics Specialist Conference and Exhibit (AIAA, Denver, CO August 14-17, 2000).*

[Yerry, 1984] Yerry, M. and Shephard, M. (1984). Three-Dimensional Mesh Generation by Modified Octree Technique. In *International Journal for Numerical Methods in Engineering, Volume 20, pp. 1965-1990.*

Appendix

Chapter 6

Appendix

6.1 Riemannian Metric Tensor Derivation

The Maple code in Figure 6.1 shows the manipulation of (2.1) to obtain an $A\vec{x} = \vec{b}$ system. This system is solved with singular value decomposition and back substitution due to conditioning, and the end product is a Reimannian Metric Tensor for a given tetrahedron. This tetrahedron was created using a surface facet and projecting a normal of a given length from its centroid, using the endpoint as the fourth vertex.

```
DP:=array(1..3,1..6,[[x3-x1,x3-x2,x2-x1,x4-x3,x4-x2,x4-x1],[y3-y1,
y3-y2,y2-y1,y4-y3,y4-y2,y4-y1],[z3-z1,z3-z2,z2-z1,z4-z3,z4-z2,z4-z
1]]);
```

$$DP := \begin{bmatrix} x3-x1 & x3-x2 & x2-x1 & x4-x3 & x4-x2 & x4-x1 \\ y3-y1 & y3-y2 & y2-y1 & y4-y3 & y4-y2 & y4-y1 \\ z3-z1 & z3-z2 & z2-z1 & z4-z3 & z4-z2 & z4-z1 \end{bmatrix}$$

```
DM:=array(1..3,1..3,[[M11,M12,M13],[M21,M22,M23],[M31,M32,M33]]);
```

$$DM := \begin{bmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{bmatrix}$$

Figure 6.1: **Maple Derivation of RMT System.**

```
> DB:=multiply(transpose(DP),DM);
```

$$DB :=$$

$$[(x3 - x1)\,M11 + (y3 - y1)\,M21 + (z3 - z1)\,M31,\ (x3 - x1)\,M12 + (y3 - y1)\,M22 + (z3 - z1)\,M32,$$
$$(x3 - x1)\,M13 + (y3 - y1)\,M23 + (z3 - z1)\,M33]$$
$$[(x3 - x2)\,M11 + (y3 - y2)\,M21 + (z3 - z2)\,M31,\ (x3 - x2)\,M12 + (y3 - y2)\,M22 + (z3 - z2)\,M32,$$
$$(x3 - x2)\,M13 + (y3 - y2)\,M23 + (z3 - z2)\,M33]$$
$$[(x2 - x1)\,M11 + (y2 - y1)\,M21 + (z2 - z1)\,M31,\ (x2 - x1)\,M12 + (y2 - y1)\,M22 + (z2 - z1)\,M32,$$
$$(x2 - x1)\,M13 + (y2 - y1)\,M23 + (z2 - z1)\,M33]$$
$$[(x4 - x3)\,M11 + (y4 - y3)\,M21 + (z4 - z3)\,M31,\ (x4 - x3)\,M12 + (y4 - y3)\,M22 + (z4 - z3)\,M32,$$
$$(x4 - x3)\,M13 + (y4 - y3)\,M23 + (z4 - z3)\,M33]$$
$$[(x4 - x2)\,M11 + (y4 - y2)\,M21 + (z4 - z2)\,M31,\ (x4 - x2)\,M12 + (y4 - y2)\,M22 + (z4 - z2)\,M32,$$
$$(x4 - x2)\,M13 + (y4 - y2)\,M23 + (z4 - z2)\,M33]$$
$$[(x4 - x1)\,M11 + (y4 - y1)\,M21 + (z4 - z1)\,M31,\ (x4 - x1)\,M12 + (y4 - y1)\,M22 + (z4 - z1)\,M32,$$
$$(x4 - x1)\,M13 + (y4 - y1)\,M23 + (z4 - z1)\,M33]$$

Figure 6.1 (continued). Maple Derivation of RMT System.

```
> multiply(DB,DP);
```

$$
\begin{aligned}
&[((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x3-x1)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y3-y1)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z3-z1),\\
&\quad((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x3-x2)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y3-y2)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z3-z2),\\
&\quad((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x2-x1)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y2-y1)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z2-z1),\\
&\quad((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x4-x3)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y4-y3)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z4-z3),\\
&\quad((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x4-x2)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y4-y2)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z4-z2),\\
&\quad((x3-x1)\,M11+(y3-y1)\,M21+(z3-z1)\,M31)\,(x4-x1)\\
&\quad+((x3-x1)\,M12+(y3-y1)\,M22+(z3-z1)\,M32)\,(y4-y1)\\
&\quad+((x3-x1)\,M13+(y3-y1)\,M23+(z3-z1)\,M33)\,(z4-z1)]\\
&\quad[((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x3-x1)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y3-y1)\\
&\quad+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z3-z1),\\
&\quad((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x3-x2)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y3-y2)\\
&\quad+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z3-z2),\\
&\quad((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x2-x1)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y2-y1)\\
&\quad+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z2-z1),\\
&\quad((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x4-x3)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y4-y3)\\
&\quad+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z4-z3),\\
&\quad((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x4-x2)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y4-y2)\\
&\quad+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z4-z2),\\
&\quad((x3-x2)\,M11+(y3-y2)\,M21+(z3-z2)\,M31)\,(x4-x1)\\
&\quad+((x3-x2)\,M12+(y3-y2)\,M22+(z3-z2)\,M32)\,(y4-y1)
\end{aligned}
$$

