Finite Element Method Programming Report

Daniel Aaron Salwerowicz

March 2, 2018

Abstract

Purpose of this report is to present my work on developing a C++ based solver for Partial Differential Equation (PDE) in $\mathbb{R}^2$ which is based on Final Element Method (FEM).

1 Introduction

In this project I have utilized a geometric modeling library developed by Simulations R&D group at UiT called GMlib. In addition to that I have used Qt, and C++. These tools were used to create a physical simulation of a circular membrane in $\mathbb{R}^2$ affected by a constant, external force. I achieved it by numerically solving Poisson equation describing distortion of the aforementioned object. To simplify the problem I have assumed that the force is applied uniformly and normally to the membrane, I have also disregarded the material properties and time.

After simplifying the original PDE to a weak expression I have implemented methods to discretize the continuous surface using "finite elements", calculate stiffness matrix and load vector, solve the equation for a given force, and show the results in form of an animation.

2 Methods

Methods used to solve this problem included:

1. Triangulating membrane and defining basis functions.
2. Calculating stiffness matrix A and load vector b.
3. Solving matrix equation:
$$x = A^{-1}b \tag{1}$$
4. Updating position for internal nodes in the triangulated membrane.

2.1 Tools used

2.1.1 GMlib

GMlib gave me several useful classes and methods that I've based my solution on. `TriangleFacets` class was used as a basis for `FEMObject` class and `Vertex` class that was used in creating `Node` class. Other classes that were used in this solution inculde `Edge` and `Triangle` as well as `triangulateDelaunay` method. This library came along with a Qt wrapper project called `QmlDemo` that was used to visualize the solution using the Qt 3D modeling engine.

2.1.2 FEMObject

This class extends `TriangleFacets` class, which in turn extends `ArrayLX` class. I've used this class to create mesh of vertices used to triangulate the membrane, calculate stiffness matrix and load vector, solving the matrix equation (1), and animating the membrane.

2.1.3 Node

This class extends `Vertex` class and implements some useful methods that `FEMObject` uses in calculation of A and b.

2.2 Membrane triangulation

To triangulate membrane in $\mathbb{R}^2$ I've used Delaunay algorithm implementation found in `TriangleFacets` since it's optimal for this case. One of the advantages of Delaunay triangulation is that it maximizes the minimal angle in all of the triangles. Delaunay algorithm requires a mesh of nodes to create triangles, therefore I have defined two types of mesh generation algorithms in `FEMObject`: regular and random. This triangulation will be used to create basis functions where overlapping functions will give me values for the stiffness matrix and load vector.

2.2.1 Regular mesh generation

Input values for the `regularTriangulation` method are: **r** Radius of the membrane. **n** Number of nodes in the first row of mesh. **m** Number of rows in mesh.

Distance between every row in mesh is equal to $x = r/m$. Points are added iteratively, staring from node in origin and then moving outwards row by row. First node in row has coordinates $(x, 0)$. To compute the rotation angle I used the formula $\alpha = i2\pi/n, \quad i = 0, ..., k-1$. k represents number of nodes in a given row, $k = j \cdot n \quad j = 1, 2, ..., m$. To complete each row I add k points with coordinates $R\binom{x}{0}$, where R is the rotation matrix.

In order to represent the rotation angle I have used `Angle` class found in GMLib. While rotation matrix R is represented by `SqMatrix` which when multiplied automatically works as a rotation matrix.

2.2.2 Random mesh generation

Input value for `randomTriangulation` method is **e_nodes** that represents number of nodes on the boundary. Starting from a regular mesh I have made an algorithm that randomly swaps internal nodes and then deletes 75% of nodes from the array. Resulting mesh makes it much easier to calculate the displacement of membrane, whilst preserving smoothness of the object that I get from regular mesh.

To perform swapping I used `uniform_int_distribution` defined in std. Then I calculate two values to limit how many times nodes will be swapped and how many nodes will be removed. It's worth noting that I use `removeBack` method, since it's most efficient removal method in `ArrayLX`. This method works best with meshes that have a lot of elements to begin with, meshes with relatively few (under 100) nodes end up looking too rough. However I don't see is as a problem, since in normal use membrane is triangulated by a lot of triangles.

