Report: Finite Element Method programming

Shvetsov Nikita

February 26, 2016

Abstract

In this report the finite element solver program is presented. The main aim is to create a program that will construct an object - a circular drum membrane. Plotted points of the membrane are acquired by 2 different methods - random and regular, after, the program simulates deformation of the drum membrane, depending on the force applied. The algorithm is designed with the solution given by finite element method.

1 Introduction

The purpose of this project is to simulate the deformation of a circular drum membrane. No specific material properties defined. Pressure, given to each point is equal. Pressure on boundary is equal to zero. The problem is solved by finite element method (numerical method used for solving differential equations). This method was implemented in C++ program using framework Qt, GMlib library and class TriangleFacets.

2 Theoretical description

Differential equation of this problem:

$$\begin{cases} \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = f(x) \\ u = 0 \end{cases}$$

$x = (x_1, x_2) \in \Omega$ on $\partial\Omega$, where Ω is a surface.

Weak formulation:

$$a(u, v) = L(v),$$

where $a(u, v) = \int_\Omega grad\ u\ grad\ v\ dx$

$$L(v) = \int_\Omega f(x) v dx$$

To solve this problem we need to find a solution to the matrix equation: $Ax = b$, where A is the stiffness matrix, where $A[i][j] = a(v_i, v_j)$ (v_i are the vertexes that are not on the boundary, or basis functions), and b is load vector with components being numerical values of the volume of the pyramids, which are limited to the plane xOy and the basis function v_i [1]. Number of the basis functions will be equal to number of vertexes that are not on boundary.

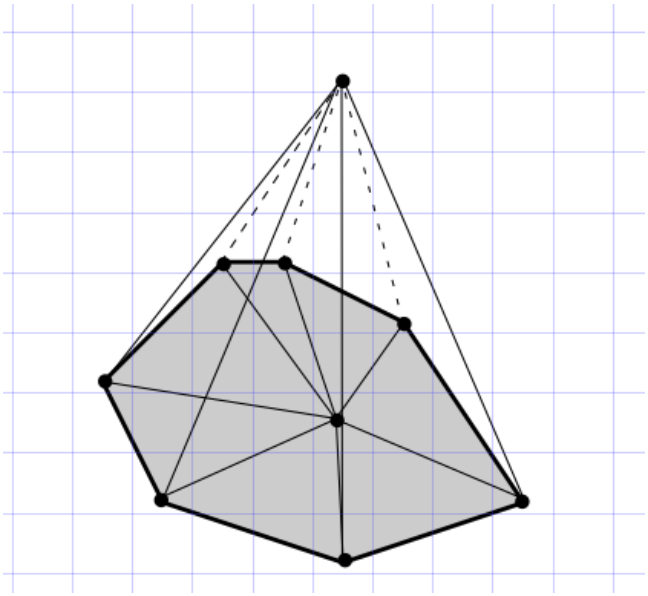

Figure 1: Basis function

3 Triangulation process

In the program class Node was used to describe an inner vertex and adjacent elements such as triangles and edges. Class Femobject contains all computations, including partitioning and triangulation processes. To prepare the circular surface for the triangulation process we need to do the partitioning first. There are two ways of partitioning: regular and random.

3.1 Regular triangulation

For this type of triangulation we use 3 parameters: number of circles in our object, number of nodes on the first circle and the radius of this circle. These parameters help us to define the rotation vector, used for getting points around each circle. First we add initial $(0, 0)$ point to the array of vertexes, it is our starting point. In each circle we redefine rotation vector (vector between 2

