AIAA SciTech Forum
6-10 January 2020, Orlando, FL
AIAA Scitech 2020 Forum

10.2514/6.2020-1408

Implementation of a Size Field Based Isotropic Hex Core Mesh Algorithm

John P. Steinbrenner[1]
Nick J. Wyman[2]
Mike S. Jefferies[3]
Steve L. Karman, Jr.[4]

Pointwise, Inc., 213 South Jennings Avenue, Fort Worth, TX 76104, USA

Jeremy Shipman[5]

Combustion Research and Flow Technology, Pipersville, PA 18947, USA

This paper introduces an isotropic hex core meshing scheme based on a scalar function size field defined by user-defined shape primitives, boundary meshes and/or solution adaptation cloud point data. Meshing starts with a small number of root voxel elements, which are then refined via an octree framework to the size field specification. Voxel elements are classified as either cutting, inside or outside of the prescribed surface mesh, resulting in sets of elements wholly outside the surface grid boundaries (external mode) and inside any defined outer boundaries (internal mode). Voxels are then converted to hex core (tet, pyramid and hex) form, processed and finally connected to isoptropic tet cells lying between hex core and surface grid regions. Examples of hex core meshes are provided.

Nomenclature

voxel	hierarchical cubic cell stored in an octree structure
hex core	tet, pyramid and hex grid elements converted from voxels
Δs_{min}	minimum size specified in size field
Δs_{max}	maximum size specified in size field
transLayers	user-specified minimum number of voxels between refinement levels
Δv	voxel edge length
In	voxel identified as lying inside the meshing region
Out	voxel identified as lying outside the meshing region
Cut	voxel cutting part of the boundary mesh

I. Introduction

Three decades ago, structured (mapped) grids were the predominant mesh type in CFD research and applications. The smoothness of the structured mesh curves that were generally aligned with the near-body flowfield allowed for accurate numerical computation of the Navier-Stokes equations using finite-difference formulations. Finite volume formulations arrived on the scene shortly thereafter, initially using (but not limited to) the same structured mesh formulations. As the complexity of the geometries meshed increased, the construction of multi-block structured grids quickly became the bottleneck for solution turnaround, due to the difficult topological constraints placed by abutting blocks in space. These constraints launched research into more efficient grid methods, namely unstructured tetrahedral meshes, which could be used to fill a general void in an automated manner. Structured and unstructured grid methods continued to be developed, and in fact the first full aircraft flow simulations using each grid type were presented within a few months of each other in 1986.[1,2]

[1] Vice-President, Research and Development, AIAA Associate Fellow
[2] Director, Applied Research, AIAA Associate Fellow
[3] Manager, Product Development
[4] Staff Specialist, Applied Research, AIAA Associate Fellow
[5] Senior Research Scientist, AIAA Senior Member

From that time, unstructured meshing methods gained traction very quickly. Support for viscous flow meshing came shortly after with the introduction of prismatic layer techniques[3], which was quickly copied and improved while gaining in popularity. Even today, many viscous meshing tools embed high aspect ratio prisms and hex layers on the geometries, filling remaining voids with unstructured tets.

Though these methods can be applied with an ever-increasing degree of automation, with much faster turnaround than a structured grid technique, using tetrahedra on the flow domain interior remains a poor replacement for structured cells in terms of quality and flow solution accuracy. Further, tet meshes are fairly expensive to generate and their solution accuracy can degrade to 1st order when the flow solution Hessian matrix is non-zero across the control volume due to a lack of central symmetry in the mesh and gradient operators (first order truncation errors are non-cancelling).[4,5] Structured grids, on the other hand, contain strong central symmetry which leads to the cancellation of truncation error.

Rather than return to a structured grid form, this paper proposes replacing the bulk of the unstructured tets with Cartesian-aligned hex core meshes formed from voxel meshes, which preserve the truncation error cancellation afforded by structured grids. As will be shown, voxels are refined locally based on values obtained from a scalar size field function, defined by shape primitives, boundary meshes and/or solution adaptation cloud point data. Voxels are then classified as either cutting the surface meshes, or lying inside or outside of the boundaries. This provides for a truncated set of voxels that lie completely within the surface boundary meshes. Remaining voids are then filled with unstructured tets, providing a watertight mesh with cells aligned with the boundary as well as interior cells aligned with the free stream.

This voxel mesh method will be explained and demonstrated via example meshes, and its effectiveness will be shown by numerical comparison between a hybrid mesh with isotropic tets and a hybrid mesh with hex core cells.

II. Size Fields

A size field can be defined as a continuous scalar function defining the expected isotropic element edge length at a point in space. In general, the range of values in a given size field will span several orders of magnitude, due primarily to differing length scales in the field solution and the fact that a uniform mesh resolving small scale phenomena would be prohibitively expensive. For example, a uniform mesh on an external flowfield might easily require viscous spacing 6-7 orders of magnitude smaller than the geometry, and 7-8 orders smaller than the flowfield extents. A uniform mesh would then require O(7*3), or sextillion cells. On the other hand, with a well-constructed size field, cell count can be reduced to manageable numbers.

In this effort, size fields are ultimately defined by a cloud of discrete points, each specifying a position in space as well as an expected isotropic element size. Evaluation of the size field at a point in space from a cloud of points is discussed below.

There are three different types of contributing components that are used to define the overall field. The first of these, **shape primitives**, prescribe the size field in geometric regions of the field bound by their shapes. Numerous shape primitives can be formed, including boxes, cones, cylinders, spheres and extruded polygons. Spatial arrangement and size field intensities of shape primitives are normally user-defined, but can also be scripted for parametric application to geometry classes with expected small scale flow phenomena such as wake regions (Figure 1). Further, size fields can be adjusted to vary across the primitive shape. Each primitive is imprinted into the size field automatically via conversion into discrete cloud point data.

The second contributor type defining a scalar size field are **mesh components,** defined as curve and surface meshes existing on, inside or in proximity to the region to be meshed. By default, a size field applied to a volume mesh will have each of its bounding curve and surface entities applied as mesh component contributors, so that the size field defined at the mesh boundaries will coincide with the true surface cell size. Mesh components outside of the meshing region may also be specified, acting as influence entities to provide size continuity across meshing regions. Each surface cell in a mesh component is added to the size field as a single discrete point positioned at the triangle or quad centroid and equal in value to the equivalent edge length of an equilateral triangle or square of equal area.

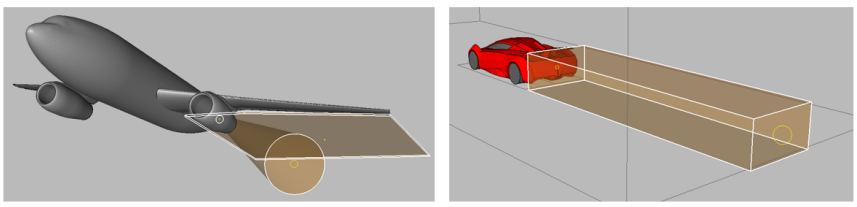


Figure 1. Shape Primitives Applied in Wake Regions.

The third contributor type is referred to as an **adaptation sensor.** As the name suggests, this type is used to allow a mesh to adapt to a previously computed flow solution. The adaptation sensor is typically chosen with the expectation that the local sensor value will be inversely related to the desired cell size. Sensors can be obtained in a number of ways, such as from evaluation of the gradient vector or Hessian matrix representations of solution variables such as pressure or vorticity, via solution error obtained by interpolation onto finer meshes, or by computing solution adjoint data producing sensitivities to changes in input. In each case the number of source points added to the size field will be a small subset of the entire solution, with the user selecting a threshold value to influence the size field. An example of size field-based adaptation applied to an unstructured tet mesh is provided in Ref 6.

The point cloud representation of all 3 contributor types blend to form a size field containing influence from all components. Figure 2. depicts a simple size field colored by size and containing shape primitives and mesh components.