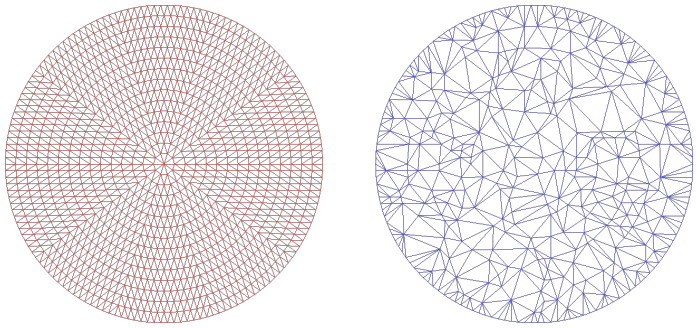

Figure 1: Triangulations made using regular and random meshes

2.3 Solving equation

Defining circular domain $\Omega \in \mathbb{R}^2$ with its boundary $\partial\Omega$ and applying force f on the domain, Poisson's equation describing it takes the following form.

Find u, which is the deformation level of circular membrane, such that

$$\begin{cases} \Delta u = f, \text{ in } \Omega, \\ u = 0, \text{ on } \partial\Omega. \end{cases} \tag{2}$$

where $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplace operator.

Using well defined theory for transforming strong formulation of PDE to weak formulation I got following equation:

$$\int_{\Omega} \Delta v \cdot \Delta u = \int_{\Omega} f(s) \cdot v \, ds \tag{3}$$

Assuming that Ω is partitioned by N nodes where only internal nodes are taken into consideration I can rewrite equation (3) as:

$$Ax = b \tag{4}$$

For brevity I have omitted showing all necessary calculations to arrive at this result. They can be found in lecture notes provided on Canvas. [1]

2.4 Calculation of stiffness matrix

2.4.1 Non-diagonal elements

Values for elements a_{ij} in stiffness matrix A are zero if there's no edge between them. If there exists an edge ij then the value of a_{ij} can be calculated using two triangles that have edge ij.

$$\begin{aligned} a_{ij} = &\left(\frac{-1}{|d|^2} + \frac{a_1 \cdot d}{|d \times a_1|}\left(1 - \frac{a_1 \cdot d}{|d \times a_1|}\right)\right)\frac{|a_1 \times d|}{2} \\ &+ \left(\frac{-1}{|d|^2} + \frac{a_2 \cdot d}{|d \times a_2|}\left(1 - \frac{a_2 \cdot d}{|d \times a_2|}\right)\right)\frac{|a_2 \times d|}{2} \end{aligned} \tag{5}$$

Where d is the vector between i and j, while a_1 and a_2 are vectors that go from i to vertices in the two triangles that lie on the intersection of two nodes and have ij as an edge.

To find vectors $\overrightarrow{a_1}$ and $\overrightarrow{a_2}$ in code I have set up my program to find points P_2 and P_3, then subtract P_0 from them to get vectors. I get triangle vertices by calling `triangle->getVertices()` method which returns array of `Vertex` objects. Then I go through vertex array and find one that doesn't have P_0 or P_1 as their parameter. The $\overrightarrow{d}$ is found by calling `edge->getVector()` method.

2.4.2 Diagonal elements

Values for diagonal elements are simply expressed as

$$a_{ii} = \sum_{k=1}^{\ell} T_k = \frac{d_3 \cdot d_3}{2|d_1 \times d_2|} \tag{6}$$

Where ℓ is the number of triangles that have corner in i, and d_1, d_2 and d_3 are vectors that form edges of triangle T_k.

To find these three vectors I get array of vertices from `triangle->getVertices()` and then order them so that first object in array is P_0, second is P_1 and last one P_2.

2.5 Calculation of load vector

The load vector b is assembled by following formula:

$$b_i = \int_K f v_i \, ds = f\frac{1}{3}\sum_{k=1}^{\ell} S_{T_k} \tag{7}$$

The function f is the specified load function. An integral of the basis function is equal to the volume of a pyramid based on the node with height equal to 1. An area of the base is equal to the sum of triangle areas $S_{T_k}, \quad k = 1, ..., \ell$. Areas of triangle are found using `triangle->getArea()` method.