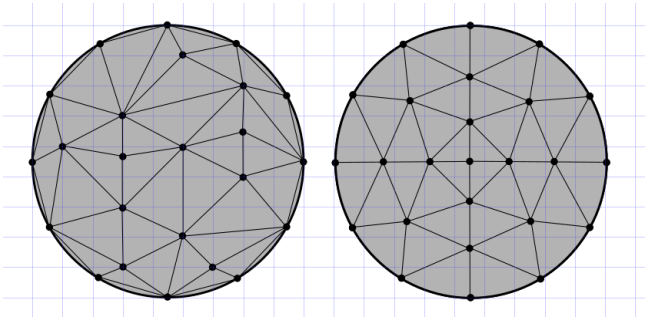

Figure 2: Result of random and regular triangulations

points - starting point and point, defined by radius of the circle) and rotation angle($2\Pi/n$, where n - number of nodes on the circle). Each found point we add to the array of vertexes. After all circles are dealt with, we use method from TriangleFacets - triangulateDelaunay().

```
void Femobject::regularTriangulation(float small_radius,
    int rings, int innerNodes) //building regular points
    for triangulation
  {
      //adding initial (0,0) point
      auto p0 = GMlib::Point<float,2>(0.0f, 0.0f);
      this->insertAlways(GMlib::TSVertex<float> (p0));
      for (int i=1; i<=rings; i++)
      {
          GMlib::Vector<float,2> rotVect(float(i)*
              small_radius,0);
          GMlib::Angle a = M_2PI/(innerNodes*i);
          GMlib::SqMatrix<float,2> matrRot (a, GMlib::
              Vector<float,2> (1,0), GMlib::Vector<float
              ,2> (0,1));
          for (int j=1; j<=(innerNodes*i); j++)
          {
              rotVect =  matrRot * rotVect;
              this->insertAlways(GMlib::TSVertex<float> (
                  static_cast<GMlib::Point<float,2>>(
                  rotVect)));
          }
      }
      this->triangulateDelaunay();
  }
```

3.2 Random triangulation

For random triangulation we use 2 parameters: radius of the circle and number of triangles, we want to see in our circle membrane. With these parameters we can find number of border nodes, inner nodes and length of the side of a triangle. Then we add all border points the same way as we did in regular, find new radius to generate random points, do checks to know that the point is valid and add the found point to the array. Just like in regular we use triangulateDelaunay() after we achieved the number of desired points.

4 Calculation of the stiffness matrix A and load vector b

4.1 Stiffness matrix A

To calculate stiffness matrix A we need an array of inner points. After we divide them from boundary points we prepare our matrix for computations. $A[i][j] = a(v_i, v_j)$ where i and j are the index numbers of nodes in array. Hence, the dimension of the matrix A is $n \times n$, where n - number of nodes in array. To compute the matrix A we need to consider 3 cases. When two different nodes do not have common edge, then the corresponding value in A matrix $A[i][j]$ will be equal to zero, due to they have no common triangles area intersection. Another case is elements outside the diagonal, i.e. elements with $i \neq j$ in matrix A. They have a common edge and intersection, which is 2 triangles [2].

