# FEM Solver Programing

## USING GMLIB AND CLASS TRIANGLEFASETS

hxu001 | STE6291 | February 26, 2016

# Abstract

This report is about a study of programming in solving FEM (Finite Element Method) problems and simulating the pressure on a drumhead. In this study, a finite element program is created base on partial differential equations and FEM. In the report, it is briefly described the theoretical problem and a description of the program and the implementation is also given in the main part. The projected is designed by C++ Programming Language in Qt environment. GMlib and the class TriangleFasets have been used. Meshing of drumhead includes both a regular point set and a random generated point set. The work was carried out by Department of Computer technology and computation oriented engineering in UiT-The Arctic University of Norway.



*Key words:* FEM, C++, GMlib, simulate, stiffness matrix

# Contents

# 1 Introduction

In this study, a finite element program is created base on partial differential equations and FEM. The problem is a 2D example of FEM which is to simulate a deformation of drumhead under the changing force. The drumhead will be meshed with triangles in both random and regular ways. This report has four main parts which are introduction, theoretical description of the problem, implementation and conclusion. This program was implemented by C++ using GMlib and class TriangleFasets.

# 2 Theoretical description of the problem

The Finite Element Analysis (FEA) is a numerical method for solving problems of engineering and mathematical physics. As shown in Figure 1, there is an example of FEM in 2-D. The surface was meshed by triangles. From the sketch, h is the maximum side length in this triangulation $T_h = \{K_1, K_2, \dots, K_m\}$, $\beta_h$ is the smallest angle of the triangulation. We assume that there is always a positive angle $\beta_h$ bigger than a constant k ($\beta_h > k > 0$), then we can imagine a sequence of such triangles in the way that they are getting smaller and smaller, which means that the h is approaching to 0. But in a way that the $\beta_h$ is not approaching to 0 but always larger than the constant k, then it is possible to show that the finite element solution converges to an actual solution in Sobolev surface for the Dirichlet problem[1]. That is why we can solve the problem in a numerical way.

As we known that from weak formulation [2] (Equation 2.1), the bilinear form $U_h$ is the finite solution and it must satisfied the weak formulation.

$$a(U_h, v) = L(v), \text{ for all } v \in V_h \tag{2.1}$$

If we consider the solution, $U_h$ must be written as following form (Equation 2.2):

$$U_h = x_1 v_1 + x_2 v_2 + \cdots + x_m v_m \tag{2.2}$$

We do not know the values, but we are going to find $z_1, z_2, \dots, z_m$ . When we find these values, we will have the finite element solution. So we put the Equation 2.2 into the Equation 2.1, we will get $L(v)$ as following (Equation 2.3):

$$\begin{aligned} L(v) &= a(U_h, v) \\ &= a(x_1 v_1 + x_2 v_2 + \cdots + x_m v_m\ , v) \\ &= x_1 a(v_1, v) + x_2 a(v_2, v) + \cdots + x_m a(v_m\ , v) \end{aligned} \tag{2.3}$$

The solution for each of the $L(v)$ can be written as a matrix form showing in Equation 2.4:

$$\begin{bmatrix} L(v_1) \\ \vdots \\ L(v_m) \end{bmatrix} = \begin{bmatrix} a(v_1, v_1) & \cdots & a(v_m, v_1) \\ \vdots & \ddots & \vdots \\ a(v_m, v_1) & \cdots & a(v_m, v_m) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \tag{2.4}$$

The equation above also can be written in a simple form showing in Equation 2.5:

$$A\vec{x} = \vec{b} \tag{2.5}$$

The entire element in the matrix A can be calculated. We also call this matrix stiffness matrix. This problem will be solved by getting the $\vec{x}$ since $\vec{x} = A^{-1}\vec{b}$ .

