MESHSTL USER GUIDE

PURPOSE

This program is designed to convert unstructured surface triangulations into volumetric hexahedral meshes for finite element analysis. The input file format can be either ASCII or binary STL and the output file format is Abaqus's INP.

DISCLAIMER

Since this program is still in the development stage, the validity of the output has not been verified for all input cases.

CONTENTS

DIRECTIONS

This program was purposely designed to be relatively simple to use. To compile and use this code, the user will need the GNU g++ compiler and Make capability. The directions are written for a Linux system.

1. Download "MeshSTL.tar.gz" and unpack it.
2. Generate the executable "meshstl" by using the `make` command and execute it.
3. Type the full path of the STL file you would like to convert and press enter. Make sure that you include the ".stl" suffix.
4. Type the number of hexes you would like per axis and press enter. The main body of the program will now execute. If the STL file is particularly large or you requested a large number of hexes, this will take some time.
5. The program has finished when it displays the total elapsed time.
6. The resulting INP file is written to the directory that the executable was stored in. Later versions will support writing to different directories.

METHODOLOGY

The process flow of this program can be reduced to four main steps:

1. Vertex Simplification
2. Octree Creation
3. Facet Assignment
4. Assessment of Void vs. Solid

VERTEX SIMPLIFICATION

Since STL format is practically ubiquitous with respect to surface representation, it was chosen as the input format. However, STL format is not optimal for data processing. It encodes the surface as a series of triangular facets, where each facet is assigned a Cartesian normal vector and three Cartesian vertices. Vertices are directly written as three 32 bit numbers in each instance instead of referencing a master set. Since each vertex is used by multiple facets, this means that the coordinate data is repeated excessively. In order to produce computational objects for analysis, each instance of the vertices has to be collated and linked to their respective facets.

It is relatively straightforward to read each successive vertex in the file and evaluate whether it has been generated yet by sequentially checking its Cartesian coordinates against the Cartesian coordinates of each existing vertex. However, this is slow and requires every digit of the coordinates to match. To allow for small discrepancies and speed the process, this program employs a coordinate conversion and hash table combination developed by Teschner et al. [1]. The hashing code (written in C++11) is shown below.

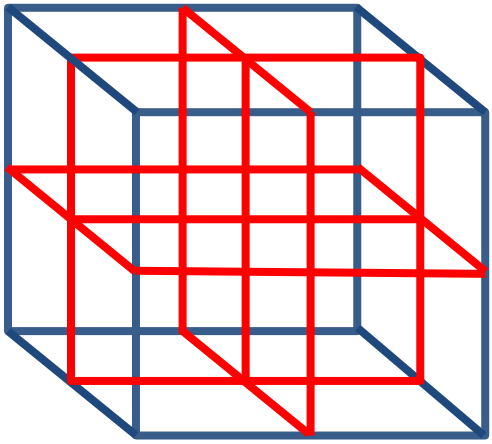
```
const unsigned long prime1 = 24036583;
const unsigned long prime2 = 13466917;
const unsigned long prime3 = 20996011;

unsigned long hash_coordinates(float x, float y, float z)
{
  unsigned long fx, fy, fz;
  fx = (unsigned long)floor(x/this_tolerance);
  fy = (unsigned long)floor(y/this_tolerance);
  fz = (unsigned long)floor(z/this_tolerance);
  unsigned long hash_val = (fx*prime1 xor fy*prime2 xor fz*prime3);
  return hash_val;
}
```

OCTREE CREATION

The octree subdivides Cartesian space into nested hexahedrals. The top level is a single hexahedral that encompasses the entire STL model. Each successive level is created by dividing a parent hexahedral in the previous level into eight equal child hexahedrals that are positioned in a 2x2x2 grid. This continues until the newly generated hexahedrals have shorter edge lengths than the threshold entered by the user. The bottom level contains all of the hexahedrals that were created by the final division.