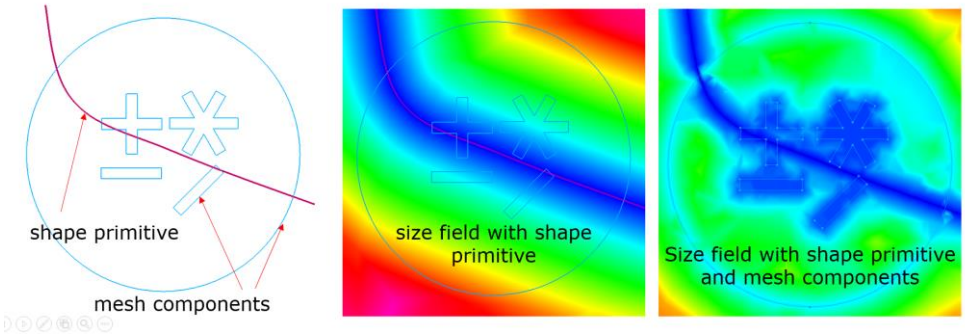


Figure 2. External Mode Voxel Mesh.

All size field cloud points obtained from one or more of the contributors above are then inserted into an alternating digital tree (ADT)[7], a general binary tree structure that allows for tree imbalance and is well-suited to storing data with regions of both high and low population. It can store anisotropic data efficiently, and can also be used for efficient determination of points in close proximity. Two points are considered duplicates of they lie within 3 times their specified size field value, with resolution being to preserve the point with the smaller size field value.

The discrete nature of the size field makes the selection of the interpolation scheme used to evaluate a point in space crucial; evaluations in close proximity need to behave with a reasonable semblance of continuity. A modified radial basis function approach (RBF) is employed[8], such that each discrete point's field evaluation is equal to its prescribed value at its prescribed position, varying linearly toward the background spacing as a function of distance from the point.

One modification to the RBF method involves a delay of growth from the cloud point data to provide a better isotropic region between the hex core and boundary meshes. Specifically, the delay distance is set equal to **transLayers *size$_i$,** where **transLayers** is a user-set minimum number of layers between refinement in the hex core mesh (controlling how quickly the hex core mesh can refine), and **size$_i$** is the prescribed i'th cloud point value. As an example, with **transLayers**=3 and **size$_i$**=0.5, point i will influence all evaluations inside a sphere of radius 1.5 with the same 0.5 value. Outside of the sphere, the influence will vary linearly with distance to the background size.

A second modification from the original RBF scheme has to do with the breadth of searches performed. Rather than have every point in space evaluated against each cloud point, querying is limited to the nearest 500 points, which is obtained efficiently from the ADT structure. Using a truncated data set has not produced any noticeable effects other than improved efficiency.

Several schemes have been implemented that control the net evaluation of the field size, given suggested values from each of the 500 closest cloud points.

1. **Minimum Distance** - the value of the cloud point closest in distance to the evaluation point
2. **Weighted by Inverse Distance** $\text{size}(p) = \Sigma\ (\text{size}_i / D(i, p)^n) / \Sigma\ (1 / D(i, p)^n)$, where D(i, p) is the distance between evaluation point p and cloud point i. In practice, n = 2 (inverse-squared law) produces the best results.
3. **Power Law Blend** – a blend of the two above, based on the ratio of the size at the Minimum Distance to the minimum distance itself.
4. **Minimum Value** – the minimum value of the all cloud points

Each of these methods has a range of applicability where it outperforms the others. For external flow type meshes, where cells sizes are expected to increase with distance from the geometry, the Minimum Distance method does best. For internal flows, where boundary cells may be in close proximity and have widely varying sizes, the Weighted by Inverse and the Power Law Blend are usually more effective.

III. Binary and Octree Structures

The ADT structure used to store size field data above was likewise implemented in early versions of the voxel meshing of this effort. Recently, however, it has been replaced with an octree method[9,10], in part because it requires slightly less memory and in part because it provides access to neighboring cells without requiring spatial queries. The latter reason significantly reduces the time required for flood filling (Section V). Details are provided below.

A typical spatial binary tree data structure is a hierarchical network of nodes, with each node containing three pointers (parent, left and right children), six real numbers for corners of the node, the number of levels below the root in each split direction, and the split direction of the node's children. A typical spatial octree data structure is similar, with each node containing the following: nine pointers (parent and eight children), six real numbers for corners of the node, and the number of levels below the root.

While a binary tree's nodes contain less data, more nodes are required to represent the same partitioning as an octree. For instance, if pointers are 64-bit and double precision real numbers are used, the binary tree node would require 76 bytes, and the octree node would require 124 bytes. However, the binary tree requires 15 nodes to represent the same partitioning as nine nodes of an octree data structure, leading to a reduction of 32 bytes (15*76 = 1140 vs. 9*124 = 1116). Further reductions in data structure size can be realized by using a block representation which stores a single pointer to a fixed length array of the children nodes. This increases the benefits of the octree node by removing seven pointers from the node, whereas it only reduces the binary tree node by a single pointer. An octree node is always a cube, so rather than storing six real numbers to represent the corners of the node, it is sufficient to only store four real numbers – a corner or mid-point and the "diameter" of the node. In the example above, that leads to a reduction of 552 bytes (15*68 = 1020 vs. 9*52 = 468). With these adjustments, the octree data structure not only uses less nodes, but each node uses less data than the binary tree node.

Another advantage of the octree data structure is the ease in encoding a location of a node within the data structure. This location code is the child index at each level of the octree as the data structure is traversed from the root node to the given node. A total of 21 levels of depth can be tracked in a 64-bit number. Further, the location code can be used to find the neighboring nodes of an octree data structure without any spatial tests.

IV. Isotropic Hex Core Meshes

When unstructured grids are used in CFD applications, four cell types are traditionally employed: tetrahedra, pyramids, prisms and hexes. In recent years, most of the cell-based CFD solvers have extended their allowable mesh types to include general polygons[11,12] containing significantly more faces than traditional elements. These extra faces presumably offer more accurate flux calculations across the cell faces. On the other hand, many vertex-based solvers require meshes formatted in the four traditional cell types[13,14] , since the vertex representation implicitly uses a dual (polygonal) representation, rendering the general form of no additional benefit. Since the meshing effort of this paper is intended to be applicable to the majority of flow solvers, cell types generated herein will be limited to tets, pyramids, prisms and hexes.

The cell types typically used in analysis reflect the region of the flowfield being resolved. In boundary layers, anisotropic prism and hex cells are often used because they can be formed into stacks with very tight spacing normal to the body. The remainder of the flow field is often filled with isotropic tetrahedra. Most tet meshers are based on complex yet nearly mature algorithms that attempt to satisfy the Delaunay criteria on the majority of the cells[15,16] . Delaunay-based tet meshes tend to appear noisy in that the orientation of adjacent cells have no discernable pattern. This, of course, could be detrimental to solution accuracy when the solution has strong gradients in the interior of the flowfield (where tets cells dominate).

A technique to avoid the geometric randomness in a mesh is to replace the isotropic tets with a Cartesian-aligned set of hex cubes, or voxels. In this way, prism and hex elements could still be used to resolve near-body phenomena, while voxel cells could be used to better resolve some problems containing gradients in the off-body regions (possibly requiring a rotation of the Cartesian frame to match the anticipated flowfield). Since the size field will vary in space, it's logical to allow for voxel refinement via repeated subdivision of voxels into octants controlled by an octree framework, which is a proven method in use now for more than two decades[17,18]. The

octree described in the previous section defines cells solely by their extent boxes, rather than by the traditional unstructured grid lists of vertices and cells containing indices into the vertex list. The octree method allows faces on adjacent voxels to be disjoint since no explicit connection is defined. As will be discussed, strict cell to cell connectivity will be enforced by converting voxels to **hex core** meshes occupying the same space as the voxels but now defined by the traditional four cell types.

A. Root Voxels

Isotropic voxel meshing has been implemented into the Pointwise® mesh generation software in three similar but distinct modes. Each of these methods begins with the construction of a root voxel system, where the root voxel dimension is small while still allowing the voxel sizes to be fairly uniform in the x,y, and z directions. The initial dimension of the root voxels is computed from the user-set extent box aspect ratio. For example, extents of length **box**[40, 80, 200] would have root voxels dimensions of **root**[1,2,5]. Extent boxes lengths not easily divisible into integer values in all three directions necessitate a scale factor be used to transform all voxel nodes into a uniform mesh size during formation, and then unscaled to the true non-uniform sizes at the completion. As another example, extents of length **box**[45, 75, 210] would still produce dimensions of root[1,2,5], but also now non-uniform scale factors of **scale**[0.889, 1.08, 0.9524].

