C++ for Scientific Computing:
MassStiff FEM Solver
Candidate Number: xxxxxxxx

1 Introduction

The finite element method (FEM) is a method for solving differential equations that is of fundamental importance in the scientific community. It has been used on a daily basis for many years to ensure the structural integrity of bridges, aeroplane wings and buildings. The power of the finite element method lies in the endless geometries it can be applied to. In this project we apply techniques from computational geometry and finite element theory to build a FEM solver from the ground up in C++. The shorthand name for the FEM solver presented is MassStiff. We will assume throughout that the reader has some basic C++ knowledge. In the interest of maintaining the page limit of the report, we do not discuss instances where the user behaves or acts maliciously. However, we do make use of `throw-exception` in the actual code to handle such behaviour.

2 C++

2.1 Classes and Objects

C++ is a hybrid object-oriented/functional programming language. Object-oriented programming is discussed in great detail in [1] and functional programming in [2]. Object-oriented programming (OOP) is what makes C++ so powerful, and to highlight this, we will use an example. Consider a car. It is certainly an *object* and we classify cars in terms of their attributes: If it has a *big engine* and an *aerodynamic exterior*, it is a sports car. Also, cars have functions or *methods*: A car can turn corners, carry passengers etc. In an abstract sense, we have described a data type. In fact, using this information we can describe what a car is to our program using a C++ *class*:

```
1  Class Car
2  {
3  private:
4          Engine eng;
5          Bodykit bdykit;
6          Passenger* passngrArray;
7          int IncreaseGear();     // Increases the gear.
8          ...
9  public:
10         Car();                  // Constructor.
11         int Start();            // Starts the engine.
12         int PassengerCount();   // Returns the number of passengers.
13         ...
14 };
```

This is referred to as a class *declaration*, and a class *definition* will usually follow. Class definitions are usually placed in a separate file and contain implementations of the functions contained in the class declaration. Notice that there are two keywords in the listing above, `public` and `private`. Variables and methods under the `public` keyword are allowed to be called by the user of the class while those under the `private` keyword are only allowed to be called by the member functions of the class.[1] This feature is an attempt to abstract the user from the inner workings of an object.[2] There are two features hidden in the above listing: *Aggregation* and *Composition*. In rigorous OOP theory a distinction is made between these concepts; see [1] for more detail on this. Throughout this report we will not make any distinction. As we can see from the above code listing, a car is a composition of many objects: An engine, four wheels, a transmission, a differential, etc. Each of these objects would also have a class declaration (and definition). To highlight the power of the OOP paradigm, the following code listing produces a similar image to that in Figure 6:

```
#include<iostream>
#include"FEMDriver.h"
int main(){
        FEMDriver femd; /* Instantiates a FEMDriver object */

        //Reads the vertices of the domain from a file passed in as an argument.
        femd.mesh.readVertices("STAR.txt"); femd.mesh.triangulate();
        femd.mesh.refine(1);femd.mesh.improve(); /* Obtain initial solution */
        femd.pfem.init(femd.mesh); /* Obtain initial solution */

        //Adapts mesh to the solution and recalculates solution.
        femd.mesh.adapt(femd.pfem.getSolution(),0.1);femd.pfem.init(femd.mesh);
}
```

This code solves the Helmholtz equation on a geometry described by the vertices contained in the file `STAR.txt`, the point being that we have thereby abstracted the user from the details of the finite element method, from the triangulation of the domain and from the display of the solution of the problem. Everything we want to achieve is mediated by the `FEMDriver` object. If we want to triangulate the mesh, we write `femd.mesh.Triangulate()`, which triangulates the mesh associated to our finite element problem. If instead we want to access some part of the finite element code, i.e. the solution, then we write `femd.pfem.getSolution()`.

[1] We use *member functions* and *methods* interchangeably.

[2] Abstraction is one of the core principles behind object oriented programming. To solidify this idea, consider driving a car. Do we need to know how the internal combustion engine works on our drive to work?

2.2 Inheritance

Inheritance is a feature of object oriented programming that encourages code reusability. In the previous section we briefly described a car class. A more generic approach would be to write a *vehicle* class and then to define a derived class *car* that inherits all of the features of *vehicle* but has extra features, such as four wheels. For example:

```
1  //Base class
2  Class Vehicle
3  {
4  protected:
5          int availableSpacesForPassengers;
6          ...
7  };
8  //Derived class
9  Class Car : public Vehicle
10 {
11 ...      //Features associated with a car.
12 };
```

This listing introduces us to another keyword: `protected`. Anything defined as protected can be used by derived classes but is essentially private outside the class tree. For more information, see [3].

3 MassStiff Structure

Figure 1 shows us the structure of MassStiff. Each box represents a class and each arrow indicates a relationship. A red arrow indicates inheritance and a black arrow indicates aggregation. Throughout this report we are going to concentrate on the objects with bright pink text and, due to the length restrictions of the report, only the class declarations will be included. However, we will include the implementation of certain interesting functions. We will now outline the functionality of the yellow classes:

- The `Matrix` class contains basic matrix functionality such as addition and multiplication as well as more complex functionality such as an implementation of GMRES, CG and partial pivoted Gaussian elimination. We will use these methods throughout the report to solve the matrices that we encounter. The `Vector` class is very similar.

- `ANTMATH` provides the data structures `Vertex`, `Edge` and `AV3INT` (used to repre-

