# AMS 562 Final Project—Spherical Triangular Meshes

Due Thursday, 12/17, **5:00 p.m.**

## 1. Introduction

*Mesh* is an essential concept in computational science, and it is used in many different applications; for instance, numerical partial differential equations (PDEs) problems. A mesh is a tessellation of a domain with a collection of simple geometry objects referred to as *elements* or *cells*. Mathematically, a geometric domain is defined infinitely; numerically, we have to tessellate the infinite domain with finite (a.k.a., discretized) and simple geometry objects.

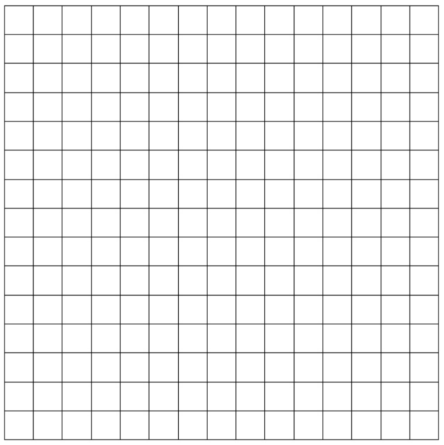
In general, meshes can be categorized into two families: 1) structured meshes and 2) unstructured meshes. As shown in Figure 1.1a, the structured mesh is topologically equivalent to a square with uniform spacing along both $x$ and $y$ axes (assuming 2D). However, it is hard for structured meshes to model complicated geometry domains despite their efficiency and user-friendliness. In contrast, unstructured meshes are widely used for modeling complex domains, as shown in Figure 1.1b. *In this project, we will learn and implement several mesh manipulation routines for unstructured meshes.*

Compared to structured meshes, unstructured meshes no longer allow us to conveniently traverse the coordinates by simple pairs of $(i,j)$ indices (or $(i,j,k)$ index triplets in 3D). Therefore, some specialized data representations are needed for unstructured meshes, which will be introduced in Section 2.

## 2. Data Representations

As mentioned in Section 1, a domain is typically tessellated with simple geometry objects; one popular choice is the triangle. We will deal with *spherical triangular meshes* for this project, i.e., unstructured meshes that tessellate the unit spherical surface (3D surface) with triangles.

2.1. **Storing coordinates and triangles.** For unstructured meshes, we, in general, do not store $xyz$ coordinates separately. Typically, an $n \times 3$ dense array is used, of which each $1 \times 3$ entry lists the coordinates


(A) Structured mesh in 2D

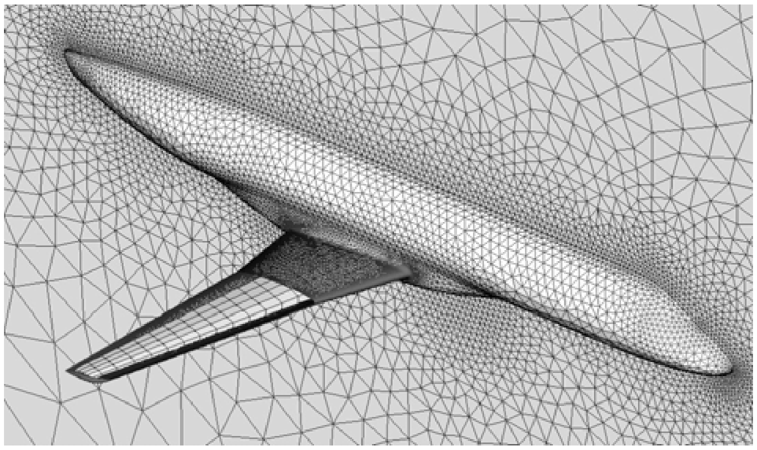

(B) Complicated modeling of an airplane with unstructured mesh

FIGURE 1.1. Two types of meshes. (A) A sample 2D structured mesh. (B) A sample 3D unstructured mesh with complicated geometry.

```
// Spherical coordinates
class SphCo {
  public:
  ...
  private:
    std::vector<std::array<double, 3>> _pts;
};
```

```
// Connectivity table
class Triangles {
  public:
  ...
  private:
    std::vector<std::array<int, 3>> _conn;
};
```

FIGURE 2.2. Triangular meshes with C++ classes for coordinates (left) and connectivity table (right)

in $x$, $y$, and $z$ axes. For instance, the following "matrix"

$$\begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix}$$

represents three points: $[x_i\ y_i\ z_i]$ for $i = 0, 1, 2$. An unstructured mesh is incomplete with only coordinates; indeed, the triangle information is missing. Triangles, which are topology connection of a set of coordinates, can be stored by an $m \times 3$ dense array, of which each $1 \times 3$ entry lists the *connectivity* of a triangle. For instance, [0 1 2] stands for a triangle formed by node indices 0, 1, and 2 in the coordinate array.