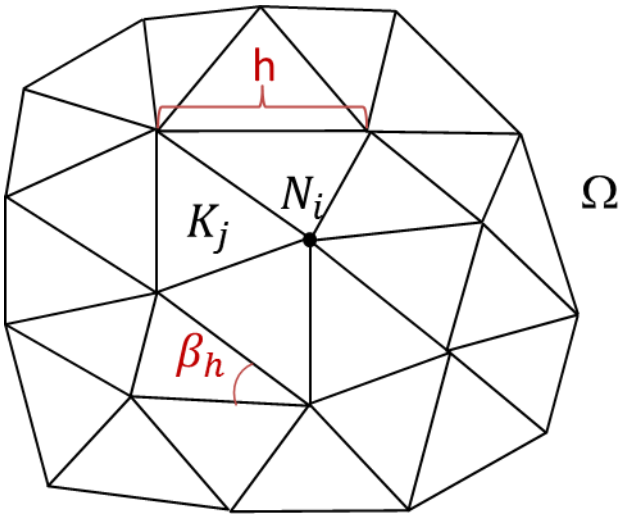


*Figure 1 A triangulation of $\Omega$*

# 3 Description of the program and the implementation

In this project, the final solution is to simulate the pressure on a drumhead with no specific material properties. So we have to compute the stiffness matrix and the load vector to get the new vertices values (z-values) according to the following formula (Equation 2.5) which we explained in previous pages. In my programming, I wrote controller which is mesh.h and mesh.cpp files. And I created an element class called and implemented some code inside the gmlibwrapper.cpp file.

## 3.1 Point set

### 3.1.1 Random triangles

First of all, I created a point set in both random and regular way. When we create the point set in a random way to get triangle mesh inside the drumhead, we need to set the number of points and number of points on the boundary first. So we have the number of triangles as shown in following formula (Equation 3.2):

$$t = 2 * n - 2 - k \tag{3.2}$$

Where t is the number of triangles, n is the number of inner points, k is the number of points on the boundary.

As shown in Figure 1, we assumed that the length of the triangle is s, and the radius of the circle is R, so we can get the number of point k as following (Equation 3.3):

$$k = 2 * \pi * R/s \tag{3.3}$$

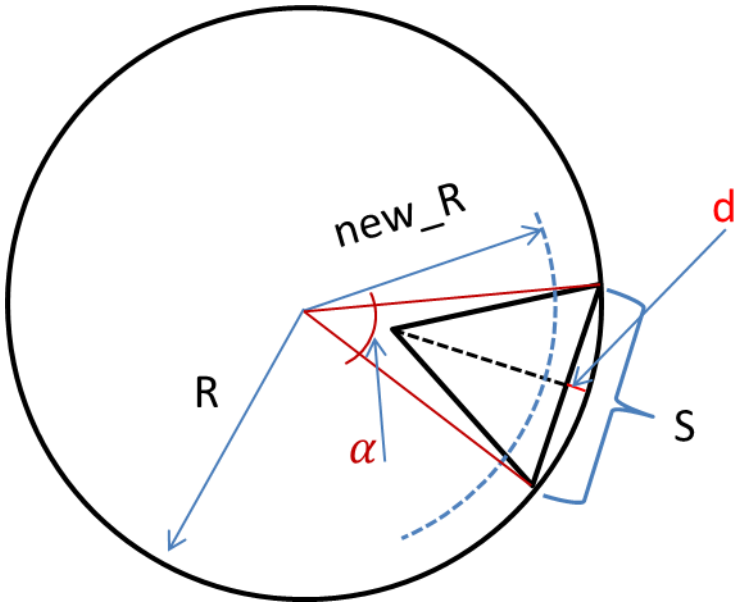


*Figure 2 Schematic diagram of creating random triangles inside the circle*

We assumed that it is the equilateral triangle inside the circle, the area of the triangle can be expressed as following:

$$A = \frac{\sqrt{3}}{4} * s^2 \tag{3.4}$$

Where s is the length of the triangle, A is the area of the triangle.

Since we know the numbers of triangles in the circle, we can compute the area of triangles in another way which is dividing the area of the circle by the number of the triangles. So we can get the length of the triangle s is:

$$s = \sqrt{\frac{4 * \pi * R^2}{\sqrt{3} * t}} \tag{3.5}$$

Where t is the number of the triangles, R is the radius of the circle.