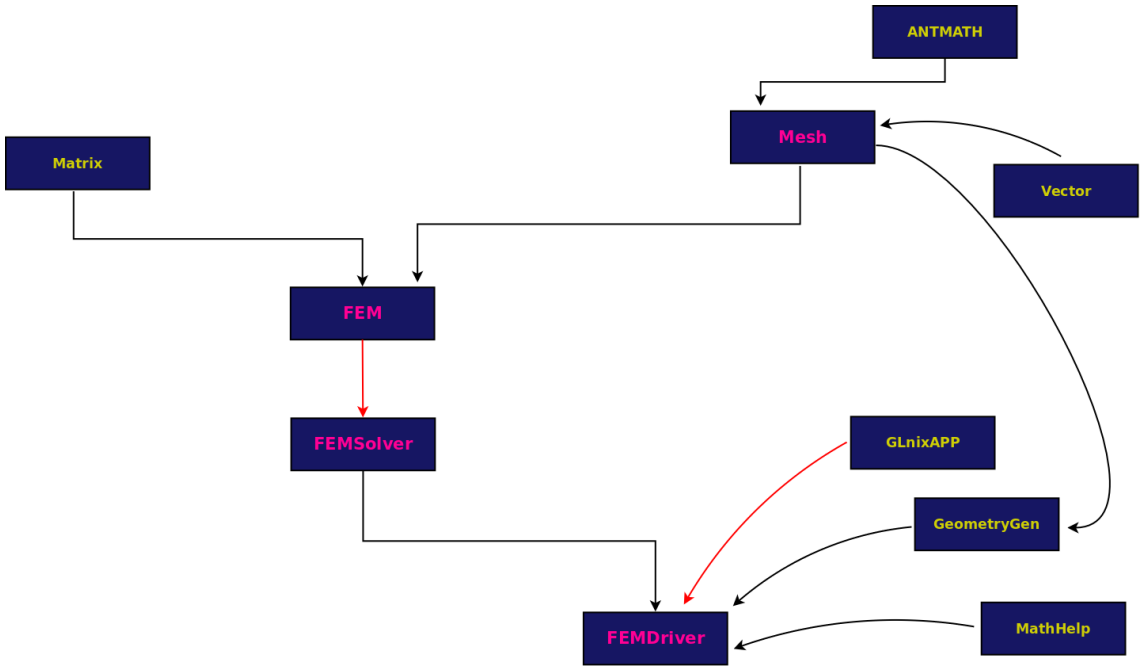

Figure 1: Structure of MassStiff.

sent a triangle).[3]

- `GLnixAPP` is a computer graphics framework that interfaces with X11 and OpenGL to obtain an OpenGL context, that is, a window that we can display 3D geometry in. This contains many `virtual` methods which are to be redefined in its child class `FEMDriver`, such as `onMouseDown` which interprets mouse movement.[4]
- `GeometryGen` is used to convert the output of the FEM solver in such a way that the display routine can use it.
- `MathHelp` provides functionality that is used in controlling the mouse input.

All code apart from the `Vector` class was written by the author.

3.1 Mesh

We describe a mesh as a collection of vertices, edges and triangles. Hence a mesh is an aggregate object: It is a collection of other objects. In order to form this collection we first have to describe to the compiler what these objects are. Vertices and edges are defined in a file called `ANTMATH.h`, and in order for us to gain access to them,

[3] `ANoTherMATHslibrary`.

[4] `openGLu-nix-APPlicationlibrary`.

we must enter the line of code `#include"ANTMATH.h"`. The user provides the mesh object with a list of vertices in the form of a `.txt` file that describes a polygonal boundary. The method `triangulate()` then forms a parental triangulation of the described polygon.

```
#ifndef MESH_H
#define MESH_H
#include<vector>
#include"ANTMATH.h"
#include"Vector.h"
// Mesh class
class Mesh
{
public:
    std::vector<Vertex> mVertices;
    std::vector<AV3INT> mTriangles;
    std::vector<Edge> mEdges;
    std::vector<int> mTriSwapInds;
    double mArea;
private:
    // Utility functions for Triangulate()
    int isEdgeIntersect(size_t n, const std::vector<Vertex> &p);
    int isVertexInterior(size_t n, const std::vector<Vertex> &p);
    int isEar(size_t n, const std::vector<Vertex> &p);

    // Area of triangle.
    double areaOfTriangle(AV3INT& triangle);

    // Calculates the smallest angle contained in triangle.
    double smallestAngle(AV3INT& triangle);

    // Corrects the orientation of triangle.
    void fixTri(AV3INT& triangle);

    // Selects candidates for edge swapping.
    void scanTriangles();
public:
    void adapt(Vector&solution,double thres);
    Mesh adaptNew(Vector&solution,double thres,Vector&oldsol,Vector&oldvel);
    void improve();
    void readVertices(const char* filename);
    void printVertices();
    void triangulate();
    void refine(int refineCount);
};
#endif
```

Triangulating an arbitrary polygon is quite a complicated process and so we will not discuss that part of the implementation here and instead refer to [3], which follows our implementation quite closely. In fact, our implementation extends on that of [3], as `Mesh` can deal with domains with holes, carry out edge-flipping, refine triangulations

and adapt the mesh to a solution (see Figure 2 below and Figures 9 and 10 in Appendix B). A new data type is introduced in the above listing. `std::vector<template>` is a container, which is taken from the Standard Template Library (STL). The STL is a collection of containers and algorithms (which operate on containers). An STL vector is different to a "mathematical" vector in many ways. For example, when we consider a vector $v \in \mathbb{R}^n$ we do not think of it as an object that can shrink, or even increase, in dimension. An `std::vector`, on the other hand, can increase or decrease in size.

3.1.1 void refine(int refineCount)

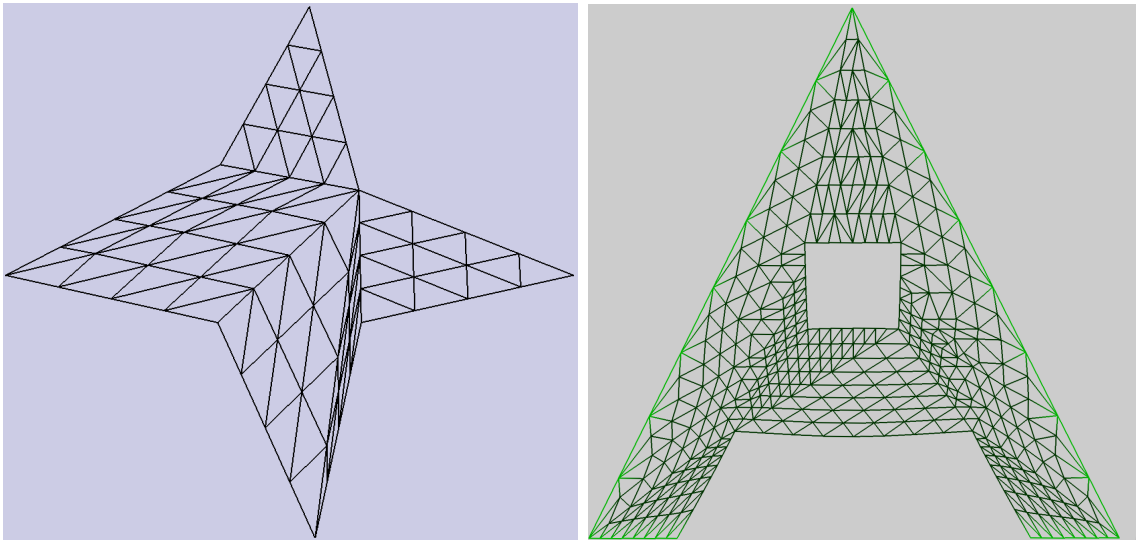

Figure 2: (Left) Refinement of domain after triangulation. The original triangulation is clearly distinguishable and results in groups of tightly bunched triangles. (Right) A triangulation of a polygon with a hole. The green boundary highlights that points along it satisfy a Dirichlet boundary condition.

