Filling the Void: Interpolating in Cartesian Cut Cells

A. Hendriks*
Massachusetts Institute of Technology,
Cambridge, MA 02139

R. Haimes†
Massachusetts Institute of Technology,
Cambridge, MA 02139

M. J. Aftosmis‡
NASA Ames Research Center,
Moffett Field, CA 94035

Abstract

The majority of scientific visualization algorithms are finite-element based. For Cartesian meshes, the definition of the boundary conditions around the body leads to inaccurate visualization data due to the nature of the grid (it is not compatible with finite-element based algorithms). In this paper, we present an algorithm to generate an intermediate, 3D, body-fitted mesh that links the Cartesian mesh to the body, and allows accurate visualization data near the body. The 3D grid consists of tetrahedral cells, and it matches to all of the nodes (including the hanging nodes) of the Cartesian grid. The only nodes created on this body are surface nodes.

1. Introduction

Cartesian and body-fitted meshes are the two forms of grid generation used for the simulation of many 3D physical systems. The orientation of the cells in a Cartesian mesh is independent of the body of interest. All of the faces of the cells are positioned in one of the three Cartesian coordinates, x, y, and z. Establishing a Cartesian mesh is therefore remarkably simple. Complications occur however where the cells are intersected by the body geometry, requiring a special treatment to deal with the boundary conditions. The end result is a quick and automatic method to mesh the domain combined with a complex set of routines that describe its boundary conditions. Body-fitted grids are more widely used. The key feature of this type of grid is that the bounds of the mesh conform to the shape of the body. While this makes the definition of the boundary conditions simple, the actual task of creating the grid becomes complex, potentially labor intensive and time consuming.[1-6] Ultimately, a hybrid grid of both these two methods would be ideal, which is what is presented in this paper.

The format of this paper is as follows. In section 2 there is an overview of the problems that exist with Cartesian meshes and why it is difficult to visualize their results. Section 3 will present the concepts developed and applied to achieve the hybrid grid, along with a more detailed presentation of the 2D and 3D algorithms. Finally, in section 4, we present some examples of the application of this hybrid grid, along with the current limitations of the code.

2. Related Work

Most scientific visualization algorithms are finite-element based (and assume a linear/bi-linear/tri-linear interpolant). Put another way, these tools require the interpolation of scalar and vector fields from within the volume elements that support the 3D domain of interest. The technique that produces geometric cuts and iso-surfaces employs a lookup table to generate the resultant surface from the volume element (or

*Graduate Student.
†Principal Research Engineer, AIAA Member.
‡Research Scientist, Senior AIAA Member.

cell). The index to this table is based upon whether each node that supports the cell is above the cut value. For hexahedra the table length is 256 in length (2^8 where 8 is the number of nodes). Implicit in this algorithm is that there is only one intersection possible along an element edge and its placement is linearly based on the scalar values of the nodes. Streamlining and particle tracing use one of a number of possible integration schemes, all of which require the velocity returned at a specified location within the domain. Almost never is this location exactly at a node that supports the mesh. Interpolation is again required.

It is clear that applying the normal visualization algorithms to the results of solvers that are Cartesian based will produce questionable results for the cut-cells. This is problematic because the data in regions close to a body is of great importance to the engineers and designers that are analyzing the results from these physical systems.

Cartesian meshes are trivial to create; yet the definition of the boundary conditions around the geometry of the body being meshed may be quite complicated. To demonstrate this problem, examine the 2D cell in Figure 1. It is an element that has been split in two by the discretization line of a body

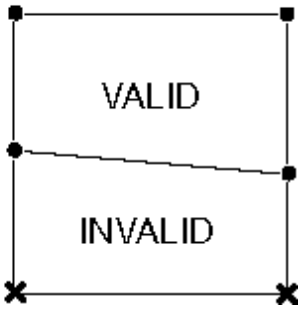


Figure 1. Surface cell for a Cartesian mesh

The top four vertices appear inside the computational domain (denoted by circles) and the bottom two are inside the body (denoted by crosses). Ordinarily, the flow quantities of any point could be determined anywhere in the cell using an interpolation scheme and the four vertices of this 2D cell. In the case of the cell shown in Figure 1, however, this becomes a problem. The cells inside the body cannot have any valid data that can be used to determine actual flow quantities, since they are not part of the solution. Take for example the no-slip condition that would be applied along the surface of the wall. It would be impossible to develop an interpolation scheme that would produce zero velocity all along the edge within the cell using only the two valid vertices at the top corners. Furthermore, one may be able to generate an interpolant for this simple example by using the invalid nodes, but this also becomes impossible as soon as the cut becomes more complex (containing more line segments).