Also we need to compute the angle of triangle. First of all, I set a vector which has length of the radius of the circle R and 0 degree initially. Therefore, by turning vector with the rotating angle $\alpha$ we can create the triangle all over the circle by using all the points on the boundary. The angle can be computed from following formula (Equation 3.6):

$$\alpha = 2 * \pi / k \tag{3.6}$$

Where $\alpha$ the rotating angle, k is the number of the points on the boundary.

In the code, I use 'auto' to get the vector from GMlib :: vector <float, 2> (rad, 0) times the Square Matrix which I got from GMlib :: SqMatrix <float, 2> to create the triangle one by one until I use all of the points on the boundary by using a 'for loop'. The angle $\alpha$ is also got from GMlib :: Angle. Inside the 'for loop', I use 'insertAlways' to get the entire vectors every time.

Since we have to create the triangle from outside the circle to the center of the circle layer by layer, we need to get new radius to repeat previous steps. First I computer the distance between the circle and the length of the triangle d which is the short red line shown in Figure 1 by using Equation 3.7:

$$d = R - R * \cos(\alpha/2) \tag{3.7}$$

Where d is the distance between the circle and the length of the triangle, $\alpha$ is the rotating angle, R is radius of the first circle.

Then I got the new radius from following equation (Equation 3.8):

$$new\_R = R - 2 * d \tag{3.8}$$

Where new_R is the radius of the new circle, R is the radius of the previous circle, and d is the distance between the circle and the length of the triangle.

In order to get the points inside the circle as well as to avoid the point to close to create random triangle become very slim, I use bool checkP function to set the conditions. I check the point if it is too close to the length that is less than half of the length of the triangle as well as if it is too long that longer than the radius of the circle. I only insert the points which are inside and also meet the requirements.

In addition, we need to change the type of random points from int to float since we need float in GMlib::TSVertex but it is int in std::rand().

### 3.1.2 Regular triangles

When I create the points set in a regular way to get triangles, I insert the center point first, and find the points on the circles from inside to outside layer by layer. In this function, we need three parameters: number of circles, radius of the circle and number of inner points.

First of all, the vector should be defined as when we made the random ones. In this situation, the radius of circles will be increasing with the number of the circles. We need to define a rotating angle as well. The angle will be decreasing with the number of the circles. From Figure 2, we can get the formula as following (Equation 3.9, Equation 3.10):

$$Rad = R * j \tag{3.9}$$

Where Rad is the new radius of the circle, R is the radius of the first circle, j is the order of the circle.

$$\beta = 2 * \pi/(n * j) \tag{3.10}$$

Where $\beta$ is the rotating angle, n is the number of the inner nodes, j is the order of the circle.