The first of these voxel modes, implemented in 2018, creates a rectangular voxel mesh that is refined locally to the size field. This type is used for overset grid applications, serving as transition meshes to better interpolate between overlapping meshes of widely varying spacing. The other two modes introduced herein involve integration of existing surface grids and/or anisotropic (boundary layer) meshes[19,20], with a voxel mesh cut in a non-overlapping manner, and made watertight via insertion of isotropic tet cells in regions between the anisotropic and cut voxel cells.

In **external** mode, the block extents are user-prescribed, and any number of user-selected closed and open surface meshes are inserted into the block, resulting in a conformal mesh that contains voxels on the exterior and surface and/or anisotropic meshes on the block interior (see Figure 3). Meshes of this type typically are used in external flow applications, where the size or surface mesh on the outer boundaries is less important than the inner boundary spacing. The sizes of voxel elements in this mode keys off the minimum cell size Δs_{min} found in the size field. Specifically, the root voxels are dimensioned so that repeated subdivision by powers of 2 will allow the minimum voxel size to be close in size to the size field minimum. Otherwise, the discrete nature of the voxel mesh would only guarantee that the minimum voxel lies within a factor of 2 (greater or less) than the size field minimum. Note that in this mode, the size of the voxels on the outer boundaries is less important than the voxel size near the interior meshes. The algorithm used to compute root dimensions in external mode is shown below.

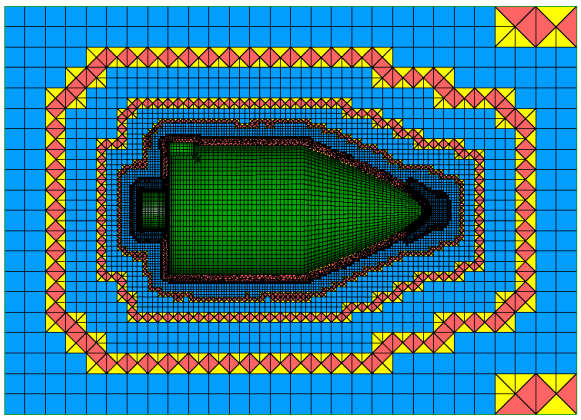


Figure 3. External Mode Voxel Mesh.

External Mode Root Voxel Computation

Define

1. Δs_{min} ≡ smallest size in size field
2. **box[n]** ≡ extent box length in n ∈ {x,y,z}
3. **M** ∈{1,3,5,7,9,11} // odd integers

Compute

1. foreach **n** ∈ **{x,y,z}** {
    a. foreach **m** ∈ **M** {

i. compute **r[m]** ≡ exponent of 2 scaling Δs_{min} to root

ii. from $\Delta s_{min} * 2^{r[m]} = box[n] / m$

b. find **k = m** satisfying $min_m (r[m] - int(r[m]))$

c. **root[n] = k** // coarse value of root dimension

d. **i[n] = int(r[k])** // integer exponent of 2

2. $j[n] = i[n] - min_n(i[n])$ // exponent of 2 to scale root dims

3. $root[n] *= 2^{j[n]}$ // final scaled root dimensions

Odd integers used above are (somewhat arbitrarily) limited to 11 in this implementation to minimize the chance of root voxel sizes from becoming smaller than the maximum values (Δs_{max}) found in the size field.

In **internal** mode, a closed outer boundary and zero or more closed and open inner boundaries are prescribed, and voxels are formed inside the outer boundary and outside the possible inner boundaries (see Figure 5). While meshes of this type usually represent internal flow geometries, they may also be used in instances when the outer boundary needs to respect a prescribed mesh or spacing, such as when two or more blocks abut one another. The sizes of voxel elements keys off the maximum cell size Δs_{max} found in the size field in this mode, because here it is important to match voxel sizes at both the outer and inner boundaries. A nominal voxel extent **box[l]** is first computed from the inner and outer surface that comprise the mesh boundaries. Since the extent box size is not prescribed, however, it is possible to shrink **box[l]** in 1 or 2 directions as shown below to ensure that all voxels have uniform lengths in x, y, and z, at the expense of exposing slightly more space between the voxels and the surface meshes.

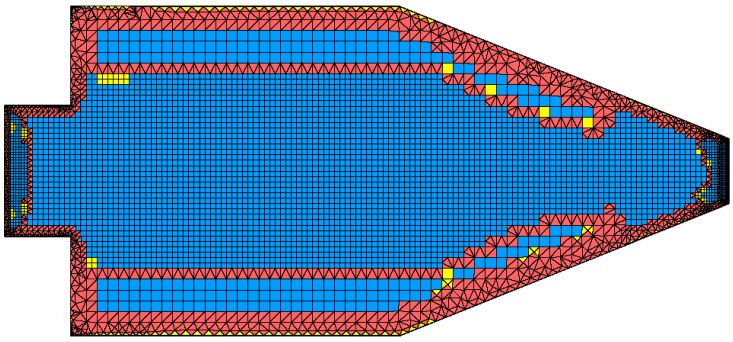


Figure 4. Internal Mode Voxel Mesh.

Internal Mode Voxel Root Computation:

Define

1. Δs_{max} ≡ largest size in size field,
2. **box[n]** ≡ extent box length in n ∈ {x,y,z}. // nominal extent box

Compute

1. $root[n] = max_n (int(box[n] / \Delta s_{max}))$ // nominal root dimensions
2. find $K = max\ k$ satisfying $4 * 2^k < max_n(root[n])$
3. $root[n] /= 2^K$ // final root dimensions
4. $leng = min(\Delta s_{max}, min_n(box[n]))$ // length of root voxel
5. $box[n] = size * root[n]$ // final extent box

B. Voxel Refinement

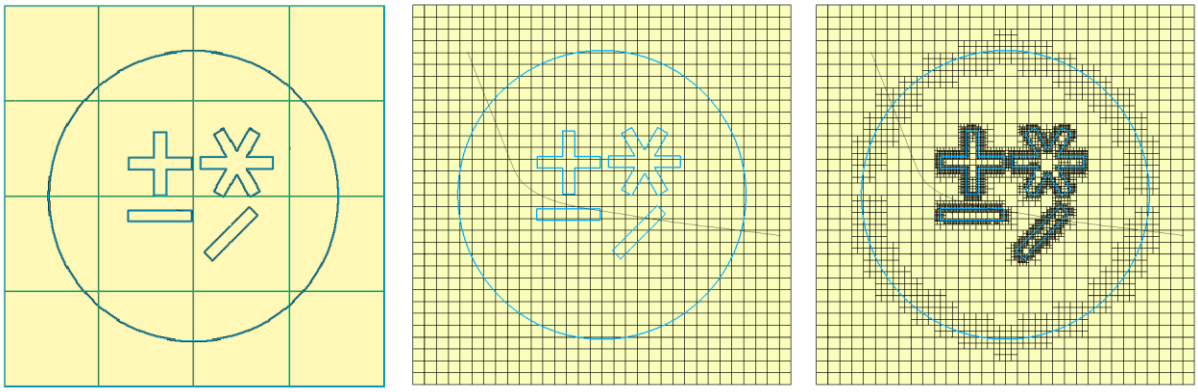
Once the root dimensions and extent box are computed, root voxels are refined recursively to match the local size field. In the outline to follow, the **target level** is equal to the number of binary divisions of the root voxel required to satisfy the size field. For each root voxel (Figure 5a):

1. Split all nodes until the leaf nodes are at the level as specified by Δs_{max}. (Figure 5b).
2. For each cloud point position and size field value, compute the target level.

a. For each leaf node:
    i. If the node contains the point, and its current level is less than the target level, split the node.
    ii. If the node was split, repeat i. for each child node (Figure 5c).
3. Enforce **transLayer** =1 to insure that adjacent voxels are refinement levels differ by no more than 1 (Figure 6a).
4. For each leaf node:
    a. Evaluate the size field at the mid-point of the node to determine the target level.
    b. If the node's current level is less than the target level, split the node.
    c. If the node was split repeat a. for each child node. (Figure 6b).
5. Enforce the user-specified **transLayer**, which specifies the number of adjacent voxels required to be at the same refinement level in the final mesh.