The interpolation problem is clearly depicted in Shultz, et. al. [7] in their figure 3. Here streamlines pass through the car body, a non-physical result. The authors attempt to remedy the problem by stopping the integration when the path intersects the body. But as far as the solver is concerned, fluid does not enter or leave the car hood. This indicates that the data handling is incorrect, not the solution.

For the sake of argument, suppose it is possible to come up with some form of linear interpolation for the cut cell based upon the cell vertices that are valid and those that support the body discretization. This could produce a non-simple and possibly concave element, which would prohibit the use of the lookup table for geometric cuts and iso-surfaces. The size of the table is 2^n and it would not be out of the question to get more than a 20-node element in 3D. How is the table generated and stored?

One could imagine using a higher order interpolant that could be developed to provide better definition of the field quantities within the cell cut by the body. Accompanying this solution however are several complications in actually constructing the function. But now we could not use the lookup table because there can be multiple crossings along an edge.

Another complication to the visualization of results from Cartesian systems is that there may be a form of hierarchical embedding used. This produces the potential for *hanging nodes*. These vertices split the edges of larger neighboring elements. Interpolating near these nodes in the large element becomes a

problem. The resultant values will not be contiguous in the larger element near the *hanging node*.

3. Concepts

3.1 Visualization for Cartesian Systems

The goal of the work described in this paper is to develop techniques so those visualization algorithms can be used on the results from Cartesian-based simulations. This is important so that the solver's results can be represented back to the investigator with correct imagery.

The method proposed here is counter-intuitive to the Cartesian methodology since it generates a body-fitted mesh. In a sense we are performing grid generation with rather specific requirements:

- Fast

The ideal situation would be to have a technique that could generate the visualization mesh *on the fly*. In this way no additional memory would be required to hold the resultant body-fit mesh. This is admittedly a lofty goal.

- Robust

The algorithm must be deterministic (require no user intervention) and always function even when applied to the most complex of geometries.

- Addition of nodes

If any additional nodes need to be inserted, they must be on the body discretization, so that an interpolated value can be constructed.

It is well know that any 2D region defined via an outer collection of ordered line segments (and optionally any number of inner loops) can be filled with triangles. There are a number of algorithms in Computational Geometry that will perform this task very rapidly and robustly with no additional nodes required. It is also well known that this can not be done in 3D. Many circumstances are present [8] which prevent the filling of a husk of nodes defined by the closed triangular discretization of surfaces with tetrahedra. A number of unstructured grid generators exist that can routinely fill arbitrary volumes with tetrahedra and they are able to overcome this volume fill problem by inserting nodes into the volume of interest. This is clearly something that can not be done here – what would be the interpolated value for the node?

The technique used to solve this problem is one that always maintains a properly filled volume by cutting existing tetrahedron into tetrahedra. A cube (hexahedron) is the starting point. This element is simple and convex and can always be broken up into either 5 or 6 tetrahedra. The body discretization is imprinted into the cube and the inside (or outside) is then removed, leaving a body-fit tetrahedral mesh bounded by the box. To best understand this algorithm, it will first be described in 2D and then in 3D.

3.2 2D Algorithm

Presented here are the steps followed by the 2D algorithm. This algorithm was developed to help understand some of the problems that may occur in 3D. Each square cell is separated into the appropriate triangles, depending on which hanging nodes are present (Figure 2a). All of the triangles have orientation flags which indicate which side of the discretization line they are on.

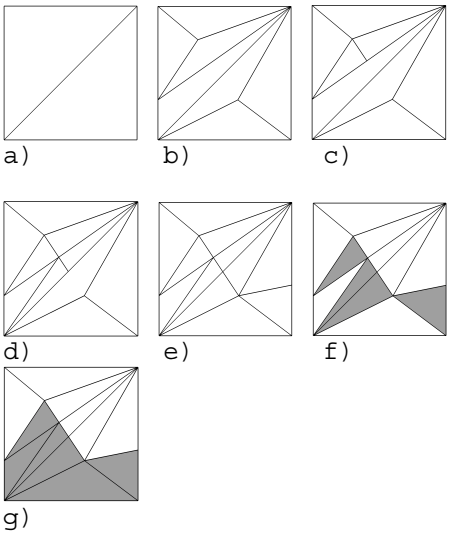


Figure 2. Example of 2D triangulation of one cell.

