AMS 562 Final Project—Spherical Triangular Meshes

Due Dec. 20, 2018 5.00 pm

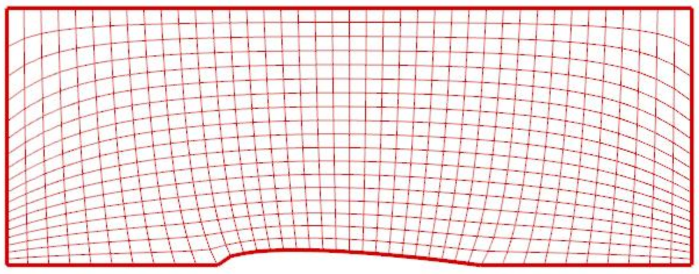
1 Introduction

Mesh is an important concept in computational science; it is used in many different applications, for instance, most PDE based engineering problems. A mesh is a tessellation of a domain with simple geometry objects—referred as *elements* or *cells*.

In general, mesh can be grouped into the following two families:

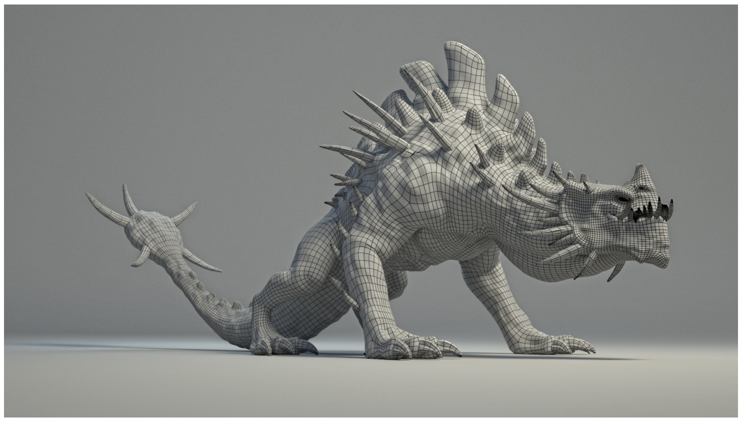
1. structured grids, and
2. unstructured meshes.

The first one is topologically equivalent to unit square (in 2D) with uniformly distributed (in some weighted sense) grid lines along x and y directions. Therefore, each of the grid points can be accessed with a pair of indices, i.e. x-/y- index, see below.



Structured grids are hard to extended to general domains that are needed in typical engineering problems. This is particularly true for structural mechanics, whereas the structural components can be very complicated.

With this consideration, people have come up with the *unstructured meshes*, which can easily represent complex geometries; see the picture below for a surface mesh on a dragon shape object.



For unstructured meshes, one of the popular choices of element types for surface domain tessellation is *triangles*, because they are simple and robust (tessellating a surface with triangles is much easier than with other cells, e.g. quadrilateral cells).

However, with unstructured grids, it's impossible to access the grid points with x-, y-, (z-) index. Therefore, special data structures are needed (later).

For this project, we will only deal with unstructured triangular meshes on spherical surfaces.

2 Data Structure

2.1 Core Concept for Storing Points and Triangles

For unstructured meshes, one cannot store, like in structured grids, xyz coordinates separately. Typically, a n by 3 storage is utilized, where each entry of the array stands for a physical point in the *Cartesian* coordinate system. For instance, you can store five arbitrary points in the following "matrix" format.

$$\begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{bmatrix}$$

In order to represent the mesh, we use a *connectivity table* that forms the connection relation between triangles and coordinates IDs. A connectivity table (for triangles) is an m by 3 integer storage, where m is the number of triangles, and the three integer IDs are the node IDs in the coordinates that form the triangle.

2.2 An Example

Let's consider a unit square plane, we then can split the plane into two triangles along one of the diagonal (say, [0 0] and [1 1]). Therefore, we have 4 points and 2 triangles and the mesh can be represented by (assume xy plane)

$$points = \begin{bmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \\ 1.0 & 1.0 \\ 0.0 & 1.0 \end{bmatrix}$$

$$triangles = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 3 & 0 \end{bmatrix}$$

In this example, we have 4 points with the coordinates listed in *points*, then in the connectivity table, i.e. *triangles*, the first triangle contains nodes 0, 1 and 2, while 2, 3 and 1 for the second one.

Hint draw this with pencil and paper.

2.3 Store The Mesh with C++ STL

For both coordinates and connectivity table, we will use the `std::vector` as the container, and for each single point and triangle, we will use `std::array`. Therefore, the overall data structures are:

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

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

With this implementation, we know that when you loop through the coordinates and connectivity table, the entries are of types `std::array<double, 3>` and `std::array<int, 3>`, resp.

2.4 Remark Regarding Triangle Node Ordering

For the spherical mesh, you can assume that the node ordering of triangles following the *right-hand rule* results vectors that always point to outward normal directions with respect to the spherical triangular faces.

3 Your Tasks

For this project, you do **NOT** need to implement any of the data structures listed above. They are already implemented and ready to use. What you need to do is to implement several algorithms in order to perform the following tasks.