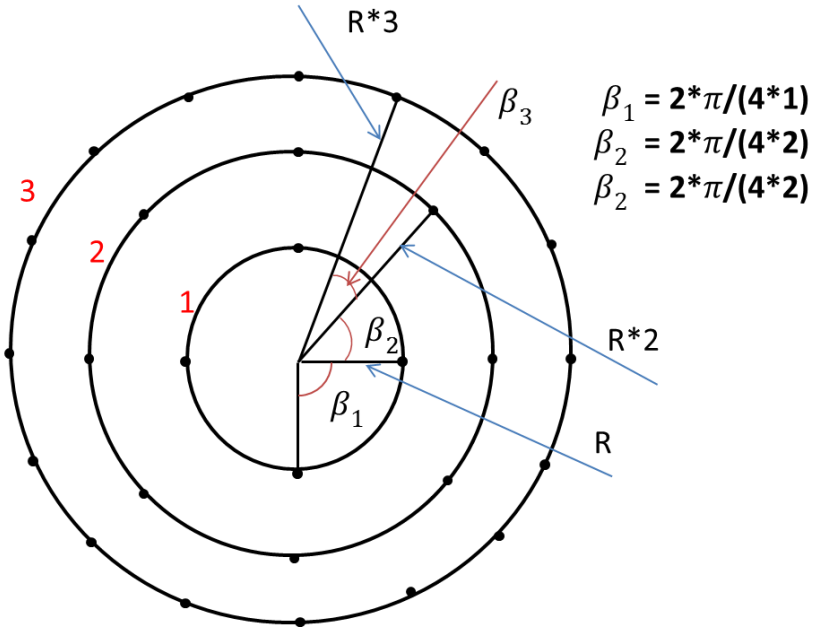


*Figure 3 Schematic diagram of creating regular triangles inside the circle*

Same as doing random triangles, I use the vector times the Square Matrix which I got from GMlib :: SqMatrix <float, 2> to insert the vectors one by one until I use all of the points on the circle each time.

## 3.2 Computing the stiffness matrix

From the general theory of finite element method (FEM) we know that stiffness matrix is a square matrix with the size of the number of elements. The stiffness matrix can be written as following form:

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix}$$

Each of the elements inside the stiffness matrix A will have following equation:

$$A_{ij} = \int_{\Omega} < d\varphi_i, d\varphi_j > d\Omega \tag{3.11}$$

Where $\varphi_i$ and $\varphi_j$ are both the elements inside the boundary. See Figure 4.

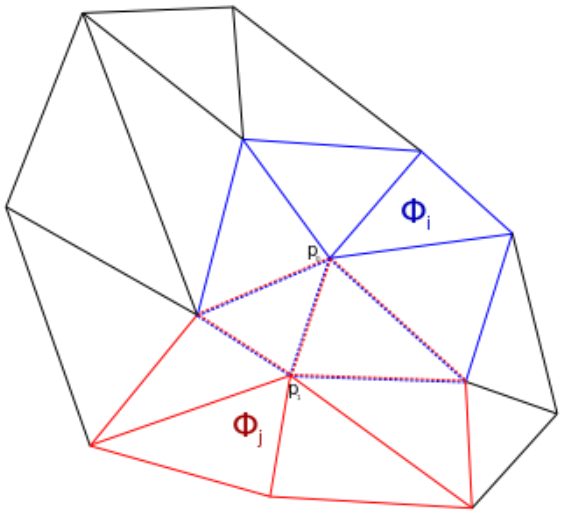


*Figure 4 Elements inside the boundary*

In this project, we mesh the drumhead with triangles, so each of the elements will look like a tent combined with triangles (see Figure 5 *[3]*). In this tent function, the shaded area is the support of the i'th (order of the nodes) basis function. This basis function equals to one in this node point. It is linear in each of these triangles and it is zero on the boundary of the support. Therefore in the stiffness matrix A, if the two elements don't have common area, the value will be zero. So we only need to compute the diagonal elements and the outside the diagonal elements when there is an overlapping.

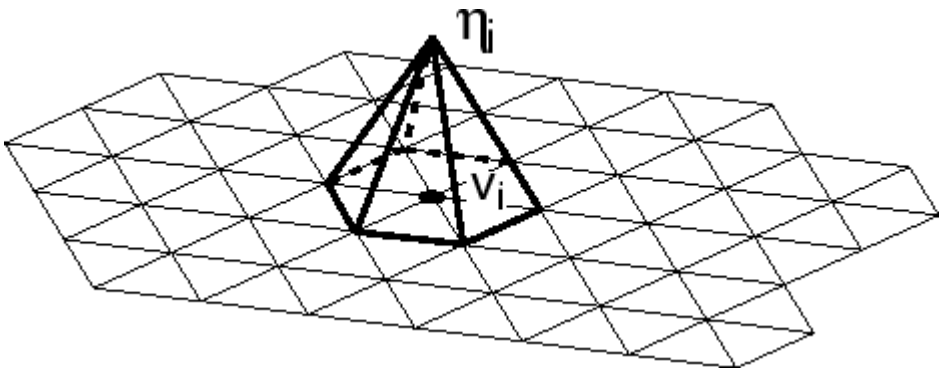


*Figure 5 Linear basis function for a triangulation in 2D* [3]

In the programming, I created a class called element and use function getEdges() and getTriangles() to make triangles with all the nodes. In my controller, I have findElement() function to compute the stiffness matrix A and the load vector $\vec{b}$. Before I start to compute the value in the stiffness matrix, I remove the boundary nodes. I made a 'for loop' to check all the members in my 'mesh' file if the member is not the boundary, added it in the container. And then I set a square matrix A and matrix b with initial value zero inside.