1. Insert all the discretization nodes in the cell, and split the triangles they are inside of into 3 smaller triangles (Figure 2b).

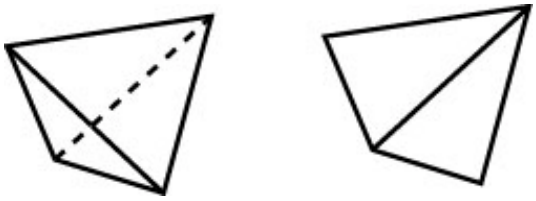
2. Connect starting discretization node to the end node with a straight line.

2.1. If the straight line cuts the edge of a triangle before reaching the end node, insert a new node at the intersection and cut the triangles accordingly (Figure 2c, 2d). Continue until the end node is reached.
2.2. Else nodes are connected; proceed to the next set of adjacent nodes.
2.3. Continue until all of the discretization nodes have been connected by straight lines and the discretization line leaves the cell (Figure 2e).

3. Move along the boundary and flag the triangles that are inside the body (Figure 2f).

4. Sweep through the remaining unmarked triangles and flag those that are inside the body (Figure 2g).

Now that the mesh has been generated and marked, it can be modified with the intention of easing the task for solving. For any grid, a simple swap algorithm [9,10] is employed to try and improve the aspect ratios of the triangles. The swap algorithm looks at the angles between edges on each triangle in their original configuration. It then compares these angles to the case where the opposite nodes of the triangle set are connected (dashed line in figure 3a). The configuration that is selected is the one that lowers the maximum angle (or increases the minimum angle). Only edges that appear inside both triangles were considered for swapping. It should also be noted that the swap operation was only performed for triangles inside the computational domain, and hence none of the discretization edges were swapped.



a) Prior to edge swap b) After edge swap

Figure 3. Edge swapping.

3.3 3D Algorithm

An algorithm was developed to handle 3D geometry based on the 2D discussion above. Additional features are added however, due to the increased difficulty in 3D. This difficulty is due to the inability to track the discretization in a single direction because each triangle has 2 directions that one could follow. It was decided that it would be cleaner (and more expedient when providing the mesh for interpolation) to color the nodes that will finally support the tetrahedral mesh. The 4 possibilities are:

- Cell vertices

These are the nodes that support the cut cell. Only those that are actually part of the volume of interest are used. This also includes any hanging nodes.

- Discretization vertices

These are the points that make up the body discretization. Collections of 3 of these nodes form the triangular tessellation that makes up the body. It is assumed that this complete tessellation is closed and *holds water*. Also, all triangles that make up the surface must have the same orientation so that the normals either point in a direction that is into or out of the body.

- Edge nodes

The 3D algorithm constructs these nodes. They are made up of the two node indices, each of which must be a discretization node with an associated weight.

- Interior nodes

These algorithm nodes are generated in the interior of body discretization triangles. The index to the triangle is stored as well as the weights (to 2 of the 3 nodes) so the linear interpolant in the triangle can be applied.

It was determined that keeping a list of neighboring tetrahedra is important from a performance standpoint, since techniques can be applied that use this information to avoid volume searches. Also, an orientation/cut flag is made available for each tetrahedron.

To ensure that each function performed on the coupled set of tetrahedra maintains a correct result, any cut operations were cast as a set of the following 3 primitives:

1. Tetrahedron insertion

A node is inserted into the interior of a tetrahedron. The result is that the parent is split into 4 tetrahedra (3 additional).

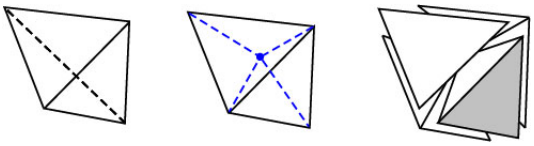


Figure 4. Node inserted inside a tetrahedron.

2. Face insertion

A node is inserted that is co-planar and coincident with a triangular face of a tetrahedron. This splits the triangle into 3 new triangles. 3 new tetrahedra take the place of the parent tetrahedron. If the face is interior (not exposed to the outside of the cube) then the neighboring cell must also be split. The end result is that 2 tetrahedra becomes 6 (the addition of 4).