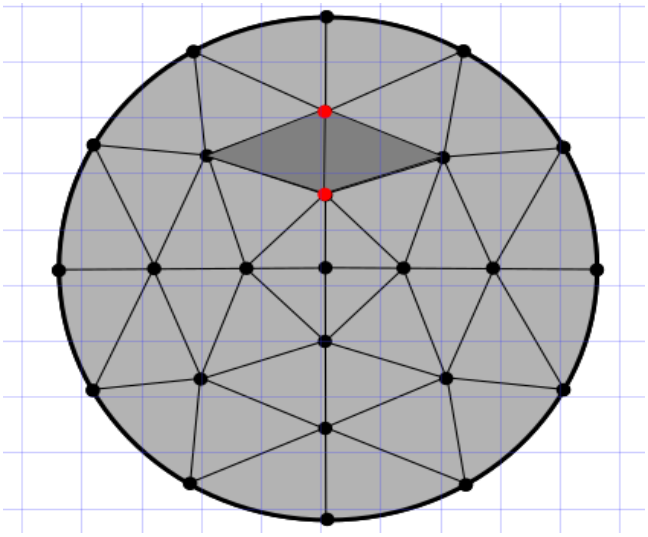

Figure 3: Adjacent nodes and intersected area

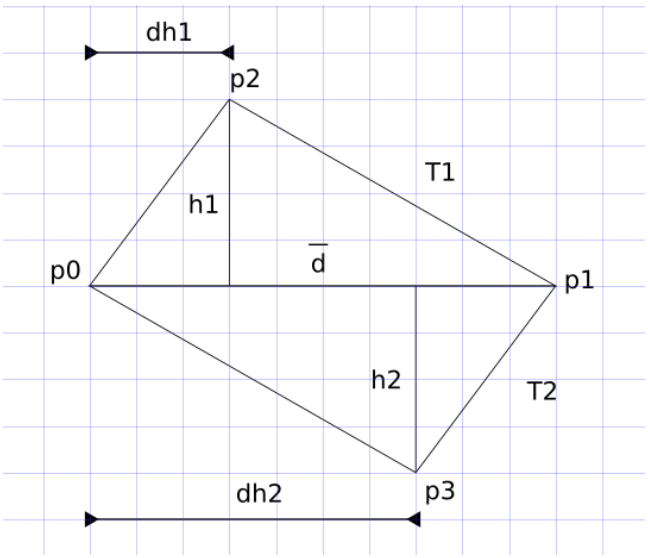

Figure 4: Triangles of the intersected area

$$a(u,v) = \int_\Omega gradugradvdx = \int_\Omega < d\Phi_i, d\Phi_j > \partial\Omega = \int_{T_1} < d\Phi_i, d\Phi_j > dT_1 + \int_{T_2} < d\Phi_i, d\Phi_j > dT_2$$

$$= \int_{T_1} \left(-\frac{1}{|d|} \quad -\frac{<a_1,d>}{|d\hat{}a_1|}\right) \begin{pmatrix} -\frac{1}{|d|} \\ 1 - \frac{<a_1,d>}{|d\hat{}a_1|} \end{pmatrix} dT_1 + \int_{T_2} \left(-\frac{1}{|d|} \quad -\frac{<a_2,d>}{|d\hat{}a_2|}\right) \begin{pmatrix} -\frac{1}{|d|} \\ 1 - \frac{<a_2,d>}{|d\hat{}a_2|} \end{pmatrix} dT_2 =$$

$$= \left(-\frac{1}{|d|^2} + \frac{<a_1,d>}{|d\hat{}a_1||d|}\left(1 - \frac{<a_1,d>}{|d\hat{}a_1||d|}\right)\right)\frac{area_1}{2} + \left(-\frac{1}{|d|^2} + \frac{<a_2,d>}{|d\hat{}a_2||d|}\left(1 - \frac{<a_2,d>}{|d\hat{}a_2||d|}\right)\right)\frac{area_2}{2},$$

where $area_1 =| a_1\hat{}d |$and $area_2 =| a_2\hat{}d |$ - area of parallelograms, which create the appropriate vector. Dividing these areas by 2, we find areas of rectangles in intersection.

In code:

```
auto colVector = findVectors(commEdg); //find 3 vectors
    with common edge
double dd = 1/(colVector[0]*colVector[0]);
double dh1 = dd * (colVector[1]*colVector[0]);
double area1 = std::abs(colVector[0]^colVector[1]);
double h1 = dd * area1 * area1;

double dh2 = dd * (colVector[2]*colVector[0]);
double area2 = std::abs(colVector[0]^colVector[2]);
double h2 = dd * area2 * area2;
_A[i][j] = _A[j][i] = ((((dh1 * (1 - dh1))/h1) - dd) *
```

```
(area1)/2) + ((((dh2 * (1 - dh2))/h2) - dd) * (area2)/2);
```

Last case is diagonal elements, when i=j in stiffness matrix A.

$A[i][i] = \int_\Omega < d\Phi_i, d\Phi_i > \partial\Omega = \sum_{k=1}^{l} \int_{T_k} < d\Phi_i, d\Phi_i > dT_k = \sum_{k=1}^{l} T_k$

$$T_k = \frac{< d_3, d_3 >}{2 \mid d_1 \hat{} d_2 \mid}$$

where l-number of the triangles in the node.

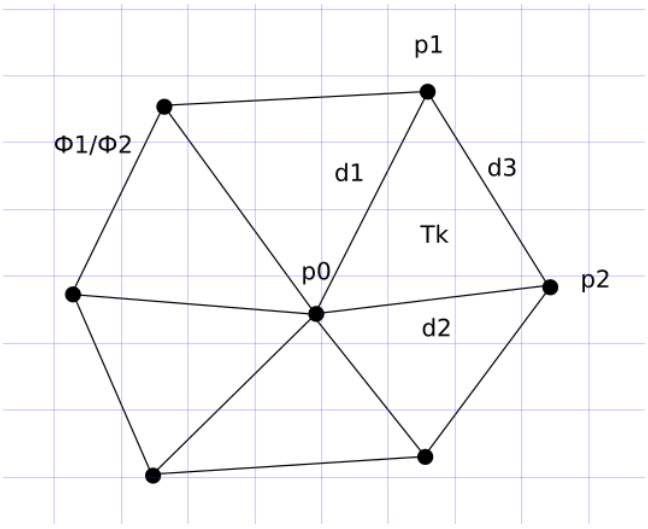

Figure 5: Handling intersection of node with itself

In code:

```
GMlib::Array<GMlib::TSTriangle<float>*> triangles =
    _nodes[i].getTriangles();
        for (int k=0; k<triangles.size();k++)
        {
            //find 3 vectors with triangle and center
                node
            auto colVector = findVectors(triangles[k],
                &_nodes[i]);
            GMlib::Vector<float,2> d1,d2,d3;
            d1 = colVector[0];
            d2 = colVector[1];
            d3 = colVector[2];
            _A[i][i] += (d3*d3)/(2*(std::abs(d1^d2)));
        }
```