### 3.2.1 Computing Stiffness matrix elements outside the diagonal

According to the theory, the outside the diagonal elements in the stiffness matrix can be derived by using Equation 3.11, so we get Equation 3.12:

$$A_{ij} = A_{ji} = \int_{T_1} < d\varphi_i, d\varphi_j > dT_1 + \int_{T_2} < d\varphi_i, d\varphi_j > dT_2 \tag{3.12}$$

Where $A_{ij}$ and $A_{ji}$ are the elements of the matrix, $T_1$ and $T_2$ are the triangles, $\varphi_i$ and $\varphi_j$ are the mesh elements.

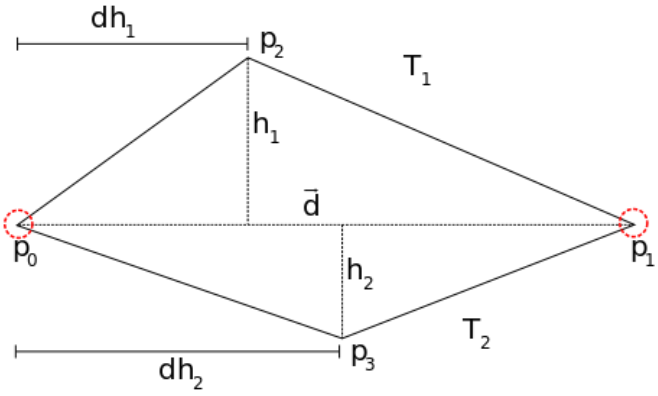


*Figure 6 Overlapping area of two neighbor elements*

First of all, we need to find the neighbor triangles since all the calculation are related to each pair of adjacent triangles (see Figure 6). In the programming, the most important is to find the common edge of the two triangles. This means that if the two triangles have common edge, they are adjacent. In order to compare each node's edge, we need to have two 'for loops' to make two containers. In the second loop, I set j=i+1 to avoid it compare with itself. Use getEdges() to make two edges container from all of the nodes except boundary nodes. Inside the two 'for loops' it still has another two 'for loops' to check if the edges1 and edges2 are equal. Common edge will be found when they are equal. If the two triangles have common edges, we will do the following computing(Equation 3.13-3.20):

$$dd = \frac{1}{< \vec{d}, \vec{d} >} \tag{3.13}$$

Triangle(T1):

$$area_1 = \left|\vec{d} \wedge \vec{a}_1\right| \tag{3.14}$$

$$dh_1 = dd \cdot < \vec{a}_1, \vec{d} > \tag{3.15}$$

$$h_1 = dd \cdot area_1 \cdot area_1 \tag{3.16}$$

Triangle(T2):

$$area_2 = \left|\vec{a}_1 \wedge \vec{d}\right| \tag{3.17}$$

$$dh_1 = dd \cdot < \vec{a}_2, \vec{d} > \tag{3.18}$$

$$h_2 = dd \cdot area_2 \cdot area_2 \tag{3.19}$$

$$A_{ij} = \left(\frac{dh_1 \cdot (1 - dh_1)}{h_1} - dd\right) \cdot \frac{area_1}{2} + \left(\frac{dh_2 \cdot (1 - dh_2)}{h_1} - dd\right) \cdot \frac{area_2}{2} \tag{3.20}$$

In order to compute the above formula, we need to know $\vec{d}$ , $\vec{a}_1$ and $\vec{a}_2$ since we have following relationship (Equation 3.21-3.23):

$$\vec{d} = p_1 - p_0 \tag{3.21}$$
$$\vec{a}_1 = p_2 - p_0 \tag{3.22}$$
$$\vec{a}_2 = p_3 - p_0 \tag{3.23}$$