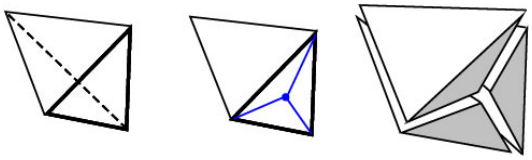


Figure 5. Node inserted on a tetrahedron face.

3. Edge insertion

A node is created along the edge between 2 nodes that support the volume elements (not to be confused with edges of the discretization). A tetrahedron that touches the edge is split in 2. Unfortunately it is not possible to specify the exact count because any number of tetrahedra can come together at an edge. All tetrahedra that touch an edge can be found by locating a single tetrahedron that contains the edge and successively examining the appropriate neighbors of each of these tetrahedra.

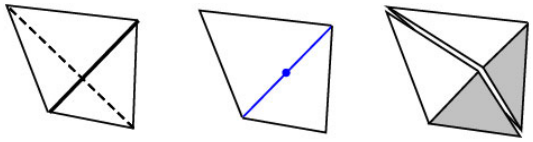


Figure 6. Node inserted along a tetrahedron edge.

For all primitives, the neighboring information is updated during the operation, so the result is always valid.

The 3D algorithm consists of the following phases:

- Generate the Box

The lower-left and upper-right set of coordinates are used to generate the 8 nodes that support the cube. This box is then subdivided into 6 tetrahedra. While the hexahedron cell could have been cut into 5 tetrahedra as well, we have chosen cutting it into 6 so that the directions of diagonals on opposite faces match. Though this is only important if one needs to patch together neighboring cubes.

- Insert the hanging nodes

Any hanging nodes are included in the volume by using Operation 2 or 3.

- Compute the intersection of the box and the tessellation triangles

The bounding box for each triangle from the discretization is compared with the coordinates of the cube. If there is any overlap, the triangle is considered for the following phases. Note: all nodes (for the triangle) can be outside the box but the triangle can still cut through.

- Insert the discretization vertices

These nodes may be inserted by any of the 3 primitive operations depending on the location of the vertex with respect to the current suite of tetrahedra. A special case is when the discretization vertex coincides with the box corner. Here the box node is overwritten with the vertex marker. It is assumed that the vertices that make up the body will not coincide.

- Scribe the tessellation edges

This is similar to the 2D cutting. The tessellation edges are inscribed in the volume by insuring that tetrahedral edges coincide. This is done by examining each tessellation edge and

determining if any node is outside (and the edge intersects the node). If either is outside, then new edge nodes get inserted by using Operation 2 or 3. Again, the special case exists where this node matches with a box vertex. If so, the box node is overwritten with the edge node.

Starting from the first vertex node (or edge node) a ray is cast toward the final node. If the ray intersects the tetrahedron's face, Operation 2 is performed, if it happens to intersect an edge along the opposite face, then Operation 3 is used. This ray casting function continues from the new inserted node until a node in a tetrahedron (that contains the new node) also has the final node.

- Cut the tessellation triangles

This process slices the volume so that faces of the tetrahedra match faces of the body tessellation. There is an outer loop for triangles that are included for this cube. Each tetrahedron is examined. Nodes that contain tessellation indices for that triangle (either discretization or edge) or the triangle index itself are marked. If there are 3 nodes marked, then this tetrahedron is complete.

The equation of the plane (that is supported by the triangle) is constructed. If the plane is found to intersect the tetrahedron (and the intersection point(s) are within the triangle) then the tetrahedron is cut using Operation 3. If there are no valid intersections then for this triangle, the tetrahedron is finished.

- Mark the orientation of tetrahedra that touch the tessellation

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

- Flood the orientation

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

- Cleave the inside away from the outside

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

- Remove possible interior and edge nodes

To reduce the count of inserted nodes (and resulting tetrahedra) all interior nodes and edge nodes are examined. Any nodes that are completely contained within tetrahedra that have one face exposed and the opposite node a match, can be removed. The node removal involves producing an outer loop that must be re-triangulated and tetrahedra extruded to that opposite node.