When we call the refinement method we split each triangle into four smaller triangles. To do this, we select a triangle from the existing triangles, which are either formed by `triangulate()` or by a previous call to `refine()`. We then calculate the midpoints of each of the edges of the triangle and form four smaller triangles which are then appended to a local buffer. To see this graphically, see Figures 2 and 3. Considering that this triangulation routine is going to be used to solve FEM problems, we have to make sure that the algorithm maintains the boundary flags correctly. Another point to mention is that when we solve a FEM problem we are solving a system

whose size is dependent on the number of nodes under consideration, and if we have repeated nodes this will lead to a bottleneck for large problems. In the subdivision process we are going to obtain repeated vertices and so the refinement routine must deal with this.

```
void Mesh::refine(int refineCount)
{
    // Calculation of how many elements we are going
    // to have after triangulation.
    int nElements = mTriangles.size();
    int nNewElements = nElements*pow(4,refineCount);
    int nVerts = mVertices.size();
    // Local buffers.
    std::vector<Vertex> lVertices;
    std::vector<AV3INT> lTriangles;
    std::vector<Edge> lEdges;
    for(int j=0;j<refineCount;j++)
    {
        // Save a copy of the input geometry.
        lVertices = mVertices; lTriangles = mTriangles; lEdges = mEdges;
        mTriangles.clear(); mVertices.clear(); mEdges.clear();

        UINT numTris = lTriangles.size();
        for(UINT i = 0; i < numTris; ++i)
        {
                Vertex v0 = lVertices[ lTriangles[i][0] ];
                Vertex v1 = lVertices[ lTriangles[i][1] ];
                Vertex v2 = lVertices[ lTriangles[i][2] ];

                // Generate the midpoints and new edges.

                Vertex m0, m1, m2;
                Edge e0,e1,e2,e3,e4,e5,e6,e7,e8;

                m0.Position = AV2FLOAT(
                        0.5f*(v0.Position.x + v1.Position.x),
                        0.5f*(v0.Position.y + v1.Position.y));
                m0.VertexNumber = nVerts++;

                // Here we use the edges to correctly allocate
                // interior nodes and to check if the segment v0v1
                // or the segment v1v0 is a diagonal.

                Edge isD(v0.VertexNumber,v1.VertexNumber);
                auto fit = std::find(lEdges.begin(),lEdges.end(),isD);
                if(fit != lEdges.end()&& fit->boundary ==1 ){m0.Boundary = 1;}
                else{m0.Boundary = 0;}
                e0.setEndPoints(v0.VertexNumber,m0.VertexNumber);
                e0.boundary=m0.Boundary;
                e1.setEndPoints(m0.VertexNumber,v1.VertexNumber);
                e1.boundary=m0.Boundary;

                // The same is done for m1 and m0.
```

```
49
50                  // Add new geometry: "push back" local buffers.
51          }
52
53      }
54      // Clear the temp buffers.
55      lTriangles.clear();
56      lVertices.clear();
57
58      // After our refinement routine we are going to have
59      // repeated vertices. In order to solve this we  carry out
60      // a routine that creates a container of unique vertices.
61
62      int newnode;
63      // For each element.
64      for(int k=1;k<nNewElements;k++)
65      {
66          // For each vertex.
67          for(int l=0;l<3;l++)
68          {
69          newnode = 0;
70          for(int i=0;i<nn+1;i++)
71          {
72          // Is the vertex we are looking at close
73          // to any of the vertices we have already
74          // looked at?
75              if(mVertices[mTriangles[k][l]].distance(lVertices[i])<eps)
76              {
77                  newnode = 1;
78                  // Replace the old vertex number with a new one
79                  // Append triangle/Correct vertex.
80              }
81          }
82          if(newnode==0)
83          {
84                  // Append the vertex container.
85                  // Append triangle/Correct vertex.
86          }
87      }
88      }
89      // Update boundary data.
90      // Update our member containers.
91  }
```

3.1.2 void improve()

We do not want the FEM mesh to have many long thin triangles. In fact, we would like most of our triangles to be the same size. If we want a finer mesh in certain areas we can then call an adaptivity routine. In order to remedy a bad mesh we seek out triangles with a very small internal angle. This is carried out by `scanTriangles()`. After this routine has completed we have a container of integers `mTriSwapInds` which

contains the indices of the triangles to be swapped, if there are any. We find a triangle in the mesh that shares an edge with the chosen triangle and form the triangles that will result after edge swapping. The swap is carried out if there is an increase in triangle quality.[5] We can see the result of running this routine by first looking at the left image in Figure 2 and then the left image in Figure 4. The long strip of thin triangles has almost dissipated and we are left with a more uniform mesh.

```
1  void Mesh::improve()
2  {
3      // Look for bad triangles.
4      scanTriangles();
5      std::vector<int>::iterator it = mTriSwapInds.begin();
6      // Iterate through the indices of the triangles to be swapped.
7      while(it!=mTriSwapInds.end())
8      {
9          int k = *it;
10         it++;
11         AV3INT triangle1 = mTriangles[k];
12         Edge edgearr[3];
13         // Fill edge array: edgearr.
14         for(int i=0;i<mTriangles.size();i++)
15         {
16             if(i==k) continue;
17             AV3INT triangle2 = mTriangles[i];
18             Edge edges[3];
19             // Fill edge array: edges.
20             for(int j = 0;j<3;j++)
21             {
22                 for(int l=0;l<3;l++)
23                 {
24                     if(/* edges are equal and have opposite orientation */)
25                     {
26                     // "Pre-swap" triangles to obtain
27                     // forecast area and smallest angle
28                     // values
29                        if(/* We are in a degenerate case */) continue;
30                        if(/* Increase in triangle quality */)
31                        {
32                                 // Actually carry out edge flip
33                        }
34                     }
35                 }
36             }
37         }
38
```

[5]This is purposely vague. One of the best measures we tested for triangle quality was to calculate the average of the minimum angles of the triangles to be swapped, and then calculate the average of the minimum angles of the forecasted edge-swapped triangles. If the average of the minimum angles saw an increase, the swap would actually be carried out. For an illustration, see Figure 3.

```
    }
}
```

3.1.3 void adapt(Vector & solution, double thres)