Therefore, we need to find out $p_0$, $p_1$ ,$p_2$ and $p_3$. From Figure 6, it is obviously that $p_0$ and $p_1$ is the nodes of common edge, $p_0$ and $p_1$ can be the third nodes of the two triangles. In the programming, I made findVector() to compute these vectors. I set $p_0$ as the first vertex of the common edge by using Gmlib::TSEdge::getFirstVertex(), and $p_1$as the last vertex of the common edge by using Gmlib::TSEdge::getLastVertex(). Use commonEdge->getTriangle() will get 2 triangles. Then make two containers for putting the vertices from each of the triangles separately. Using 'for loop' to check the vertex if it is $p_0$ or $p_1$. Put the point which neither is $p_0$ nor it is $p_1$to the container. Inside the 'for loop', it needs to check the two triangles respectively to set $p_2$ and $p_3$.

### 3.2.2 Computing stiffness matrix diagonal elements

The values on the diagonal of the stiffness matrix are where the elements is computed against itself (see Figure 7). According to the theory, we know that the elements can be computed by using following formula (Equation 3.24-3.28):

$$A_{ii} = \sum_{k=0}^{l} T_k \tag{3.24}$$
$$\vec{d}_1 = p_2 - p_0 \tag{3.25}$$
$$\vec{d}_2 = p_1 - p_0 \tag{3.26}$$
$$\vec{d}_3 = p_2 - p_1 \tag{3.27}$$
$$T_k = \frac{< \vec{d}_3, \vec{d}_3 >}{2 \cdot |\vec{d}_1 \wedge \vec{d}_2|} \tag{3.28}$$

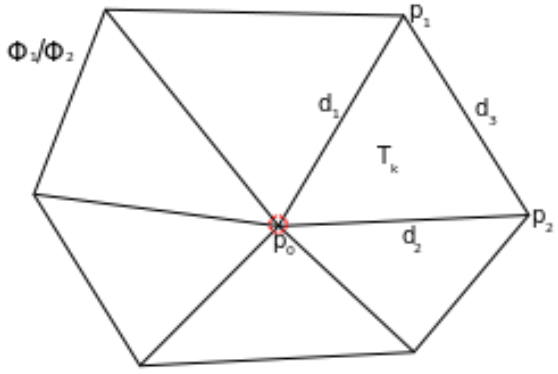


*Figure 7 Two meshing elements overlapping on itself*

Same here, we still need to find out $p_0$, $p_1$ ,$p_2$ first. $p_0$ is the special nodes which is connecting with all the triangles in one element, I set $p_1$ and $p_2$ followed $p_0$ in the counterclockwise direction. I want all the nodes of triangles to have fixed order with $p_0$, $p_1$ and $p_2$ as shown in Figure 8. If the position of $p_0$ is taken by $p_1$, use std::swap() function to change it back. First change $p_0$ and $p_1$, and then change $p_1$ and $p_2$. If the position of $p_0$ is taken by $p_2$, still use std::swap() function to change it back. First change $p_0$ and $p_2$, and then change $p_1$ and $p_2$. In the end, we can use the formula to compute $T_k$ to get $A_{ii}$.