- Face swap

A face swapping operation (as in the 2D algorithm) is used on the desired volume to produce a better interpolant (similar to Delaunay triangulation). The procedure is first done (in 2D) on the exposed box faces. This will insure that neighboring boxes will match at their internal faces. Then the 3D analogue is done for all interior faces. [10,11]

4. Discussion

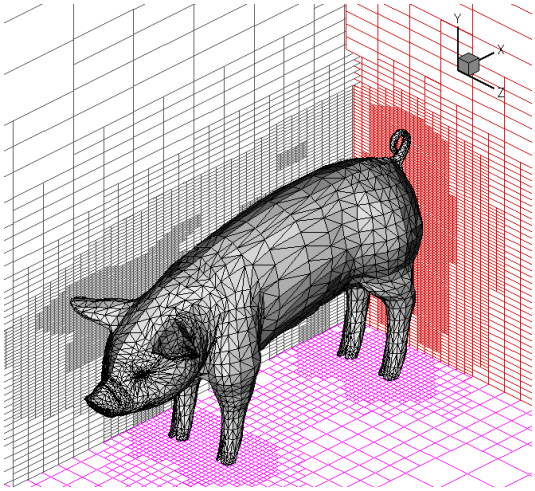


Figure 7. Makin' bacon. A Cartesian pig.

4.1 3D Examples

Presented here is a visual example of the 3D algorithm applied to the Cartesian pig (Figure 7). For the entire domain around the pig, there are 21,314 cut cells. The total number of tetrahedra generated is large. This result should not be a surprise; each primitive operation produces many new cells. For all the cut-cells around the domain, the average number of tetrahedra created per cell is 76 (both inside and

out), with a maximum of 15,104 for the cell intersecting the most geometry. The total number of tetrahedra generated outside the body was 840,333, giving an average of 39.4 per cut-cell.

In Figure 8, one cell (not directly from the mesh of the pig) is shown intersecting the body. Note that it encapsulates more geometry than usually found in a normal Cartesian cut-cell. It contains 16 vertices, 98 triangle edges and 57 discretization triangles.

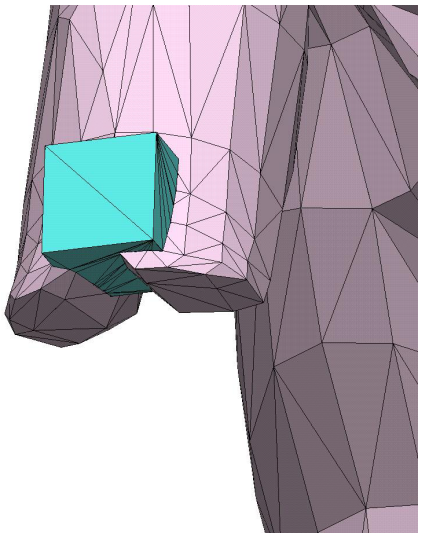


Figure 8. Pig's feet. Intersection of a cell with the pig body.

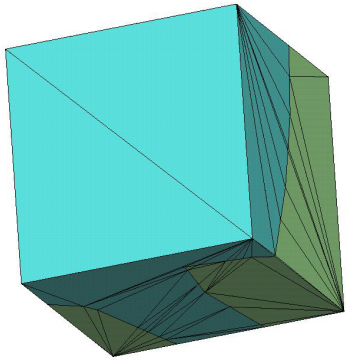


Figure 9. Contrast between the tetrahedra inside and outside the body. There were 451 new edge nodes and 215 new interior nodes created for this cell.

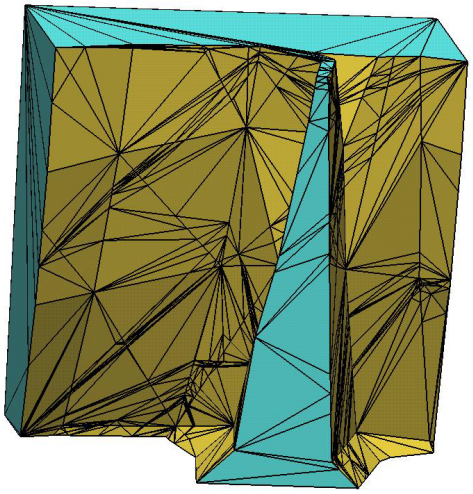


Figure 10. View of the surface of the body before node removal (cells inside the body removed).