Adaptivity is fundamental to modern finite element methods, it allows us to refine the mesh where it is most needed. This process can be applied as many times as the user wants to, which results in the patches of highly refined mesh. We check the solution at the vertices of each of the triangles. If the value of the solution differs by an amount greater than `thres` then we refine that triangle only. The result of adaptivity can been be seen in Figure 4.

```
void Mesh::adapt(Vector&solution, double thres)
{
    int vertNum = mVertices.size(), trilen = mTriangles.size();
    std::vector<Vertex> toBeAdded;
    for(int i=0;i<trilen;i++)
    {
        AV3INT triangle = mTriangles[i];
        AV3INT tri1,tri2,tri3,tri4;
        std::vector<double> solutionTri;
        solutionTri.push_back(solution(triangle.x));
        solutionTri.push_back(solution(triangle.y));
        solutionTri.push_back(solution(triangle.z));
        std::sort(solutionTri.begin(),solutionTri.end());
        if(fabs(solutionTri[0]-solutionTri[2])>thres)
        {
           Vertex v0 = mVertices[ triangle[0] ];
           Vertex v1 = mVertices[ triangle[1] ];
           Vertex v2 = mVertices[ triangle[2] ];
           Vertex m0, m1, m2;

           // CALCULATION OF MID-POINTS m0,m1,m2

           // Append flags
           int m0add=1;int m1add=1;int m2add=1;
           int size = toBeAdded.size();
           if(size>0)
           {
                for(int i =0;i<size;i++)
                {
                // Check if any of the mid-points are scheduled
                // to be added to mVertices. If so then
                // we just need to take note of the vertex number

                // If not then we leave the append flags as they are
                }
           }
           if(m0add) {m0.VertexNumber=vertNum++;toBeAdded.push_back(m0);}
           // The same is done for the m1, m2

```

```
40              // Set vertex numbers of triangles to be added and add new geometry
41          }
42      }
43      // Add temporary vertex container to Mesh's member vertex container
44  }
```

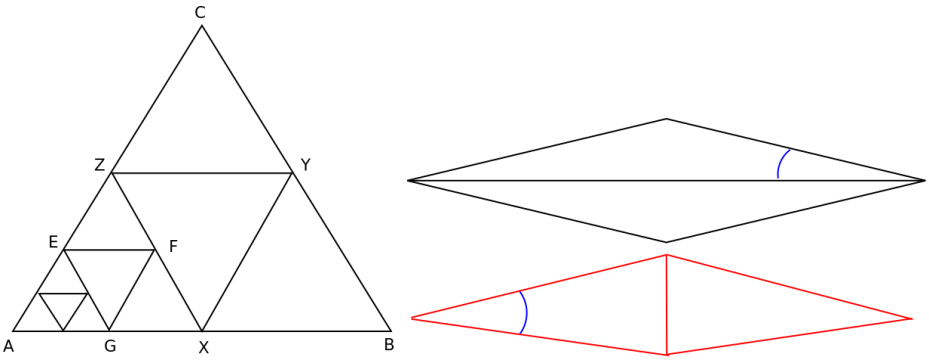

Figure 3: (Left) Refinement and Adaptation. Before refinement we start with the triangle ABC, after refinement we are left with the triangles AXZ, XBY, ZYC, XYZ. In this graphic we also show adaptation. We test the value of the current solution at each of the vertices A, B, C, X, Y, Z and if the difference between the largest value and the smallest value on that triangle is greater than some threshold, we subdivide that triangle into four smaller triangles. (Right) An illustration of what `improve()` does to each triangle.

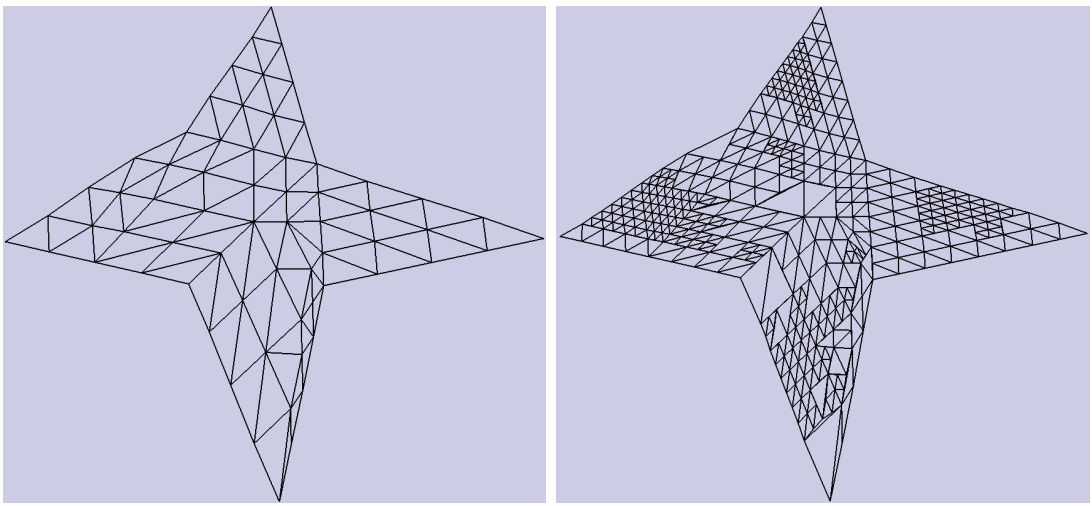

Figure 4: (Left) Refinement with improvement. (Right) Refinement with improvement followed by adaptation to the solution of the Helmholtz equation.

3.2 FEM

Our finite element class exists to provide helper functions to its derived classes. We endow the class with protected functions to form the elemental mass and elemental stiffness matrices. So long as we are using linear basis functions, the elemental mass and stiffness matrices will be the same for all dimensions.

```
#ifndef FEM_H
#define FEM_H
#include"Mesh.h"
#include "Matrix.hpp"
class FEM
{
protected:
    // Generates elemental mass matrix.
    Matrix elementMass(double x1,double x2,double x3,double y1,double y2,double y3);
    // Generates elemental stiffness matrix.
    Matrix elementStiff(double x1,double x2,double x3,double y1,double y2,double y3);
    int nodeCount; /* number of nodes */
    int elementCount; /* number of triangles or elements */
    Matrix mGlobalMatrix;
    Matrix mGlobalMass;
    Matrix mGlobalStiff;
    Vector U; /* Solution */
    Vector b; /* Right-hand-side */
};
#endif
```