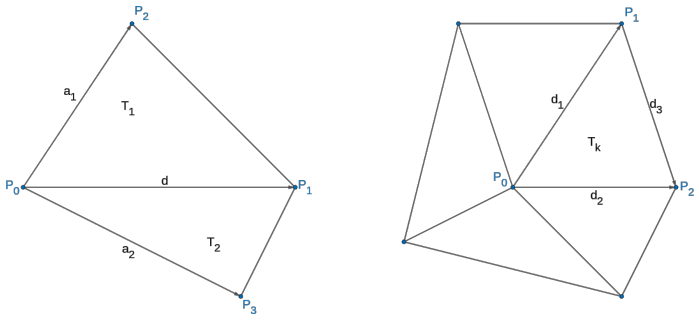

Figure 2: Diagrams to better explain how equations (5) to (7) were derived

2.6 Animation

Animating the solution involved adding some helper members and methods to `FEMObject` class and overriding `localSimulate` method. This method simply updates the value of f based on the dt, and updates position of internal nodes. dt represents timestep in animation.

Minor changes to `simulator.h` and `simulator.cpp` files were done to call `replot` on `FEMObject` objects in scene. This is simply a method that goes through vector of scene objects and calls `replot` on them. This method was connected in `application/guiapplication.cpp` so that it is called during simulation.

3 Results

Using the aforementioned methods I have been able to successfully implement a program that is able to simulate distortion of a circular membrane. Program can be programmed to use regular and random triangulation, change maximum force, and speed of animation.

There exists a bug in animation itself. When it is run at sufficiently high speed, speed factor set to be under 0.2, then the membrane might stop at one of the peaks and stop updating its position. I was unable to resolve this problem as I needed to focus on other projects and exams.

Video showing the animation can be found in the project directory or by following this link: https://youtu.be/wwwnGKdEI7A

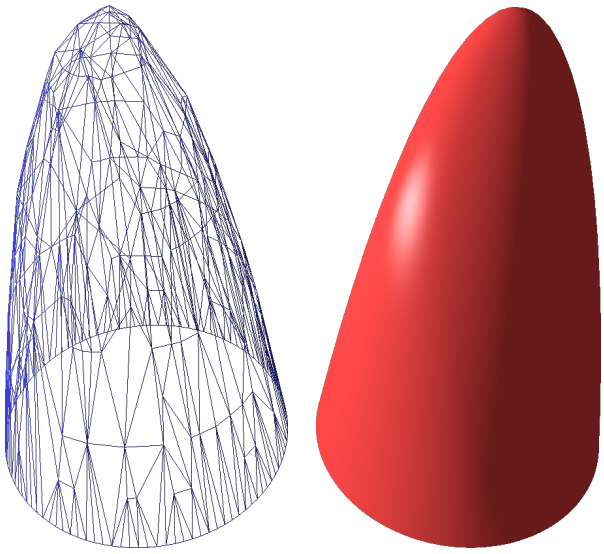

Figure 3: Distortion of membrane calculated using my program

4 Analysis & Discussion

After consultation with Tatiana Kravtec and Arne Lakså I am sure that my program functions correctly and finds the proper distortion for a given force. Furthermore the animation runs very smoothly provided I use random triangulation instead of regular one, as the number of necessary calculations for regular triangulation slows down the program too much.

Even though I haven't run benchmarks on my code, it's clear that the program runs fast. The big bottleneck is visualization of results in 3D. This claim based on a test run on a regularly triangulated membrane with 15 nodes in first row and 20 rows. Calculation itself took around 5 seconds on my computer while it took 15 minutes to show the result on my computer.

Using an iterative method for inverting and multiplying matrices would cut down on calculation time greatly. All of the professional FEM products implement that, which cuts down on calculation time greatly. Seeing how a lot of element in A are zero using a textbook version of matrix inversion takes too much time on a huge matrix. In my opinion this is the main bottleneck in my program.

Further development can be done. It would be very good to implement a way to change triangulation method on the fly or change value for max force or animation speed in the GUI, instead of hard-coding these values in program. I should also fix animation error that was mentioned in the results section. Another improvement would be to have a function that deletes nodes that are very close to each other to cut down on calculation time. Another function that if there are too few nodes in an area and fixing that would be a good choice too.

5 Bibliography

[1] Tatiana Kravetc. Finite element method programming. February 2018.