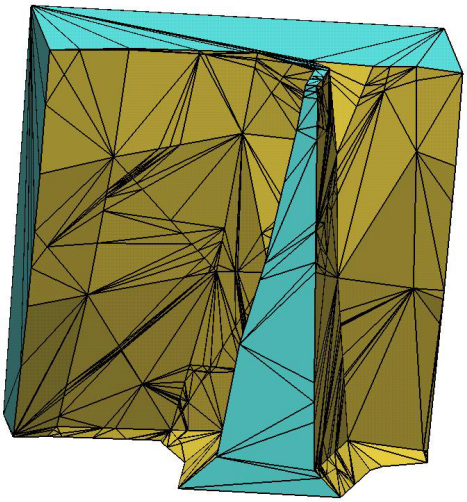


Figure 11. View of the surface of the body after node removal.

A more detailed view of this cell in Figure 9 shows the numerous tetrahedra edges that connect to vertices created along the discretization surface. As can be seen in the figure, the face that does not intersect with the body has only two triangles on it. Its nodes match those of any hexahedron cell that neighbor it. The hexahedron cell in this example has 8 nodes and hence no *hanging nodes*. If it did, the number of faces would change correspondingly so that the nodes always match. This cut-cell has a relatively large number of tetrahedra – 3026

(1402 inside the body and 1624 in the computational domain).

Removing the tetrahedra that are inside the body, we are able to look at those that will be used for visualization (Figure 10). This figure clearly shows the complex nature of the surface discretization. Keep in mind that all of the nodes created using this algorithm are on this surface.

Finally, in Figure 11, we see the same view as in Figure 10, except with a simplified geometry due to the removal of interior nodes and edge nodes. For this cell in particular, 35% of the interior nodes were removed and 32% of the edge nodes were removed. As a result of these node removals, 442 tetrahedra (27%) in the computational domain were removed.

Further analysis of all of the cut-cells that surround the pig shows that 7% of all the interior nodes and 26% of all the edge nodes can be removed. Figure 12 shows the distribution of all these nodes as a function of the number of nodes per cell.

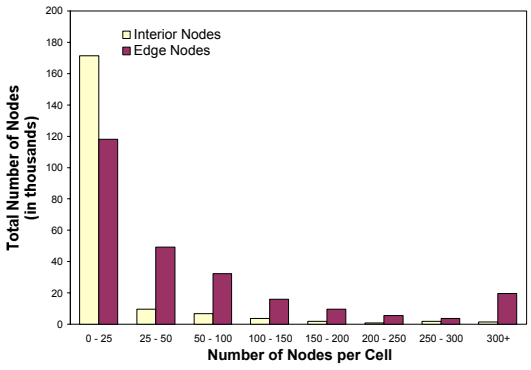


Figure 12. Distribution of all interior and edge nodes versus the number of nodes per cell.

These percentages are misleading because the majority of the nodes are found in cells which have very small (less than 25) numbers of nodes candidate for removal (see Figure 13). Cells containing such low numbers of nodes are generally not very complex and do not need node removal as much as much as cells with a higher degree of complexity. As the number of nodes per cell increases, so does the percentage of nodes actually removable.

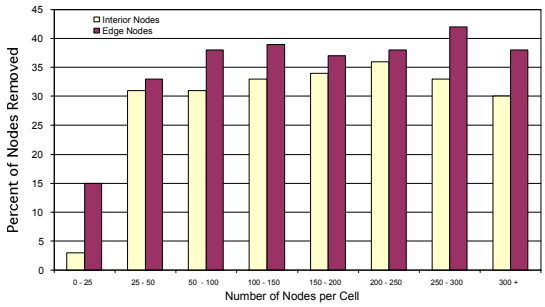


Figure 13 Distribution of removable nodes versus the number of nodes per cell.

Timings were conducted on all cut-cells for the example seen in Figure 7. This is an important element since one of the original motivations behind this idea was to be able to generate grids *on the fly*. Using a 250 MHz R10000 Octane, all of the 21,314 cut-cells that intersect with the pig body were filled with tetrahedra in 58 seconds. It should be noted that the 733 cut-cells that intersect the most geometry took 24 seconds. Although this is not a bad result, the timings are not good enough to consider not storing the results in a pre-processing step.