3.3 FEMSolver: Wave Equation

The class `FEMSolver` inherits from `FEM` all its basic features, but is a lot less cluttered with basic functionality. The user of this class would be encouraged to edit the member functions `solve()`, `getSolution()` and `initialSetup(Mesh&mesh)`. When we look at the class `FEMDriver` we will see that `getSolution()` is periodically called to retrieve a new solution. For time-independent problems, like the Helmholtz equation, this function would be empty. `Solve()` fills the solution vector with solutions at all time steps. This function would also be empty for time-independent problems. The function `initialSetup` is more versatile: It can either be used to fill an initial condition or to solve the system. The `FEMSolver` implementation given in this report adds further functions to make use of the mesh class adaptivity features. It is a nice feature that the amount of user code matches that of MATLAB. In the following implementation of `FEMSolver` we solve the wave equation using the Crank-Nicholson

scheme. That is,

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u, \qquad u \in \Omega, \tag{1}$$

$$\frac{\partial u}{\partial n} = 0, \qquad u \in \partial\Omega. \tag{2}$$

To get from these equations to the sequence of matrix equations for the implementation is quite an involved process and will be included in Appendix A. Our choice of domain and initial condition is largely irrelevant as our triangulation routine is fairly robust. We choose the domains given in Figure 4 and on the title page of this report.

```
#ifndef FEMSOLVE_H
#define FEMSOLVE_H
#include"FEM.h"
class FEMSolver : public FEM
{
private:
    Vector V;
    double mTheta;              /* Used in Crank-Nicholson. */
    double mAlpha;              /* Parameter for Wave Equation. */
    double mIntervalEnd;        /* End of interval. */
    double dt;                  /* Time interval between time steps. */

    // The user is to edit this function.
    void initialSetup(Mesh&mesh);
public:
    std::vector<Vector> mSolutions;
    std::vector<Vector> mVelocities;
    std::vector<Vector> mAdaptedSolutions;
    std::vector<Vector> mAdaptedVelocities;
    std::vector<Mesh> mAdaptedMeshs;
    int mNumberOfTimeSteps;
    int mTimestep;

    // Methods that the user is to edit.
    void init(Mesh&mesh);
    void solve(Mesh&mesh);
    Vector getSolution();

    // Supplementary functions to deal with adaptivity.
    void adaptMeshToSolutions(Mesh&mesh);
    Mesh &getAdaptMesh();
    Vector getAdaptSolution();
};
#endif
```

The implementation (or class definition) is given as:

```
#include"FEMSolver.h"
void FEMSolver::init(Mesh& mesh)
{
    // Memory is allocated for the matrices and vectors.

```

```
6       // Formation of global mass and stiffness matrices.
7       for(int l=0;l<elementCount;l++)
8       {
9           AV3INT tempTri  = mesh.mTriangles[l];
10          double x1 = mesh.mVertices[tempTri[0]].Position.x;
11          double y1 = mesh.mVertices[tempTri[0]].Position.y;
12          double x2 = mesh.mVertices[tempTri[1]].Position.x;
13          double y2 = mesh.mVertices[tempTri[1]].Position.y;
14          double x3 = mesh.mVertices[tempTri[2]].Position.x;
15          double y3 = mesh.mVertices[tempTri[2]].Position.y;
16
17          Matrix emm = elementMass(x1,x2,x3,y1,y2,y3);
18          Matrix esm = elementStiff(x1,x2,x3,y1,y2,y3);
19
20          for(int i=0;i<3;i++)
21          {
22              int i1=tempTri[i];
23              for(int j=0;j<3;j++)
24              {
25                  int j1 = tempTri[j];
26                  mGlobalStiff(i1,j1) = mGlobalStiff(i1,j1) + esm(i,j);
27                  mGlobalMass(i1,j1) = mGlobalMass(i1,j1) + emm(i,j);
28              }
29          }
30      }
31      initialSetup(mesh);
32  }
33
34  // The user of the class would be advised to modify these functions.
35
36  void FEMSolver::initialSetup(Mesh&mesh)
37  {
38      timestep = 0;
39      intervalEnd = 8;
40      numberOfTimeSteps = 400;
41      dt = (1.0*intervalEnd/numberOfTimeSteps);
42
43      // Used to store the derivatives of the solution.
44      V.setVector(Vector(nodeCount));
45
46      // User-defined constants.
47      theta = 0.5; /* Controls the numerical scheme. */
48      alpha = 1; /* Wave equation parameter. */
49
50      // Set initial profile, that is, the initial condition.
51  }
52
53  // Calling solve will essentially solve the problem.
54  void FEMSolver::solve(Mesh&mesh)
55  {
56      double a = -1.0*dt*dt*mTheta*(1-mTheta)/mAlpha;
57      // We solve the wave equation using the Crank-Nicolson scheme.
58      for(int i=0;i<numberOfTimeSteps;i++)
59      {
```

```
60          // How we get to these equations is explained in Appendix A.
61          Vector temp = (mGlobalMass+a*mGlobalStiff)*U + mTheta*dt*mGlobalMass*V;
62          Vector Unew = temp/(mGlobalMass+((dt*mTheta*dt*mTheta)/mAlpha)*mGlobalStiff);
63          temp = mGlobalStiff*(mTheta*Unew+(1-mTheta)*U);
64          Vector Vnew = ((mGlobalMass*V) - (dt/mAlpha)*temp)/mGlobalMass;
65          U = Unew;
66          V = Vnew;
67          mSolutions.push_back(U); mVelocities.push_back(V);
68      }
69  }
70
71  void FEMSolver::adaptMeshToSolutions(Mesh&mesh)
72  {
73      for(int h=1;h<solutions.size();h++)
74      {
75      Vector oldsol = solutions[h-1]; Vector oldvel = velocities[h-1];
76      // Creates a new mesh and approximates oldSol and oldVel
77      // oldSol and oldVel are modified. The advantage of this
78      // is that we still have our original solutions and mesh.
79      Mesh tMesh = mesh.adaptNew(solutions[h],0.25,oldsol,oldvel);
80
81      // Obtains dimensions for new temp mass and stiff matrices.
82      nodeCount = tMesh.mVertices.size();
83      elementCount = tMesh.mTriangles.size();
84
85      // Temporary global mass matrix and global stiffness matrix.
86      Matrix gmm(nodeCount,nodeCount);Matrix gsm(nodeCount,nodeCount);
87
88      // Formation of global mass and stiffness matrices
89      for(int l=0;l<elementCount;l++)
90      {
91          // Recalculate the mass and stiffness matrices
92          // for a temp mesh tMesh.
93
94          // Apply a single step of Crank-Nicholson
95          // to approximated previous time step solution.
96          // Then append to adapted solutions/velocities.
97      }
98  }
```

We have omitted some simple member functions from the above listing. In the code listing above we outline solving the wave equation in an adaptive sense. This is carried out in stages. The first stage is to calculate the solution at all time steps. We then use a gradient-based routine to determine the quality of the solution. If the solution at the current time step fails the quality test, we interpolate the solution of the previous time step over new triangles formed by `adaptNew(...)` and then solve the time step again using the Crank-Nicholson scheme with the interpolated previous time step as the previous time step to get an adapted current time step solution. A screen shot of the solution is given in Figure 7. We understand that this approach would be more useful on a parabolic problem like the heat equation. It is used in this

case to demonstrate functionality.