4.2 Load vector b

The elements of vector b are volumes of the pyramids.

$$b = \begin{pmatrix} L(v_1) \\ \vdots \\ L(v_n) \end{pmatrix}$$

$$L(v_i) = \int_\Omega v_i dx = V_{pyramid} = \frac{1}{3} \cdot S_{base} \cdot h = \frac{1}{3} \cdot S_{base} \cdot 1$$

$$S_{base} = \sum_{k=1}^{l} S_{triangle_k}$$

where n - number of vertexes, l - number of triangles in the node. Height of the pyramid is equal to one, because $v_i$is the basis function.

```
for (int i = 0; i<_nodes.size(); i++)
      {
            _b[i]=0;
            auto triangles = _nodes[i].getTriangles();
            for (int j=0; j<triangles.size();j++)
            {
                  _b[i]+=triangles[j]->getArea2D()/3;
            }
      }
```

5 Z-component of the points

After inverting stiffness matrix A and multiplying with load vector b we get values of X on which we have to update the Z-coordinate of all inner nodes. Also we multiply vector b with desired force to simulate it further. Method updateFem:

```
void Femobject::updateFem(double force) //setting Z(
    height) value for every node using force value
  {
       GMlib::DVector<float> x = _A*(force*_b);
       for (int i=0; i<_nodes.size();i++)
       {
           _nodes[i].setZ(x[i]);
       }
  }
```

6 Simulation

In localSimulate method of our Femobject class we constantly change the force, from minimum to maximum values, and update our object using time dt.

```
void Femobject::localSimulate(double dt) //simulating of
    the calculated model
  {
      //managing _force direction
      if(_goUp) _force += 2*dt;
      if (!_goUp) _force -= 2*dt;
      if (_force > _maxForce) _goUp = false;
      if (_force < -1*(_maxForce)) _goUp = true;
      updateFem(_force);
      this->replot();//updating screen model
  }
```

Resulting simulation in program:

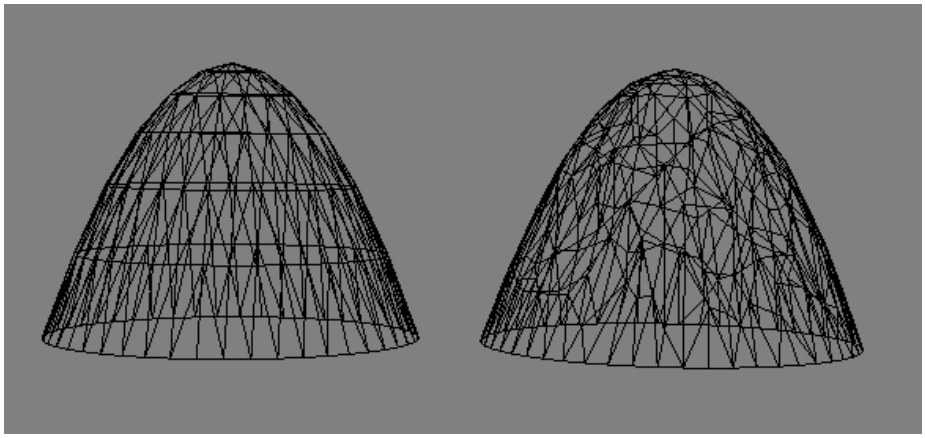

Figure 6: Resulting simulation

7 Conclusion

This project was quite interesting and challenging. The mathematical model of moving drum membrane was implemented using finite element method and then programed with C++, Qt and GMlib. Disassembling the process was informative and helped me to understand the physical side of surface deformation better.

References

[1] Dag Lukkaseen, *A short introduction to Sobolev-spaces and applications for engineering students*. November, 2004.

[2] NUC, *STE6291 FEM solver programming*. 2016.