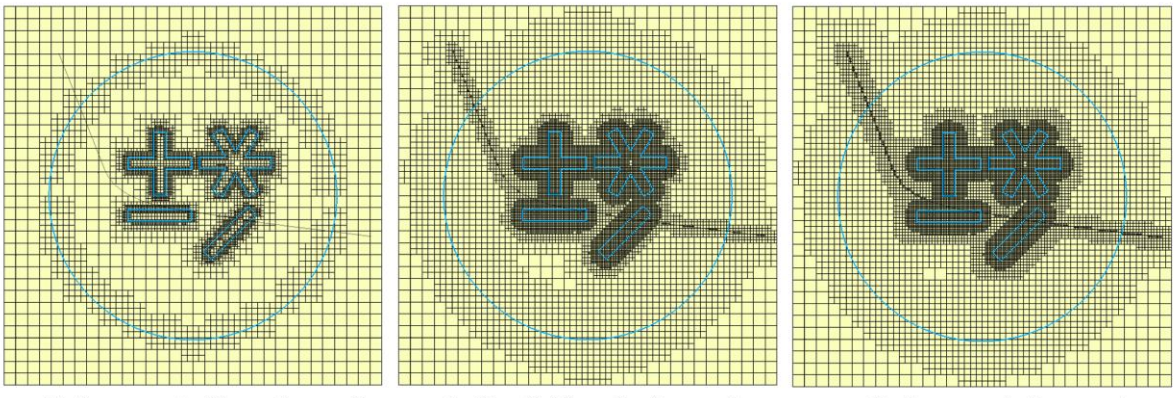
The algorithm for enforcing **transLayer** applied to each root voxel:

1. Set the target level to one less than the maximum level of the leaf nodes.
2. While the target level is greater than 0:
    a. Calculate the influence radius as **transLayer** times the size of a node at the target level.
    b. Get the mid-point of all leaf nodes whose current level is the target level plus one.
    c. Recursively split leaf nodes that are within the influence radius of the mid points until their level is the target level (Figure 6c).
    d. Subtract one from the target level.



a. Root voxels **b. Root refinement to match Δs_{max}** **c. Δs_{max} voxel refinement to match size field**

Figure 5. Initial refinement of root voxels.



a. Enforcement of transLayer=1 **b. Size field evaluation and refinement** **c. Enforcement of user-set transLayer**

Figure 6. TransLayer enforcement and further voxel refinement.

V. Voxel Classification

The mesh at this point will consist of a rectangular block of voxels that fully envelop the surface and boundary meshes, with each voxel refined to the level specified by the size field. In this section voxels will be classified either as cutting the boundary meshes (**Cut**), lying outside of the boundary meshes (**Out**), or lying inside the boundaries (**In**), the latter classification representing the only non-discarded set.

A. Cut Voxels

A voxel cell cuts a triangle if any portion of the voxel or triangle occupies the same space, either by intersection or adjacency. While each triangle must ultimately be intersection-tested against each voxel, the Cartesian shape of the voxels allows most triangle-voxel tests to be marked non-intersecting with as few as one geometric extent box comparison. Those that cannot be discerned via extent boxes alone are intersected using the efficient triangle-box overlap test of Akenine-Möller[21].

The procedure for assessing voxel **Cut** status involves inflating the triangle extent box to determine candidate voxels to test, followed by inflating the voxel extents passed to the cutting algorithm (Figure 7). First, the extent box (black) of the triangle are inflated to a width of ¾ of the max triangle edge length in all positive and negative directions (red box). Using fast hierarchical voxel geometric tests, the collection of voxels intersecting the inflated tri box is found (light gray voxels). The buffer offered by the inflated box ensures that all potential intersecting voxels are considered. Next, each voxel in the list is inflated by the maximum of the voxel edge length **Δv** and **0.75** times the max triangle edge length (medium grey voxel and blue box). The triangle is then passed to the Akenine-Möller algorithm with the inflated voxel to assess intersection. This approach enforces defined minimum distances between a given triangle and its nearest non-cut voxel. Specifically, when **Δv > ¾ maxEdge,** the distance **D** between the triangle and the nearest voxel is given by **½ Δv < D < 1½ Δv** in either the x, y or z direction**.** When **Δv < ¾*maxEdge,** the nearest distance is defined by **¾ maxEdge - ½ Δv < D < ¾ maxEdge + ½ Δv.** In both cases, the max distance is prescribed within ½ of a voxel width **Δv.**

B. In/Out Voxels

Voxels not identified as **Cut** in the section above will now be classified as either **In** or **Out**. Only a small number of voxels, specifically one per contiguous region of voxels, will need to be classified explicitly. All other voxels will be assigned **In/Out** status via a recursive flood fill of the classified voxel.

In preparation for the classification process, all non-**Cut** voxel bounding quads and triangles are loaded into a non-manifold boundary (**NMB**) representation based on the radial edge structure methodology of Ref 22. This method first builds topological Models consisting of Shells (signed collections of contiguous quads and tri faces) and Regions (finite or infinite 3D spaces bounded by 1 or more shell), and then loads the Face (tri, quad) entities into a special binary space partition tree optimized for computing ray tracing and closest point projections. Rays are then fired in random directions from the voxel centroid, looking for "clean" (not within tolerance of an edge) intersections with the faces. It may be necessary to fire multiple rays to find a definitive clean intersection. The containing Region (**In** or **Out** in this case) is determined from which side of the Face is hit by the ray, and the Shell which lies on that side of the Face.

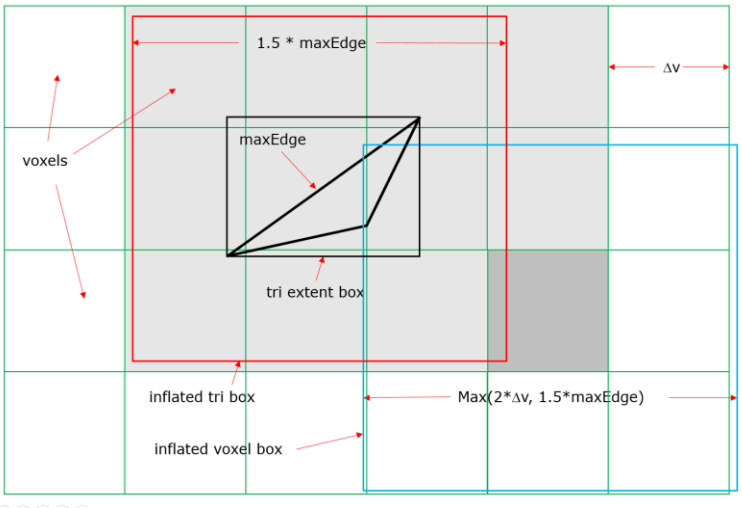


Figure 7. Inflated Triangle and Voxel Extent Boxes to Assess Cut Status

Next, all non-**Cut** voxels are assigned to group 0. From here, any one of the group 0 voxels is classified as **In** or **Out** using the procedure above. If the classification is **Out**, that voxel and all contiguous voxels via flood fill are placed in the **Out** group (Figure 8). If the classification is **In**, the group number is incremented and the voxel and flood fill neighbors are assigned to the group. This procedure is repeated iteratively by identifying a group 0 voxel, marking it as either **Out** or an incremented group number, followed by finding all adjacent voxels via flood fill, until all voxels are classified.

Voxels belonging to different group numbers will correspond to physically disjoint regions of voxels. After classification is complete, a tally is made of the number of voxels assigned to each region, and voxels belonging to groups containing less than 1% of the total number of voxels are marked **Out**. This prevents very small pockets of voxels (which have no advantage in isolation from tets) from being present in the mesh.