3.4 FEMDriver

`FEMDriver` threads all the objects discussed previously together to form a complete package. FEMDriver inherits from GLnixAPP.

```
#include"GLnixAPP.h"
#include"GeometryGen.h"
#include"MathHelper.h"
#include"FEMSolver.h"

class FEMDriver : public GLnixAPP
{
public:
        FEMDriver(); /* Constructor. */
        bool init(); /* Initialisation routine. */
        void updateScene(float dt); /* Calls Solve() from FEMSolver. */
        void redrawTheWindow();

        // Functions used to override mouse input.
        void onMouseDown(XButtonEvent btn,int x, int y);
        void onMouseUp(XButtonEvent btn,int x, int y);
        void onMouseMove(int x, int y);
private:
        // Functions used to package data ready to
        // send over to the GPU.
        void buildLandGridBuffers();
        void buildMeshBuffer();
public:
        // Used in the graphics pipeline.
        // VAO : Vertex Array Object.
        UINT mVAOmesh;
        UINT mVAOLAND;
        UINT mMeshIndexCount;
        UINT mGridIndexCount;
        UINT mBUFFERS[2];

        // Used to display the solution.
        GeometryGenerator::MeshData mMeshObj;
        GeometryGenerator mGeoGen;

        Mesh mesh;
        FEMSolver pfem;

        int mousex;
        int mousey;
        int but;

        // Projection matrices for displaying
        // 3D geometry.
        AV4X4FLOAT mViewModelMatrix;
        AV4X4FLOAT mProjMatrix;
```

```
47
48          // Used in mouse movement overrides.
49          float mTheta;
50          float mPhi;
51          float mRadius;
52  };
```

Notice that there are two instances of `mTheta`. This is not an issue as one might be accessed via `femd.mTheta` and the other via `femd.pfem.mTheta` in the application-specific file (discussed next).[6]

4 Application-Specific File

In order to use MassStiff the user is required to edit three member functions of `FEMSolver` and provide an application-specific script. The following script solves the wave equation:

```
1   #include<iostream>
2   #include"FEMDriver.h"
3   int main()
4   {
5           // Instantiates a FEMDriver object.
6           FEMDriver femd;
7           // Reads the vertices in from a file passed in as an argument.
8           femd.mesh.ReadVertices("STAR.txt");
9           // Initial triangulation.
10          femd.mesh.Triangulate();
11          // Refinement section, which includes multiple improvements
12          // The improvements are cheaper than the refinements and
13          // so do not need to be used frugally.
14          femd.mesh.Refine(1); femd.mesh.Improve();
15          femd.mesh.Refine(1); femd.mesh.Improve();
16          femd.mesh.Refine(1); femd.mesh.Improve();
17
18          // Forms mass and stiffness matrix and sets intial condition.
19          femd.pfem.init(femd.mesh);
20
21          // Solves the problem
22          femd.pfem.solve(femd.mesh);
23
24          // Attempts to improve the solution by using
25          // adaptivity.
26          femd.pfem.adaptMeshToSolutions(femd.mesh);
27
28          // To be included in every MassStiff project.
29          if( !femd.Init() )
30                  return 0;
31          return femd.Run();
32  }
```

[6] This point is not really valid as `mTheta` is private in `FEMSolver`.

We have made an anonymous YouTube account with footage of the solver running.[7] In the demonstration we use two different initial conditions:

$$u_0(x,y) = \begin{cases} \sqrt{x^2+y^2} & \sqrt{x^2+y^2}, \leq 0.4, \\ 0, & \text{otherwise.} \end{cases}$$

$$u_0(x,y) = 1 \text{ for } x \in \partial\Omega \text{ or } x > 1 \text{ or } y > 1.$$

In the terminal window of the linked screen capture we can see that "`Left Button Pressed`" is being displayed in the terminal. We move the scene around using the right and left mouse buttons. When the middle button is pressed we restart the solution. That is, the current time step is set to 0.

5 Error analysis: Poisson's Equation

We have not yet shown that the MassStiff actually solves FEM problems. To show this we will solve the following problem:

$$\nabla^2 u = \frac{1}{2}((x+1)(x-1) + (y+1)(y-1)) = s(x,y), \qquad u \in \Omega, \tag{3}$$

$$u = 0, \qquad u \in \partial\Omega, \tag{4}$$

where $\Omega = [-1,1] \times [-1,1]$. This is chosen as it has the exact solution

$$u(x,y) = \frac{(x+1)(x-1)(y+1)(y-1)}{4}. \tag{5}$$

If we multiply both sides of (3) by a test function v and integrate by parts, we obtain the weak form

$$-\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega = \int_\Omega sv \, \mathrm{d}\Omega. \tag{6}$$

For more information on the finite element method see [4]. If we now substitute the approximations

$$u = \sum u_i \phi_i, \quad v = \sum v_i \phi_i, \quad s = \sum s_i \phi_i, \tag{7}$$

where ϕ_i are the nodal basis functions and u_i, v_i and s_i are scalars, we obtain

$$-\int_\Omega (\nabla \phi_i \cdot \nabla \phi_j) u_j \, \mathrm{d}\Omega = \int_\Omega \phi_i \phi_j s_j \, \mathrm{d}\Omega. \tag{8}$$

[7] Search for `Mass Stiff MMSC Project` in the YouTube search bar and look for a channel called `Mass Stiff`. There will be two videos showing the capabilities of MassStiff.