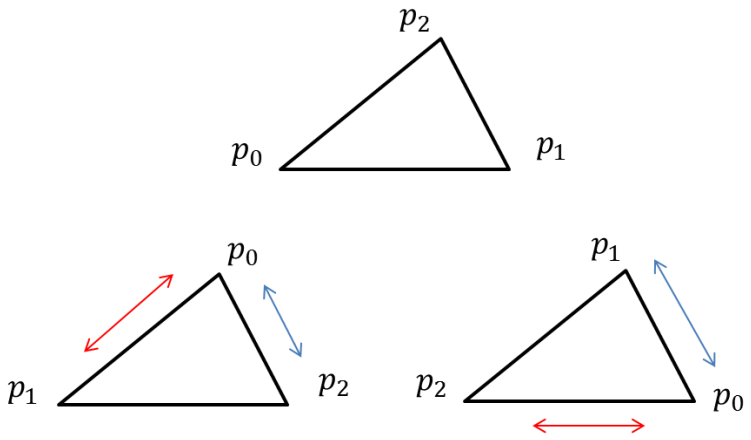


*Figure 8 Find* $p_0$

## 3.3 Computing the load vector

The load vector is the volume of the triangles. Since it is the element itself, it will use the same formula to get $\vec{d}_1$ and $\vec{d}_2$ . The volume of the triangles can be computed by using following equation (Equation 3.29):

$$volT_k = \frac{|\vec{d}_1 \wedge \vec{d}_2|}{6} \tag{3.29}$$

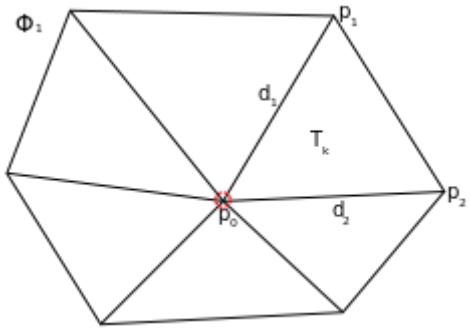


*Figure 9 Load vector for each of elements*

## 3.4 Compute values for new vertices and replot

In the localSimulate() function, I made a oscillating function that force (defined in the head file and set initial value zero) is changing with the time. When x times the force, it makes the x value (in fact it is shown in z direction) changing with the force. And then replot it.

In the gmlibwrapper file, insert both random and regular mesh on the same scene. Use translate() locate them properly. For better effect, set AliceBlue color for the background by using GMlib::GMcolor to add in projection camera section in gmlibwrapper file.

# 4 Conclusion

As shown in Figure 10, to the same radius of drumhead, we got the same result in both random mesh and regular mesh when giving the same force. From this course I learn a lot, not only C++ and GMlib, but also FEM. Solving FEM problem used to be my specialty. When I programming this simulation, I understand more about the theory and also feel easier to practice coding in C++.

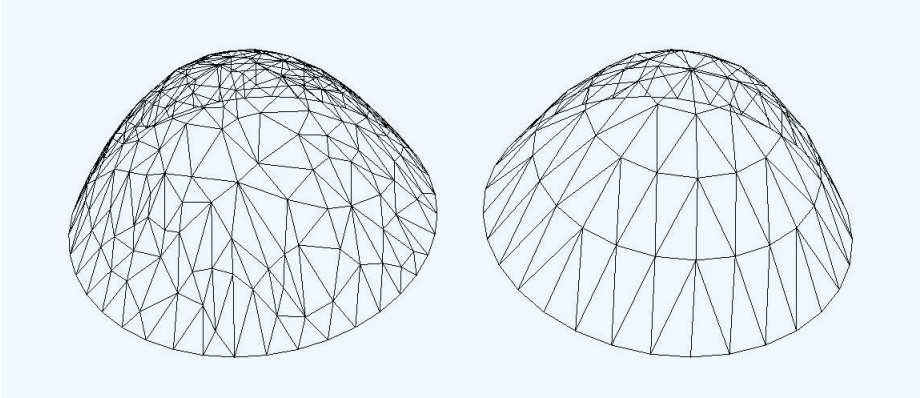

*Figure 10 Result of comparing in both random mesh and regular mesh*

# Acknowledgement

I think I am still a beginner on the way of learning programming. I am still struggling with finding logical errors. I got lots of help from teachers and classmates. I would like to acknowledge their helps.

## REFERENCES

1. Johnson, C., *Numerical Solution of PDE by FEM*. 1987, Cambrige Univ. Press.

2. Reddy, J.N., *An introduction to the finite element method*. 1989, American Society of Mechanical Engineers.

3. Scholz, W., et al., *Scalable parallel micromagnetic solvers for magnetic nanostructures.* Computational Materials Science, 2003. **28**(2): p. 366-383.