Figure 6.1 (continued). Maple Derivation of RMT System.

$$
\begin{aligned}
&+((x3-x2)\,M13+(y3-y2)\,M23+(z3-z2)\,M33)\,(z4-z1)]\\
&[((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x3-x1)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y3-y1)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z3-z1),\\
&((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x3-x2)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y3-y2)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z3-z2),\\
&((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x2-x1)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y2-y1)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z2-z1),\\
&((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x4-x3)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y4-y3)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z4-z3),\\
&((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x4-x2)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y4-y2)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z4-z2),\\
&((x2-x1)\,M11+(y2-y1)\,M21+(z2-z1)\,M31)\,(x4-x1)\\
&+((x2-x1)\,M12+(y2-y1)\,M22+(z2-z1)\,M32)\,(y4-y1)\\
&+((x2-x1)\,M13+(y2-y1)\,M23+(z2-z1)\,M33)\,(z4-z1)]\\
&[((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x3-x1)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y3-y1)\\
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z3-z1),\\
&((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x3-x2)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y3-y2)\\
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z3-z2),\\
&((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x2-x1)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y2-y1)\\
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z2-z1),\\
&((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x4-x3)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y4-y3)\\
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z4-z3),\\
&((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x4-x2)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y4-y2)\\
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z4-z2),\\
&((x4-x3)\,M11+(y4-y3)\,M21+(z4-z3)\,M31)\,(x4-x1)\\
&+((x4-x3)\,M12+(y4-y3)\,M22+(z4-z3)\,M32)\,(y4-y1)
\end{aligned}
$$

Figure 6.1 (continued). Maple Derivation of RMT System.

$$
\begin{aligned}
&+((x4-x3)\,M13+(y4-y3)\,M23+(z4-z3)\,M33)\,(z4-z1)]\\
&[((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x3-x1)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y3-y1)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z3-z1),\\
&((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x3-x2)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y3-y2)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z3-z2),\\
&((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x2-x1)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y2-y1)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z2-z1),\\
&((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x4-x3)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y4-y3)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z4-z3),\\
&((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x4-x2)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y4-y2)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z4-z2),\\
&((x4-x2)\,M11+(y4-y2)\,M21+(z4-z2)\,M31)\,(x4-x1)\\
&+((x4-x2)\,M12+(y4-y2)\,M22+(z4-z2)\,M32)\,(y4-y1)\\
&+((x4-x2)\,M13+(y4-y2)\,M23+(z4-z2)\,M33)\,(z4-z1)]\\
&[((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x3-x1)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y3-y1)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z3-z1),\\
&((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x3-x2)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y3-y2)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z3-z2),\\
&((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x2-x1)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y2-y1)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z2-z1),\\
&((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x4-x3)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y4-y3)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z4-z3),\\
&((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x4-x2)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y4-y2)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z4-z2),\\
&((x4-x1)\,M11+(y4-y1)\,M21+(z4-z1)\,M31)\,(x4-x1)\\
&+((x4-x1)\,M12+(y4-y1)\,M22+(z4-z1)\,M32)\,(y4-y1)\\
&+((x4-x1)\,M13+(y4-y1)\,M23+(z4-z1)\,M33)\,(z4-z1)]
\end{aligned}
$$

Figure 6.1 (continued). Maple Derivation of RMT System.

Vita

Vincent Charles Betro was born in Salem, Ohio, on June 28, 1980. After attending Bartlett High School in Bartlett, TN, he pursued an undergraduate degree in Secondary Mathematics at The University of Tennessee at Chattanooga, where he was a member of the University Honors program. After completing his degree in December 2002, he taught mathematics at Ooltewah Middle School and Meigs County High School before returning to UTC in January 2005 to accept a position as adjunct instructor in the Developmental Mathematics Department. At UTC, he worked with the Web Homework System (www.mathclass.org) in conjunction with The University of Kentucky and UTC professors Dr. Stephen Kuhn and Dr. Terry Walters. From 2005 until 2009, he was an adjunct lecturer in the math department, where he taught college algebra and precalculus. He received his Master's Degree in 2007 in Computational Engineering, and in 2009, he became the STEM Outreach Coordinator at the SimCenter while finishing his doctoral research.