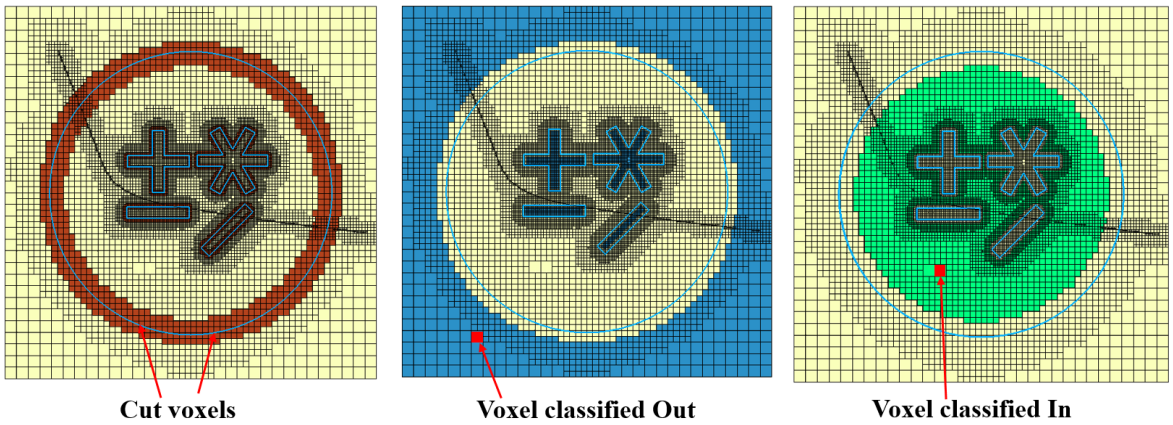


Figure 8. Classification of Cut, Out and In Voxels.

VI. Voxel Conversion to Hex Core Form

A voxel representation is converted into hex core form by replacing each voxel rectangle with either a single hex or a small set of tets and pyramids occupying the same physical space (prisms are not created from voxels). This begins by enumerating a list of xyz vertices representing the corners of all voxels marked **In**. Next, each voxel is checked for face consistency with its neighbors. If all 6 faces fully match neighboring cell faces (or represent a boundary), the voxel is exported as a hex element. For all others cases, there will be at least one voxel face whose neighbor's face is either a subset or superset of the face. Conversion of these cells to general elements could proceed simply by replacing 1 face in the voxel with 4 faces when the neighbor refined an additional layer. Conversion to tets, pyramids and hexes, however, requires more complex logic[23]. In general, only the regions of cells in the layer between voxels of differing refinement levels need be converted to tets and pyramids. As shown in Figure 9, pyramid bases are placed adjacent to hex elements on both sides, while tets are used to link the exposed pyramid triangle faces.

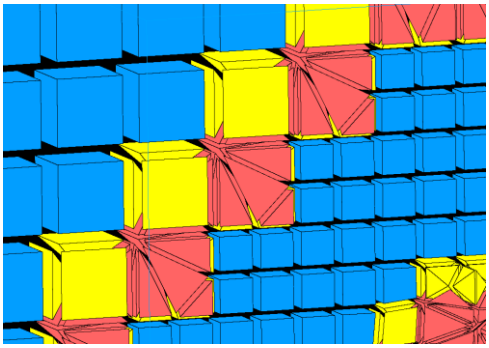


Figure 9. Transition Pyramids and Tets Connecting Levels of Hex Refinement

A situation that frequently arises after cutting voxels (and removing **Cut** and **Out** cells) is the production of **In** cells (now tets, pyramids and hexes) that connect in a non-manifold manner. All of the faces in the converted voxel cells will be used by exactly one or two cells. Considering only the faces of cells that are used by one cell (boundaries to the voxel cells), an edge on the face will be non-manifold if it is used by more than two faces. Defining a vertex fan as a collection of faces that are connected via edge boundaries, a vertex on the face will be non-manifold if there exists more than one unique vertex fan. These situations are illustrated in Figure 10. On the left, the edge is non-manifold because it is used by a total of four faces (two from each cell). On the right, the vertex is non-manifold because there exist two vertex fans at the coincident vertex – one containing three faces from the lower cell, and the other containing three faces from the upper cell.

The mere existence of non-manifold edges and vertices does not disqualify the mesh, because the isotropic mesher described later can fill regions with this type of topology. However, the division of the boundary faces into contiguous regions and the subsequent grouping of fronts into boundaries of regions (Section VII) to be filled with isotropic tets is greatly streamlined by removing non-manifold edges and vertices. Therefore cells touching other cells via non-manifold edges and vertices are removed iteratively from the mesh, with the non-manifold status updated after each iterative sweep.

It has been observed that removing these cells creates the potential for small islands of cells to be formed, in much the same way as when **In** voxels were categorized into groups. Again, the solution is to remove voxel core cells from groups comprising less than one percent of the remaining cell count.

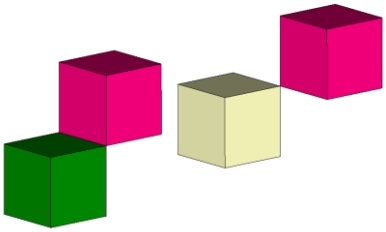


Figure 10. Cells with Non-Manifold Edges (left) and Vertices (right)

As will be described in Section VII, the final step in the hex-core meshing procedure is to fill the regions between the hex-core cells (tets, pyramids and hexes) and the surface/anisotropic meshes with isotropic tets. Therefore, quad faces from pyramids and hexes on exposed boundaries of the hex-core cells need to be replaced with diagonalized right angle triangles. This proceeds in 2 steps. First, hex cells adjacent to the hex-core boundaries are replaced with 6 pyramids occupying the same space and sharing a tip vertex at the hex centroid. Next, pyramids with quads on the exposed boundary (many created in Step 1) are split into two tets, diagonalizing the boundary quad in the process.

Unfortunately, the right-angle triangle pairs formed from quads present a difficult issue for a Delaunay isotropic tet mesher, because it results in non-simplex vertices that lie on the circumsphere of nearby triangles (not strongly Delaunay). This condition increases the burden (and inefficiency) of constrained Delauanay tetrahedralization [24]. A simple solution that has proven effective is to perturb boundary vertices by a small amount along the direction of their geometric normal to the surface in order to improve the Delaunay condition (Figure 11). The perturbation distance is determined by a random number, and is capped at ten percent of the overall voxel length for convex and concave corners, and one percent for all other boundary vertices. Further, perturbation can proceed without checking modified cell quality because the perturbation is small and the cells formed from voxels are of very high quality.

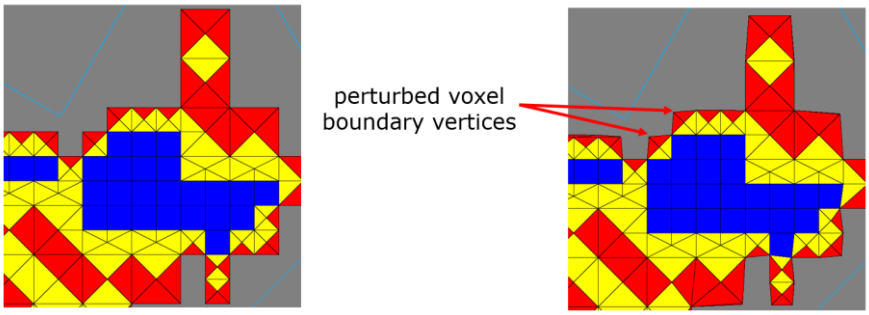


Figure 11. Perturbation of Boundary Vertices to Improve Isotropic Tet Mesh Efficiency.