2.2. **An example of triangular mesh.** Here, we demonstrate the aforementioned concepts by giving a simple example of an unstructured mesh containing two triangles, as shown in Figure 2.1. As we can see, the mesh has four points; we can store them by $\boldsymbol{u} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$, where node $i$ in Figure 2.1 has coordinate $[x_i\ y_i\ z_i]$ for $i = 0, 1, 2, 3$. For the two triangles, a *connectivity table* of $2 \times 3$ can be employed as $\text{conn} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 3 & 2 \end{bmatrix}$, which indicates that the first triangle is formed by nodes 0, 1, and 2, and the second is by nodes 1, 3, and 2. To access each triangle's coordinates, we need to combine conn and $\boldsymbol{u}$. For instance, the first node's $x$-coordinate in the first triangle is given by $\boldsymbol{u}_{\text{conn}_{00},0}$ (or in C++ syntax, `u[conn[0][0]][0]`) and $\boldsymbol{u}_{\text{conn}_{11},1}$ (or `u[conn[1][1]][1]`) is for the second node's $y$-coordinate in the second triangle.

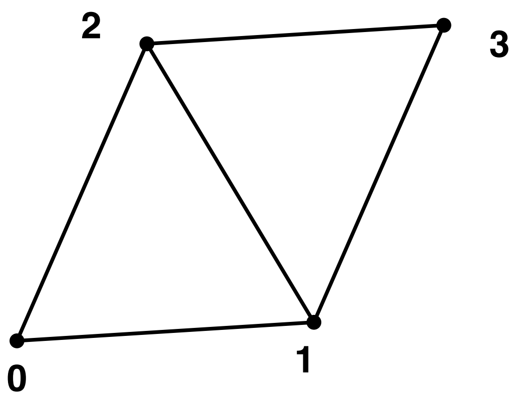


FIGURE 2.1. An unstructured triangular mesh with two triangles and the node IDs

2.3. **Implementation with C++ STL.** For both coordinates and connectivity table, we use `std::vector` as the container, and we use `std::array<double,3>` and `std::array<int,3>` for every single point and triangle connectivity, respectively. Figure 2.2(left) shows the interface for spherical coordinates, and Figure 2.2(right) is for the connectivity table. As we can see, the coordinates are stored with `std::vector<std::array<double,3>>` and `std::vector<std::array<int,3>>` for triangles.

2.4. **Note on node ordering.** The ordering of nodes within a triangle is important because it encodes certain critical geometry information. In this project, each triangle's node ordering follows the right-hand rule with the thumb direction pointing outward and perpendicular to the spherical surface. For instance, given a triangle with nodes $[A\ B\ C]$, then $\overrightarrow{AB} \times \overrightarrow{AC}$ is (approximately) the outward normal vector of the spherical surface locally.

**Algorithm 1** adj= **compute_n2e_adj**($n$, conn)

```
Inputs:
    $n \leftarrow$ total number of coordinates in the mesh
    conn $\leftarrow$ connectivity table of the input triangular mesh
Output:
    adj $\leftarrow$ node to triangle (cell) adjacency
1: initialize adj with size $n$.
2: for each triangle $\mathcal{T}_e$ in conn do
3:     for each node $v$ in $\mathcal{T}_e$ do
4:         push triangle ID $e$ to adj$_v$
5:     end for
6: end for
7: return adj
```

```cpp
// vector of vector, where the inner vector is the list of triangle ID (not node ID)
// that are shared by a specific node (index of the outer vector)
std::vector<std::vector<int>> n2e_adj;
```

Figure 3.1. Node-to-triangle adjacency with STL

## 3. Your Tasks

For this project, you do **not** need to implement the data representations introduced in Section 2. Instead, you need to focus on several algorithms that manipulate spherical meshes.

3.1. **Determine node to triangle adjacency.** Essentially, a connectivity table defines the adjacency information between triangles and nodes, i.e., a triangle in the connectivity table lists the adjacent nodes of the triangle. Starting from this, we want to write an algorithm that computes the *dual* adjacency, i.e., for a given node, its adjacent triangles are those that contain it. Unlike the connectivity table, for unstructured meshes, the node to triangle adjacency, in general, cannot be stored in matrix-like storage (why?), thus we need to use a nested `std::vector` to represent such adjacency information, as shown in Figure 3.1; Algorithm 1 provides a pseudo-code for computing node-to-triangle adjacency.

3.2. **Compute averaged outward unit normal vectors.** Given a triangle and assuming the node ordering in Section 2.4, each triangle face's outward normal vector can be carried out by *cross product*, i.e., $\boldsymbol{n} = \overrightarrow{AB} \times \overrightarrow{AC}$ given triangle $\triangle ABC$. We are interested in computing approximate nodal outward normal vectors, which can be done using a unit average of corresponding adjacent triangular facial outward normal vectors by leveraging the node-to-triangle adjacency information. This algorithm involves two stages: 1) Compute outward normal vectors of each triangle. 2) Average the triangular facial outward normal vectors based on node-to-triangle adjacency; for more details, refer to Algorithm 2; line 12 is the key to use the node-to-triangle adjacency computed by Algorithm 1.