We now have a system of equations

$$-Ku = Ms. \tag{9}$$

Further, we have to account for the Dirichlet boundary conditions. This is carried out in the following code:

```
void FEMSolver::initialSetup(Mesh&mesh)
{

        for(int i=0;i<mesh.mVertices.size();i++)
        {
                b(i) = // RIGHT HAND SIDE OF (3)
        }

        GlobalMatrix = GlobalStiff; b = GlobalMass*b;
        // Dirichlet boundary conditions
        for(int i=0;i<mesh.mVertices.size();i++)
        {
               if(mesh.mVertices[i].Boundary == 1)
               {
                     for(int j=0;j<mesh.mVertices.size();j++)
                     {
                         GlobalMatrix(i,j)=0;
                         GlobalMatrix(j,i)=0;
                      }
                      GlobalMatrix(i,i)=1;
                      b(i) = 0;
               }
        }
        U = b/GlobalMatrix;
}
```

In order to calculate the error we export the solutions to a `.txt` file and approximate the H_1 norm of the error.[8] As we can see from Figure 5, the error doubles if we double the mesh size. In addition, we found that intermediate adaptive meshes also decreased the error by an amount proportional to the number of triangles that were refined. For a plot of this solution, see Figure 8 of Appendix B.

6 Summary and Improvements

In this report we have discussed a FEM solver called MassStiff and a few examples of how it is used. There are many things that we would like to improve and in the remainder of this section we will discuss a few of these things. The first thing is the triangulation routine. Our journey into Computational Geometry started with

[8] MATLAB has some very useful commands such as `integral2` and `interp2` which were used in this calculation.

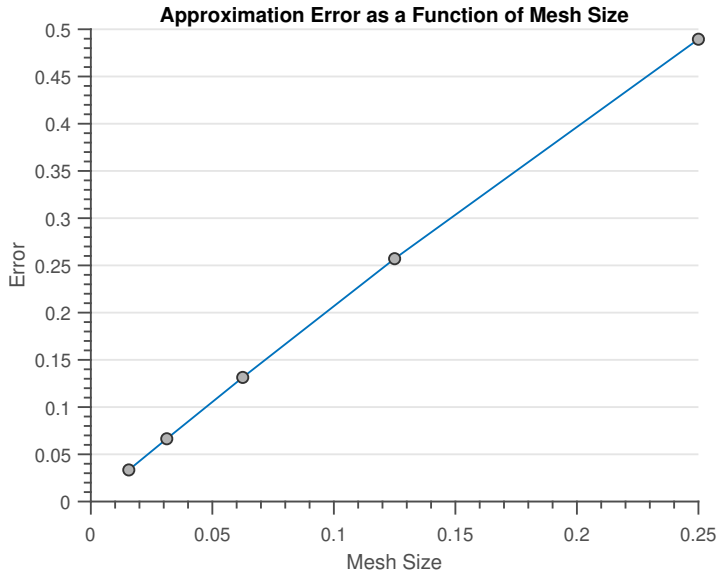

Figure 5: The relationship between the H_1 error and the mesh size.

[3], which introduces the ear clipping method as a method of triangulation. A more effiecient solution to the triangulation problem would be to implement a Delauney triangulation routine, which would also have been more robust.[9] Another issue is that we constructed and implemented all of the underlying data structures ourselves as opposed to seeking help from a textbook. A far better approach would have been to consult [5] which provides a full Delauney meshing routine as well as good underlying data structures. We would also attempt to improve the edge flipping routine so that it is more consistent. To highlight this inconsistency refer to the left image in Figure 2 and the left image in Figure 4 and compare that difference with the images in Figure 10. The result displayed in Figure 10 is more successful. We refrained from discussing the algorithmic complexity of our routines as, for a meaningful comparison we would have needed alternative methods for empirical testing. If we were to write this report again we would concentrate less on making a complete piece of software and more on comparison of triangulation routines.[10] From a software design aspect, we would have arranged the classes quite differently. For example, we would include some class that would allow us to make use of higher order finite elements such as a `DOF` manager, or `DegreesOfFreedomManager`.

[9] As a result of this it is the industry standard in FEM mesh generation.

[10] Although large-scale reusable software solutions are a ramification of languages such as C++.

References

[1] David Parsons. *Object Oriented Programming with C++*. Continuum.

[2] Tomas Petricek. *Real-World Functional Programming: With Examples in F# and C*. Manning Publications, 2010.

[3] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press.

[4] Patrick Farrel. Finite Element Methods for PDE's Lecture Notes. 22 June 2018.

[5] Michael J Laszlo. *Computational Geometry and Computer Graphics in C++*. Prentice Hall.

7 Appendix A: Crank-Nicholson Scheme

If we start with (1) and then define $v = u_t$, we obtain the system of equations

$$u_t = v, \tag{10}$$

$$v_t = \nabla^2 u, \tag{11}$$

$$\frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega. \tag{12}$$

If we now discretise in time we obtain

$$\frac{u^n - u^{n-1}}{dt} = [\theta v^n + (1-\theta)v^{n-1}], \tag{13}$$

$$\frac{v^n - v^{n-1}}{dt} = \nabla^2[\theta u^n + (1-\theta)u^{n-1}]. \tag{14}$$

We would like to remove the v_n term from the first equation. We achieve this by substituting the second equation into the first. The end result is that we avoid having to solve a very large system. After completing this step we are left with

$$[1 - (dt)^2\theta^2\nabla^2]u^n = [1 + (dt)^2\theta(1-\theta)\nabla^2]u^{n-1} + dtV^{n-1}, \tag{15}$$

$$v^n = v^{n-1} + dt\nabla^2[\theta u^n + (1-\theta)u^{n-1}]. \tag{16}$$

After multiplying both sides by a test function w, integrating by parts and making substitutions like that in (7), we are left with the matrix-vector equations

$$(M + (dt)^2\theta^2 K)u^n = Mu^{n-1} - (dt)^2\theta(1-\theta)Ku^{n-1} + kMv^{n-1} \tag{17}$$