VII. Front Classification and Isotropic Tet Generation

The remaining voids to fill with mesh elements are the 'zipper*' regions- those bounded by the surface meshes and the boundaries of the hex-core cells. However, there may exist several closed and open surface meshes and hex-core boundaries, which greatly complicates the proper grouping of cells representing individual mesh regions. To illustrate, consider the schematic arrangement in Figure 12, consisting of 11 boundaries (fronts) and 5 zipper regions (yellow). The large white region represents the space occupied by the hex-core mesh. It is clear from the schematic that zipper regions are bound by (f1, f3), (f2, f5, f4), (f6, f7), (f8, f9) and (f10,f11), but codifying this information in a robust way is considerably more difficult. In general, the determination of the fronts to be grouped into zipper regions follows the **In/Out** tests used to classify voxel cells, consisting of arranging all face elements into a radial-edge based NMB model. However, now the surface/anisotopic mesh fronts are included in addition to the hex-core fronts, and among the former there can exist non-manifold open faces, which can reduce efficiency for complicated cases. Nevertheless, the method incorporated is quite reliable [25].

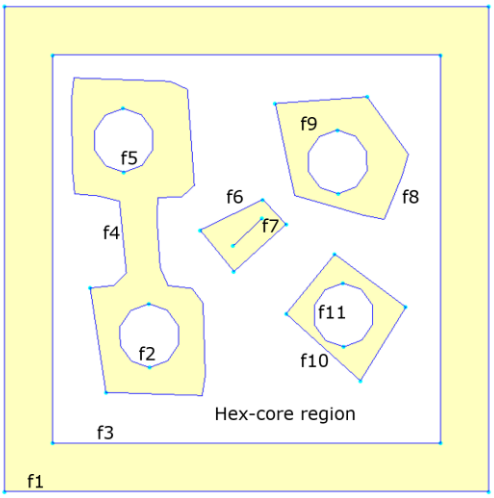


Figure 12. Schematic of Eleven Boundaries and Five Meshing Regions

Fortunately, a few simple heuristics can be implemented to identify in many cases which surfaces should be matched. First, surface meshes and hex-core boundary meshes are independently divided into manifold fronts, oriented so that normals point into the direction yet to be filled. For hex-core meshes, all fronts will be closed, but open fronts may exist on surface meshes in the event that baffles have been inserted into the mesh.

If there are no open fronts, Green-Gauss integration is applied to all of the surface elements of a given front to determine if the enclosing volume is positive or negative. Next, each front is classified as either

PosSurf – surface front with positive volume,
NegSurf – surface front with negative volume,
PosVox – hex-core front with positive volume, or
NegVox – hex-core front with negative volume.

Also set

regOuter = 1 if exactly 1 **PosSurf** and 1 **NegVox** fronts,
= 0 otherwise,
regInner = 1 if exactly 1 **NegSurf** and 1 **PosVox** fronts,
= 0 if exactly 0 **NegSurf** and 0 **PosVox** fronts,
= -1 otherwise.

Finally, if

regOuter = 1 & **regInner** = 0, there is only one meshing region bounded by the **PosSurf** and **NegVox** fronts,

* so named because they zip together anisotropic and hex-core meshes.

and if

regOuter = 1 and **regInner** = 1, there are two meshing regions; one bounded by the **PosSurf** and **NegVox** fronts, and the other bounded by the **NegSurf** and **PosVox** fronts.

One other heuristic can be applied when there exists exactly one closed front and one or more open fronts. If *all* of the fronts are directly or indirectly connected (meaning that there are no free floating interior fronts, either opened or closed), there will be a single meshing region consisting of all of the fronts.

Each group of 2 or more front surface elements that comprise a zipper region is passed into the isotropic tet mesher independently. Since these regions are generally thin compared to the anisotropic cells or hex-core cells, the mesher is run in a minimal way, inserting interior grid points only when required for surface recovery. As a final step of the entire meshing process, the isotropic tets are integrated in a fully connected way with the hex-core cells, surface boundaries and/or anisotropic cells.

VIII. Demonstration

Three representative hex core mesh examples are provided in this section to illustrate the capabilities of this technique. A notional sports car model is meshed in Figure 13. The geometry was meshed in external mode, meaning that only surface meshes on the car were required *a priori*, with outer surface meshes on the faces of the user-prescribed rectangular region provided as output. Hex core cells lie from the outside inward to a narrow region filled with isotropic tets that separates the geometry and hex core cells. A total of 2.5 million cells (1.46 M tets, 600K pyramids, and 456K hexes) were formed in a total of 34 s.

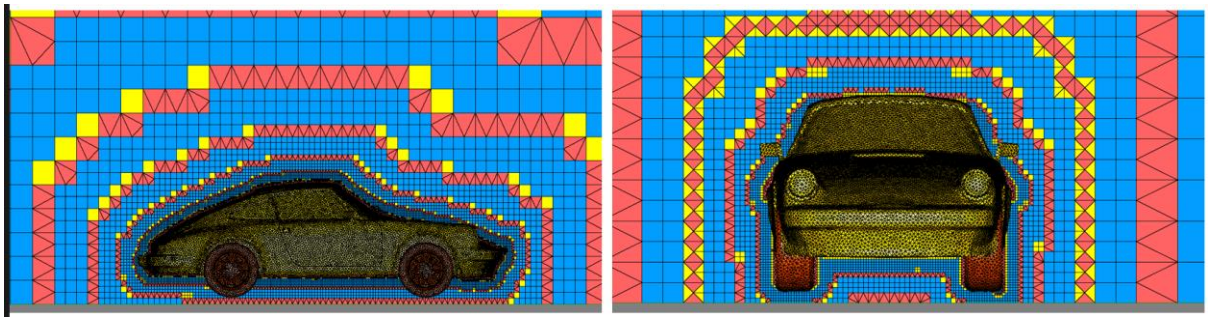


Figure 13. Hex Core on Notional Sports Car Model (External Mode)

Figure 14 depicts a hex core mesh on an aerobatic plane generated in internal mode, meaning that surface meshes, plane of symmetry and farfield meshes were required as a starting point. In this example, 15 layers of anisotropic cells were extruded from the quads and triangle surface cells using the method of Ref 19, resulting in 97K pyramids, 18K prisms and 138K hexes in approximately 18s. Anisotropic hexes (light blue) can be seen adjacent to the wing and fuselage. Next , the region between the anisotropic front surface elements, the symmetry and farfield surface meshes was run through the voxel mesher, resulting in a hex core containing a total of 2.3M cells (1.24M tets, 400k additional pyramids and 430K additional hexes), formed in 32 s.

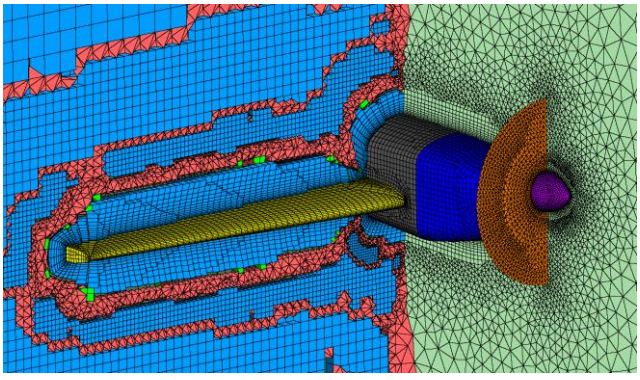


Figure 14. Hex Core and Anisotropic Core Meshes on Aerobatic Plane (Internal Mode)

The final example is a notional waverider configuration with a sizefield generated from an adaptation sensor point cloud. An inviscid mesh was created on this geometry with unstructured tets and was run to convergence in the SU2 code [13] at a supersonic Mach number. Using a sensor variable equal to the determinant of the Hessian of the Mach Number, a total of 120,000 sensor cloud points were extracted (the mesh had 1M vertices) , and the unstructured tet mesh was refined in multiple passes using the adaptation size field. The resultant grid is shown at the top of Figure 15. Next, the interior tet cells were removed from the mesh, and the voxel solver was run with the same adaptive size field. The bottom image of Figure 15 indicates strong similarities in the meshes subject to the same size field, even though they are formed from different methods and cell types.