Each hexahedral is defined by its vertices. When a parent hexahedral is subdivided, a new vertex is created at its center. The new child hexahedrals are dependent on the preexisting vertices of the parent hexahedral and the newly created center vertex. There are 2 significant limitations on the geometry of the hexahedrals. All angles must be right angles, and all edges must be of equal length. Future version will address these issues. A diagram of the hexahedral division is shown below.



FACET ASSIGNMENT

Before each of the bottom level hexahedrals can be evaluated as solid or not, the closest surface facet must be located. Testing the distance between each facet and hex would result in $O(n^2)$ operations, which is often too expensive for large, detailed surface meshes. Therefore, the octree is used to increase the efficiency of this task.

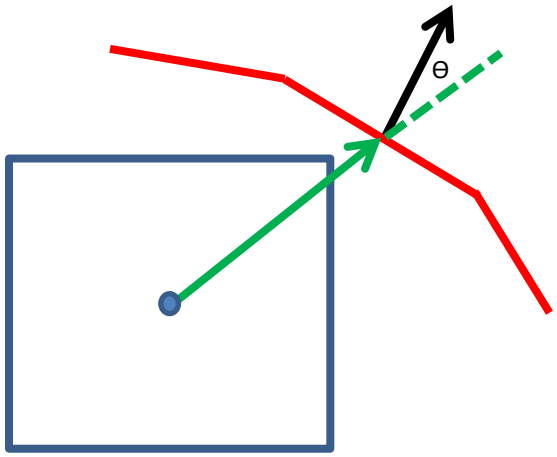
Each facet begins at the top level hexahedral. Since this hexahedral contains the entire surface mesh, the facet is guaranteed to lie within it. The program then begins to test which of the child hexahedrals of the current hexahedral contains the facet. Once the correct child hexahedral is found, the program begins to evaluate the child hexahedrals of that hexahedral. In this fashion, the facet trickles down the octree levels until it reaches a bottom level hexahedral. It is then assigned to this hexahedral for future use. To avoid confusion when the facet lies on a hexahedral boundary, the facets are treated like points, with the average of the vertices' locations dictating the point location.

If multiple facets are assigned to the same bottom level hexahedral, the Pythagorean distance between the center of each and the center of the hexahedral must be calculated to determine which lies closest to the center of the hexahedral. Because of this, a smaller edge threshold (and therefore smaller hexahedrals) can actually speed computation in some cases.

ASSESSMENT OF VOID VS. SOLID

Once each bottom level hexahedral has been assigned a facet, they can be assessed as either void or solid. This assessment is based entirely on the normal vector of the facets. It is therefore imperative that the normal vectors be pointed outward (away from solid areas).

First, a vector is constructed from the center of the bottom level hexahedral in question and its closest facet. Next, the angle between this vector and the normal vector of the closest facet is evaluated. Finally, the hexahedral is declared inside the surface (i.e. solid) if the angle is less than $\pi/2$ radians. A diagram of the 2D analogy is shown below. The blue square is the hexahedral, the red lines are the surface facets, the black vector is the normal, and the green vector is the constructed vector between the hexahedral and facet centers.



The most straightforward way to calculate the angle would be to evaluate the equation shown below.

$$\theta = \cos^{-1}\left(\frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|}\right)$$

However, this is an expensive calculation if there are a large number of hexahedrals to evaluate, and it produces more information than necessary. To determine whether the angle is less than or greater than $\pi/2$ radians, the program only needs to calculate $v_1 \cdot v_2$ because that dictates the orientation. If this value is greater than zero, then the hexahedral is labeled “solid”. Otherwise, it is labeled “void”. Once this has been accomplished, the solid hexahedrals and their corresponding vertices are written to the output file, which follows Abaqus’s INP format. That completes the execution of the program.

WORKS CITED

[1] M. Teschner, "Optimized Spatial Hashing for Collision Detection of Deformable Objects," *VMV,* vol. 3, pp. 47-54, 2003.