3.3. **Analyze the approximation errors.** In general, Algorithm 2 introduces numerical errors because the normal vectors are not carried out on the exact geometry. Indeed, each triangle is a flat surface approximation to the spherical geometry. Therefore, we want to analyze how much error has been introduced with such approximation in this task. The metric we use is to determine the angles between our approximate nodal outward normal vectors and the analytic ones, i.e., the coordinates on a unit spherical surface. Specifically, the angle between the $i$th approximate and analytic normal vectors reads

$$\theta_i = \cos^{-1}\left(\frac{\boldsymbol{n}_i \cdot \boldsymbol{n}_i^e}{\|\boldsymbol{n}_i\| \, \|\boldsymbol{n}_i^e\|}\right),$$

**Algorithm 2** $\boldsymbol{n}$ = **compute_avg_normals**($\boldsymbol{u}$, conn, adj)

```
Inputs:
        $\boldsymbol{u}$: coordinates of size $n \times 3$
        conn: connectivity table of the input triangular mesh
        adj: node-to-triangle adjacency computed per Algorithm 1
Output:
        $\boldsymbol{n}$: average outward normal vectors for all nodes.
 1: $m \leftarrow$ number of triangles.
 2: initialize workspace for facial normal vectors $\boldsymbol{n}^f$ with size $m \times 3$.
 3: initialize $\boldsymbol{n}$ (output node-based normal vectors) with size $n \times 3$.
 4: for each triangle $\mathcal{T}_e$ in conn do
 5:     form vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$ in $\mathcal{T}_e$                    // use coordinates
 6:     $\boldsymbol{n}^f_e \leftarrow \overrightarrow{AB} \times \overrightarrow{AC}$
 7:     $\boldsymbol{n}^f_e \leftarrow \boldsymbol{n}^f_e / \lVert \boldsymbol{n}^f_e \rVert$                    // normalize the normal vector
 8: end for
 9: // perform averaging
10: for each node $v$ in the mesh do
11:     $k \leftarrow$ number of adjacent triangles of $v$
12:     $\boldsymbol{z} \leftarrow \left( \sum_{j=0}^{k-1} \boldsymbol{n}^f_{\mathsf{adj}_{v,j}} \right) / k$          // sum over all adjacent normals and average them
13:     $\boldsymbol{n}_v \leftarrow \boldsymbol{z} / \lVert \boldsymbol{z} \rVert_2$                    // normalize the normal vector
14: end for
15: return $\boldsymbol{n}$
```

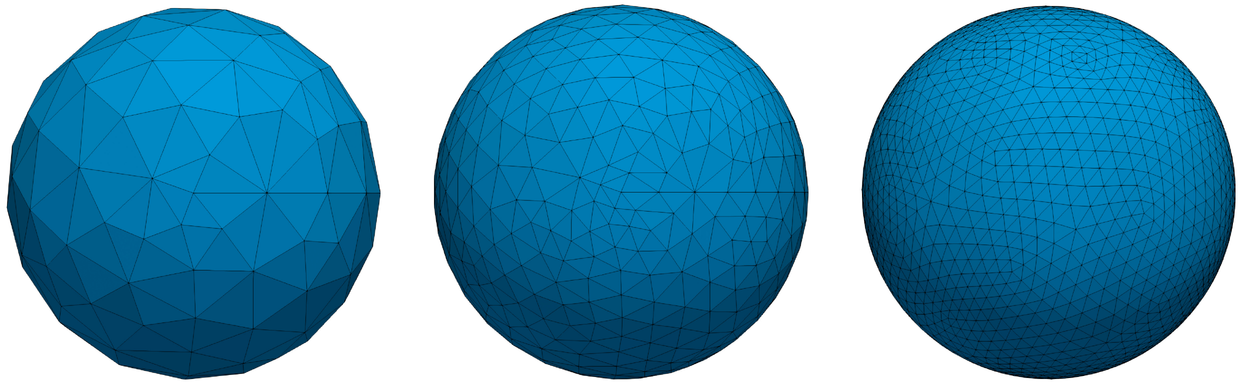


FIGURE 4.1. Three levels of spherical triangular meshes used in this project

where $\boldsymbol{n}$ is our approximated results per Algorithm 2 and the analytic unit normals for $\boldsymbol{n}^e$;[1] $\cdot$ is the inner product of two vectors. It is relatively easy to verify that $\theta$ are the angles between $\boldsymbol{n}$ and $\boldsymbol{n}^e$, which we refer to as *error angles*. We omit the pseudo-code for this part.

## 4. Requirements

You need to implement the aforementioned tasks in Section 3 in the `manip.cpp` file and make sure you pass all tests. The workflow is similar to the midterm project. In addition, there are four global[2] iterative loops, of which you **must** implement two of them with `std::for_each` with either `lambda` or *functors*. The test program will be performed on three different meshes with different densities; see Figure 4.1.

[1]Notice that $\lVert \boldsymbol{n}_i \rVert \equiv 1$ and $\lVert \boldsymbol{n}^e_i \rVert \equiv 1$.

[2]By "global," it means loops performed over the whole mesh, either over all nodes or all triangles.

## Supplementary points for midterm

Using the combination of `std::for_each` and `lambda`/functors for more than two loops will give you a chance to recover some points for the midterm project. Specifically speaking, you can recover **at most** 20 points for the midterm if you got **less than** 100 in the midterm.