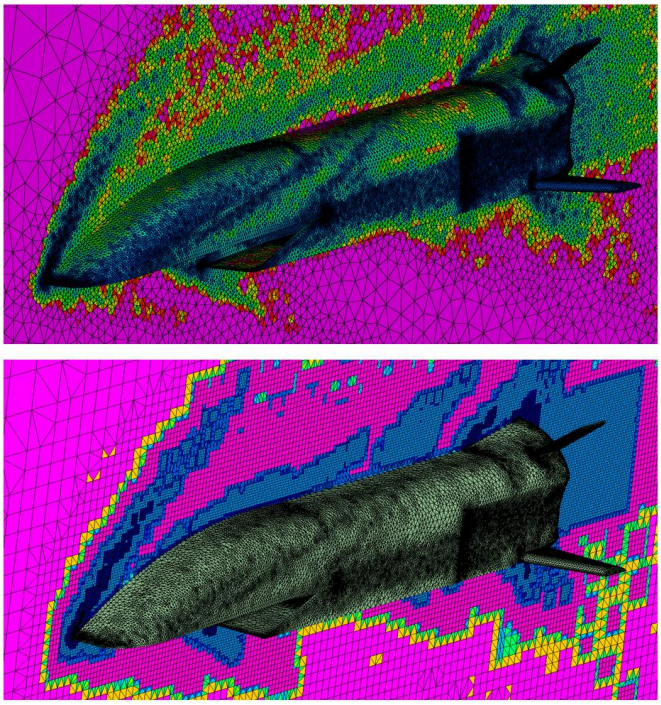


Figure 15. Notional Waverider Configuration with Tet (top) and Hex Core (bottom) Cells.

IX. Analysis and Conclusions

To validate the implementation and effectiveness of the voxel meshes, a total of 3 meshes have been generated around the Onera M6 wing. All 3 of these meshes were then run to convergence in the CRUNCH CFD solver[26,27],a vertex-centered finite volume code offered by CRAFT Tech. The flow parameters used include a Mach Number of 0.8395, temperature of 288.15K, and an angle of attack of 5 degrees. All 3 of the meshes utilize nearly the same inner core of 3.0 million boundary layer prism and pyramid cells generated using Pointwise's T-Rex anisotropic advancing layer technique[19,20]. The grids differ only in the type of mesh used to model the off-body portions of the flow field (Figure 16). The first grid, **IsoTets**, contains 2.4 million isotropic tetrahedra generated from Pointwise's modified Delaunay mesher. The second, **InternalVoxel**, contains 4.7 million hex, pyramid and tet cells formed by running the voxel solver in internal mode. Here, the outer boundary of the mesh (not shown) consists of a rectangular box defined by quad and triangular elements. This results in two transition regions filled by istropic tets – one between the outer boundary and neighboring voxel elements, and one between the anisotropic fronts and the inner voxel cells. Finally, the **ExternalVoxel** mesh contains 2.8 million cells (hex, pyramid and tet) formed by running the voxel solver in external mode, resulting in a single transition region filled by isotropic tets, located between the anisotropic fronts and the inner voxel cells. In this case voxel cells extend to the outer boundary.

Convergence histories of the average L2 norm velocity residuals are plotted in Figure 17 for these three meshes. Notice that all three converge in similar manners, though the **IsoTets** mesh has the absolute lowest residual. The two spikes in the chart are due to increasing the CFL numbers at 1000 and again at 2000 iterations. This suggests that the different mesh composition offered by the two voxel methods are as stable as the traditional tet-only isotropic mesh. The resolution of the transonic shock in the flowfield is illustrated for the three mesh types in Figure 18. The shock discontinuity is clearly more dissipative in the IsoTets mesh than in either of the

voxel meshes. This is not unexpected, however, because the axis-aligned voxels are also aligned with the principal flow direction.

The minimal analysis described above suggests the legitimacy of the concept of blending Cartesian-aligned hex-core meshes representing off-body portions of the flowfield with boundary layer meshes, connected to each other with isotropic tet zipper meshes. The logical nature of the overall voxel creation algorithm and subsequent conversion into hex-core (tet, pyramid, prism and hex) cells allows it to coded robustly for automated, general use. Two different approaches are used for external or internal flow problems. In **ExternalVoxel mode**, the minimum voxel is sized (by default) from the minimum value in the size field so that the mesh size transitions continuously between the anisotropic or surface front and the hex-core cells. In **InternalVoxel** mode, the max voxel is sized from the maximum value in the size field, since the largest surface elements usually lie on the outer boundary.

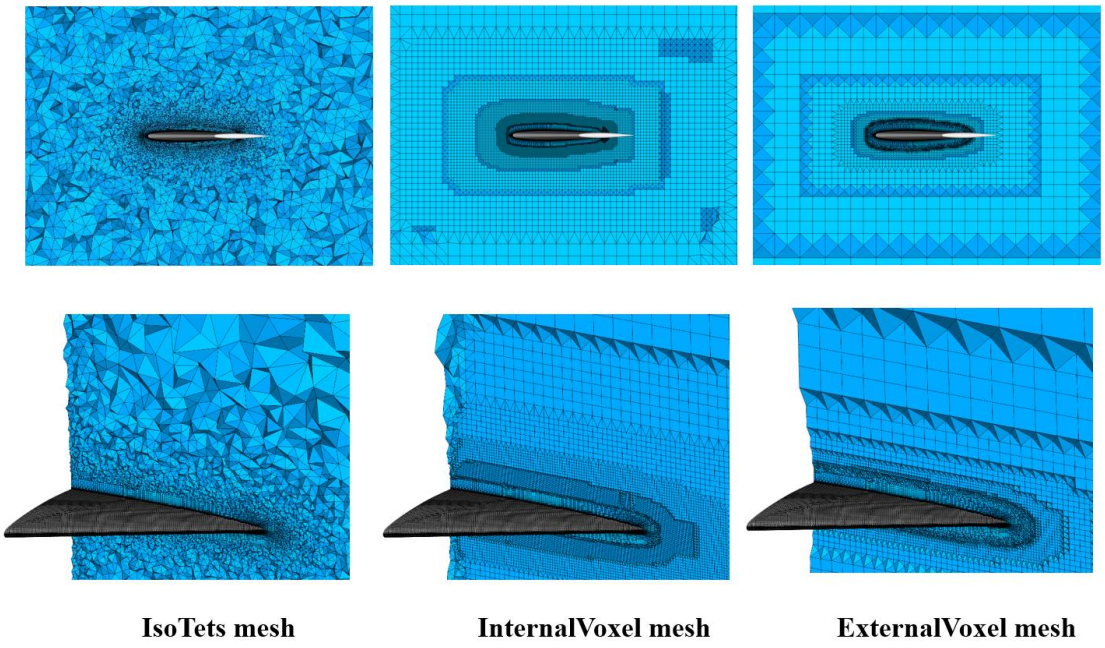


Figure 16. IsoTets (5M cells), InternalVoxel (7.6M cells) and ExternalVoxel (5.8M cells) Meshes.

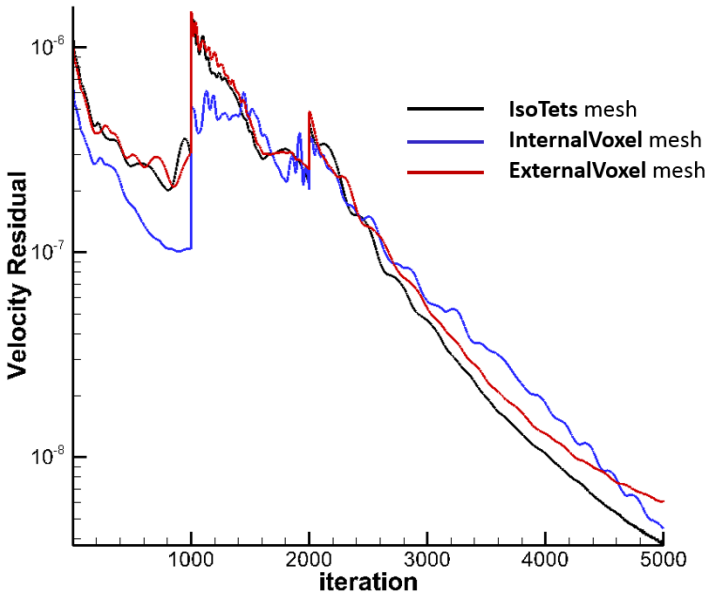


Figure 17. Residual History.