3.1 Determine Node to Triangle Adjacency

Essentially, a connectivity table defines the adjacency information between triangles and nodes, i.e. a row of the table is the adjacent nodes of that triangle. Starting from this, we can build its dual graph, i.e. node to cell adjacency, so that, given a node, we can find all triangles contain it.

Unlike the connectivity table, for unstructured meshes, the node to cell adjacency information, in general, cannot be stored in a matrix-like storage, we, then, can use

```cpp
// vector of vector
// where the inner vector is the list of element IDs that
// are shared by a specific node
std::vector<std::vector<int>> n2e_adj;
```

In order to build the adjacency information, you need to implement the following algorithm.

```
*Algorithm I*: Determine the node to cell adj
Inputs: Connectivity table $conn$ and number of points $np$
Output: Node to triangle adjacency mapping, $adj$
Initialize $adj$ with size $np$
**for** each triangle $T_i$ in $conn$, **do**
    **for** each node $v_j$ in $T_i$, **do**
```

```
        push back triangle ID $i$ to list $adj_j$
    end for
end for
```

3.2 Compute the Averaged Outward Normal Vectors

Given a triangle, we can compute its normal vector by using the *cross product*, i.e. for a triangle ΔABC, the normal vector $\vec{\boldsymbol{n}}$ is computed by

$$\vec{\boldsymbol{n}} = \vec{AB} \times \vec{AC}$$

where $\times$ denotes cross product.

Since we already know that the triangle is oriented in the way that the right-hand rule gives the outward normal, directly using the formula above will give you the correct answer.

We want first compute and store all outward normals of triangles, then use the adjacent mapping to average them on nodes.

```
Algorithm II.1: Compute the outward normals of spherical triangles
Inputs: Connectivity table $conn$ and coordinates $co$
Output: Outward normals of triangles, $f\_nrms$
Initialize $f\_nrms$ with size $ne = conn.size()$
for each triangle $T_i$ in $conn$, do
    Get coordinates A, B, C = $co(T_{i,0})$, $co(T_{i,1})$, $co(T_{i,2})$
    Assign $f\_nrms_i$ with the outward normal vector
end for
Normalize $f\_nrms$ so that each of the vector has Euclidean length 1
```

In order to easily normalized the vectors, you should consider use `SphCo` to store f_nrms.

```
Algorithm II.2: Compute the averaged normal on nodes
Inputs: Node to cell mapping $adj$, $np$, and $f\_nrms$
Output: Outward normals of nodes, $n\_nrms$
Initialize $n\_nrms$ with size $np$
for each node $v_j$, do
    Get the local adjacency information $adj_j$
    Assign $n\_nrms_j$ to be the average of $f\_nrms(adj_j)$
end for
Normalize $n\_nrms$ so that each of the vector has Euclidean length 1
```

3.3 Determine the Computation Errors

For points located on the spherical surface, their outward normal directions are uniquely defined. In this case, we want to see how much error we have introduced from our computation. The metric we use is

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = \|\vec{\boldsymbol{a}}\| \|\vec{\boldsymbol{b}}\| \cos(\theta)$$

where θ is the angle between $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{b}}$. We, then, have

$$\theta = \cos^{-1}\left(\frac{\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}}}{\|\vec{\boldsymbol{a}}\| \|\vec{\boldsymbol{b}}\|}\right)$$

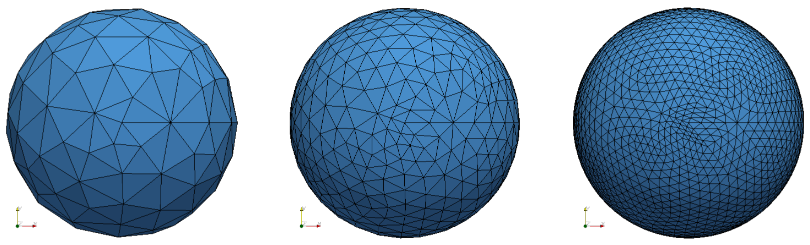
Since we know that both $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{b}}$ are normalized, we can simplify the computation by

$$\theta = \cos^{-1}\left(\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}}\right)$$

```
Algorithm III: Compute the angle errors
Inputs: Analytic normals (coordinates) $exact$ and numerical normals $num$
Output: Error angles $arccos\_theta$
Initialize $arccos\_theta$ with size $exact.size()$
for each node $v_j$, do
    Assign $arccos\_theta_j$ with the error angle given $exact_j$ and $num_j$ normal vectors
end for
```

4 Requirements

You need to implement the algorithms list above in `manip.cpp` file and make sure you pass all tests. There are three test cases, which are list below.



Note that there are 4 *for* loops, you must implement **at least** two of them with `std::for_each` with `lambda` or traditional *functors*.

You will **gain** extra credits by using `std::for_each` for all iteration loops.

5 Hints

There are some sample usages of `std::for_each` and `lambda` in `co.hpp` and `conn.hpp`. Also, type `make debug` to use Valgrind during development. Please look at the comments in `manip.hpp` carefully.

If you get error values of `nan`, then it means you probably have some bug in `adj` array (you have at least a node with no adjacent triangles). If you get error values of ~ 180 degree, then it means you mess up with the node ordering so that the right-hand rule gives you inner normal directions.