4.2 Limitations

What became apparent in 3D, more so than 2D, is that there are large numbers of tetrahedra being created from this algorithm. In some cases where the cut-cell intersects a great deal of body geometry, the number of tetrahedra can exceed 10,000. A good tetrahedral mesh should have between 5 and 6 tetrahedra per node [12] and it is not unheard of to see examples of 2 to 3 tetrahedra per node in our test cases. This does not bode well for bodies that could potentially be surrounded by several hundred thousand cells. However, another observation from the example cases was that many of the tetrahedra generated all shared a similar face and an opposing node. When this is the case, all of the interior nodes and edge nodes that tie the tetrahedra together can be removed and a much simpler configuration following the perimeter of this patch can be developed. It would appear that about 35% of them can be removed for the example used in this paper. This simplification can be repeated even further, since these modified tetrahedra may share to another node with other volumes.

5. Conclusion

The importance of this work is two-fold. First, it involves the development of a new grid generation scheme that is fast, robust, yet provides accurate boundary representations. This has the potential to simplify the general grid generation process significantly. The algorithms presented here can be applied to any type of Cartesian mesh. There are, however, still a few issues that must be addressed. Currently, aspect ratios of the cells have not been given any consideration. Sliver elements may appear. This is not a problem for the use in visualization because these cells will have little impact (the chances of being in the cell are small). But if the mesh is to be used by solvers directly then sliver elements can cause problems. It also remains to be seen whether all of the box interfaces will match up. The 2D face swapping needs to be fully tested to determine if this problem is resolved.

Secondly, and the thrust of this paper, is the ability to apply classical, finite-element based, visualization techniques to Cartesian meshes. Up until now, visualization data along the boundaries of bodies that employed these meshes were not very precise and produced inaccurate imagery.

Acknowledgements

This work was sponsored by Sandia National Laboratories contract #BE-8246 with Timothy J. Bartel as technical monitor.

References

[1] Melton, J.E., Berger, M.J., Aftosmis, M.J., Wong, D.M., "Development and Application of a 3D Cartesian Grid Euler Method," Proc. from NASA Wkshp on Surf. Modeling, Grid Gen., and Related Issues, NASA Lewis Research Center, May 9-11, 1995.

[2] Melton, J.E., Berger, M.J., Aftosmis, M.A, and Wong, M.J., "3D Applications of a Cartesian Grid Euler Method," *AIAA Paper 95-0853*, January 1995.

[3] Aftosmis, M.A., Melton, J.E., and Berger, M.J., "Adaption and Surface Modeling for Cartesian Mesh Methods," *AIAA Paper 95-1725-CP*, 1995.

[4] Coirier, W.J., and Powell, K.G., "An Accuracy Assessment of Cartesian-Mesh Approaches for the Euler Equations," *AIAA Paper 93-3335-CP*, June 1993.

[5] Berger, M.J., and Melton, J.E., "An Accuracy Test of a Cartesian Grid Method for Steady Flow in Complex Geometries," Proc. of the 5th International Conf. Hyp. Prob., Sonybrook, NY, 1994.

[6] Thompson, J.F., "A Reflection on Grid Generation in the 90s: Trends, Needs, and Influences," 5th International Conference on Grid Generation in CFS, 1996.

[7] Schulz, M., Reck, F., Barthelheimer, W., and Ertl, T., "Interactive Visualization of Fluid Dynamic Simulations in Locally Refined Cartesian Grids," *Visualization '99*, pp. 413-416. IEEE Computer Society, 1999.

[8] "Application Challenges to Computational Geometry," *The Computational Geometry Impact Task Force Report, Technical Report TR-521-96*, Princeton University, April 1996.
http://www.cs.princeton.edu/~chazelle

[9] Webster, B.E., Shephard, M.S., Rusak, Z., and Flaherty, J.E., "Automated Adaptive Time-Discontinuous Finite Element Method for Unsteady Compressible Airfoil Aerodynamics," *AIAA Journal, Vol. 32*, pp 748-757, April 1994.

[10] Lawson, C.L., "Properties of n-Dimensional Triangulations," *Computer Aided Geometric Design, Vol. 3*, pp 231-246, 1986.

[11] Freitag, L.A., and Ollivier-Gooch, C.F., "Tetrahedral Mesh Improvement Using Swapping and Smoothing," *International Journal for Numerical Methods in Engineering, Vol. 40*, pp 3979-4002, 1997.

[12] Marcum, D.L., "Unstructured grid generation using automatic point insertion and local reconnection." In Thompson, J., Soni, B., and Weatherhill, N., *Handbook of Grid Generation*, CRC press, Boca Raton FL., 1999.