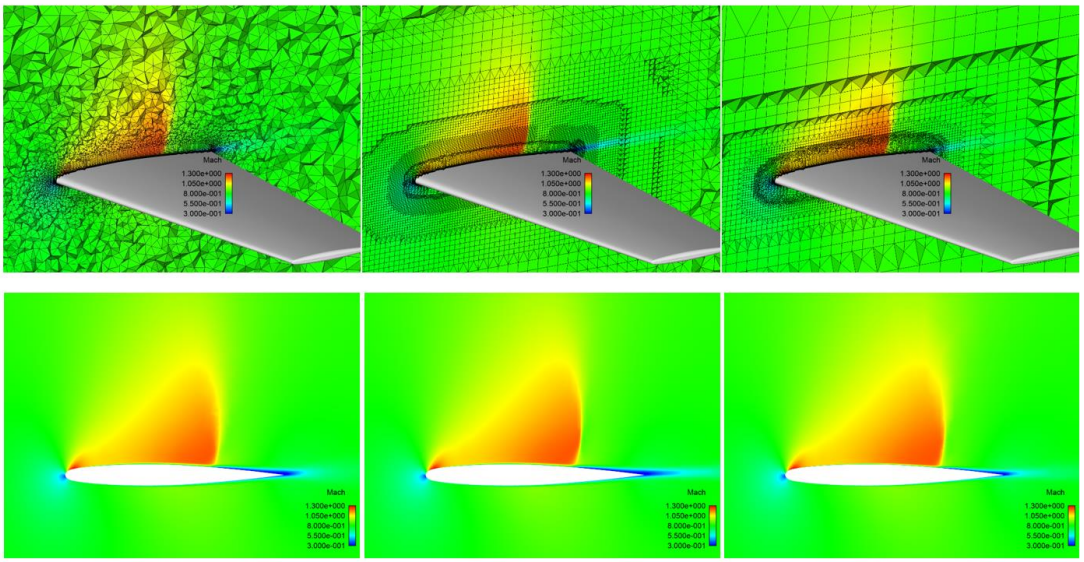


Figure 18. Transonic Shock Resolution Comparison.

Acknowledgments

The authors are indebted to J.P. Abelanet of Basis Software for detailed explanations of the point classification scheme used in non-manifold boundary modeling.

References

1. Jameson, A., Baker, T.J., and Weatherill, N.P., "Calculation of Inviscid Transonic Flow Over a Complete Aircraft," AIAA-86-0103, Reno, NV, 1986.
2. Karman, Jr., S.L., Steinbrenner, J.P., and Kisielewski, K.M., "Analysis of the F-16 Flow Field by a Block Grid Euler Approach,", AGARD CP-412, 1986.
3. Nakahashi, K., and Obayashi, S., "Viscous Flow Computations Using a Composite Grid," Proceedings of the AIAA 8th Computational Fluid Dynamics Conference, pp. 303-12, Honolulu, Hi, 1987.
4. Baker, Timothy J., "Mesh Generation: Art or Science?" Progress in Aerospace Sciences, Vol 41, pp. 29-63, Jan 2005.
5. Mavriplis, D.J., "Revisiting the Least-Squares Procedure for Gradient Reconstruction on Unstructured Meshes," AIAA 2003-3986, June 2003.
6. Davis, Z. S., Carrigan, T. J., Wyman, M and McMullen, "Reducing the Computational Cost of Viscous Mesh Adaptation," AIAA-2017-3109, Denver, CO, 2017.
7. Bonet, J. and Peraire, J., "An Alternating Digital Tree (ADT) Algorithm for Geometric Searching and Intersection Problems," *Int. J. Num. Meth. Engng*, 31:1-17, 1991.
8. Broomhead, David H. and Lowe, David, "Multivariable Functional Interpolation and Adaptive Networks," *Complex Systems*. Vol. **2**, pp. 321–355, 1988.
9. Yoder, R. and Bloniarz, P., "A Practical Algorithm for Computing Neighbors in Quadtrees, Octrees, and HyperOctrees," Proceedings of the 2006 International Conference on Modeling, Simulation & Visualization Methods, MSV 2006, Las Vegas, NV, 2006.
10. Geier, David, "Advanced Octrees 2: node representations," https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations/, 2014.
11. Chen, Goong; Xiong, Qingang; Morris, Philip J.; Paterson, Eric G.; Sergeev, Alexey Sergeev; Wang, Yi-Ching, "OpenFOAM for Computational Fluid Dynamics," *Notices of the American Mathematical Society*, vol 61, No. 4, pp. 354–363, 2014.

12. Zou, Y., Zhao, X., and Chen, Q. 2018. “Comparison of STAR-CCM+ and ANSYS Fluent for simulating indoor airflows,” *Building Simulation*, Vol. 11, No. 1. pp. 165-174, 2018.
13. Palacios, F., Colonno, M.R., Aranake, A.C., Campos, A., Copeland, S.R., Economon, T.D., Lonkar, A.K., Lukaczyk, T.W.,Taylor, T.W.R., and Alonso, J.J., "Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design", AIAA Paper 2013-0287, Grapevine, TX, 2013.
14. Pandya, M.J., Frink, N.J., Abdol-Hamid, K.S., and Chung, J.J., “Recent Enhancements to USM3D Unstructured Flow Solver for Unsteady Flows,” AIAA-2004-5201, Providence, RI, 2004.
15. Baker, Timothy J., “Generation of Tetrahedral Meshes Around Complete Aircraft,” Numerical *Grid Generation in Computational Fluid Mechanics ’88*, Pineridge Press, 1988.
16. George, P.L., “Tet Meshing: Construction, Optimization, And Adaptation,” Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, 1999.
17. Aftosmis, M.J., Berger, M.J., and Melton, J.E., “Robust and Efficient Cartesian Mesh Gerneration for Component-Based Geometry,” AIAA-97-0196, Reno, NV, 1997.
18. Karman Jr., S.L., “SPLITFLOW: A 3D Unstructured Cartesian/Prismatic Grid CFD Code for Complex Geometries,” AIAA-95-0331, Reno, NV, 1993.
19. Steinbrenner, John P. and Abelanet, J.P., “Anisotropic Tetrahedral Meshing Based on Surface Deformation Techniques,” AIAA-2006-0554, AIAA 45th Aerospace Sciences Meeting, Reno, NV, 2006.
20. Steinbrenner, John P., “Construction of Prism and Hex Layers from Anisotropic Tetrahedra,” AIAA-2015-2296, Grapevine, TX, 2015.
21. Akenine-Möller, T., “Fast 3D Triangle-Box Overlap Testing,” *Journal of Graphics Tools*, Vol 6, No. 1, pp. 29-33, 2001.
22. Weiler, Kevin J., "The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Modeling", in: *Geometric Modeling for CAD Applications* (First IFIP WG5.2 Working Conference Rensselaerville, N.Y., 12-14 May 1986), M. Wozny, H. McLaughlin, and J. Encarnacao, (eds.), North-Holland, pp. 3-36, 1988.
23. Karman, Jr., Steve L. “Hierarchical Unstructured Mesh Generation,” AIAA-2004-0613, Reno, NV, 2004.
24. Shewchuck, J.R., “General-Dimensional Constrained Delaunay and Constrained Regular Triangulations,” I: Combinatorial Properties,” *Discrete & Computational Geometry*, vol 39 no.1-3, pp. 580-637, March 2008.
25. Weiler, Kevin J., "Non-Manifold Geometric Boundary Modeling", SIGGRAPH 1987 and 1989 Advanced Solid Modeling Tutorial Notes, Anaheim CA, Boston MA, and Dallas TX, August 1987, August 1989, and August 1990.
26. Hosangadi, A., Lee, R.A., Cavallo, P.A., Sinha, N., and York, B.J., “Hybrid, Viscous, Unstructured Mesh Solver for Propulsive Applications,” AIAA Paper 98-3153, July 1998.
27. Hosangadi, A., Lee, R.A., York, B.J., Sinha, N., and Dash, S.M., "Upwind Unstructured Scheme for Three Dimensional Combusting Flows," Journal of Propulsion and Power, Vol. 12, No. 3, 1996, pp. 494-503.