$$Mv^n = Mv^{n-1} - dtK[\theta u^n + (1-\theta)u^{n-1}] \tag{18}$$

8 Appendix B: Extra Figures

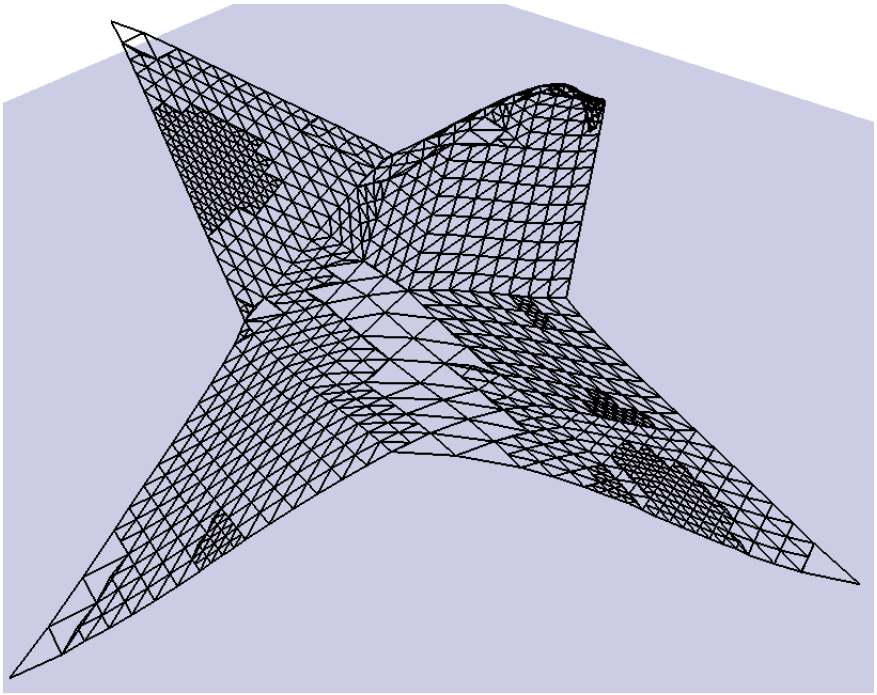

Figure 6: Solution of the Helmholtz equation.

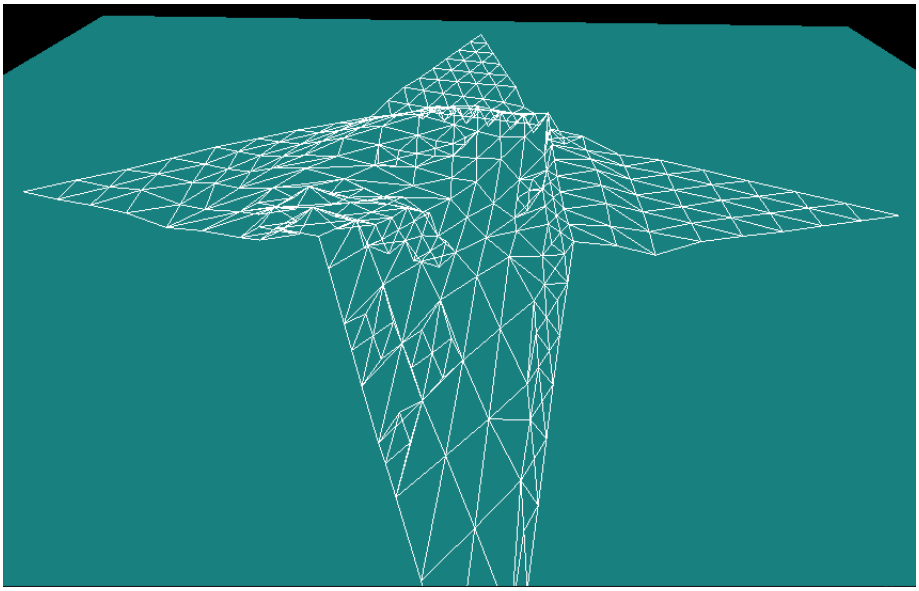

Figure 7: Solution of the wave equation described in Section 5.

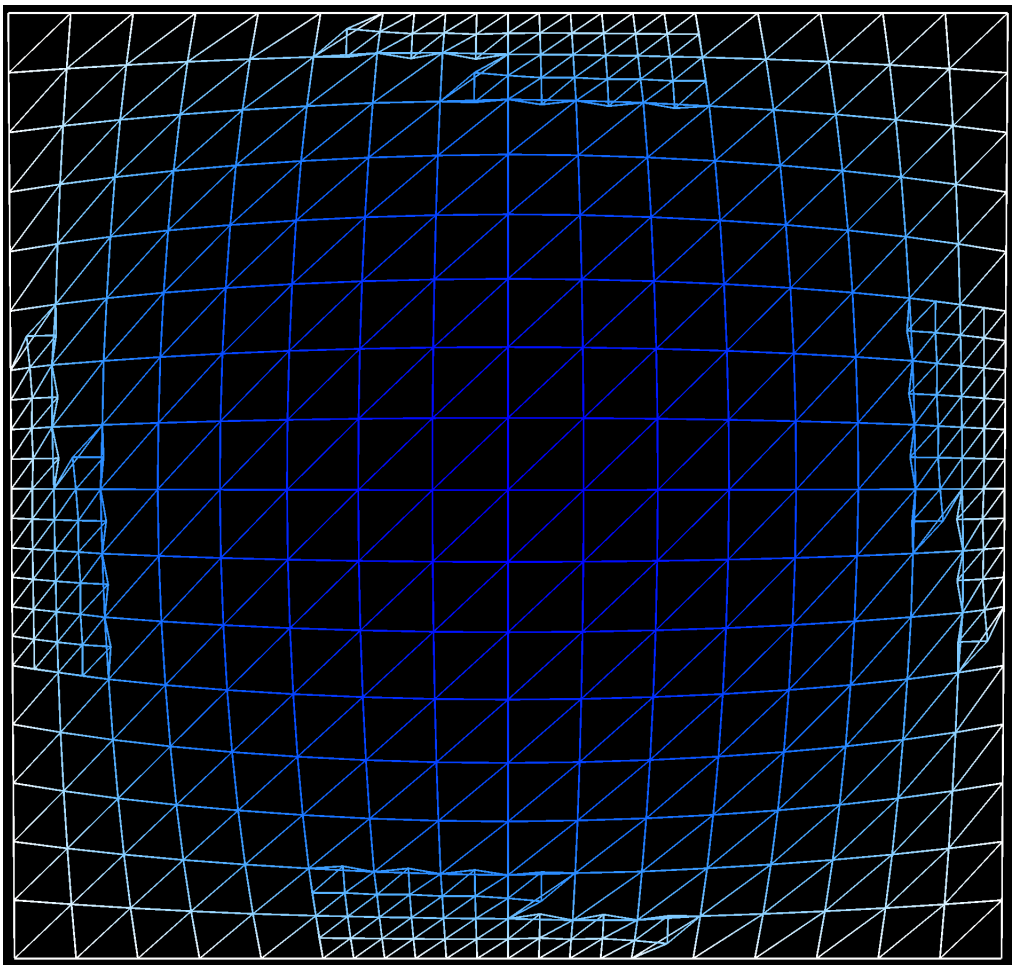

Figure 8: Solution of Poisson's equation on the domain $\Omega = [-1, 1] \times [-1, 1]$. White indicates values closer to 0 and dark blue indicates large negative values. This plot is different to the other plots in the sense that we assign a pixel a colour based on its displacement from the xy-plane.

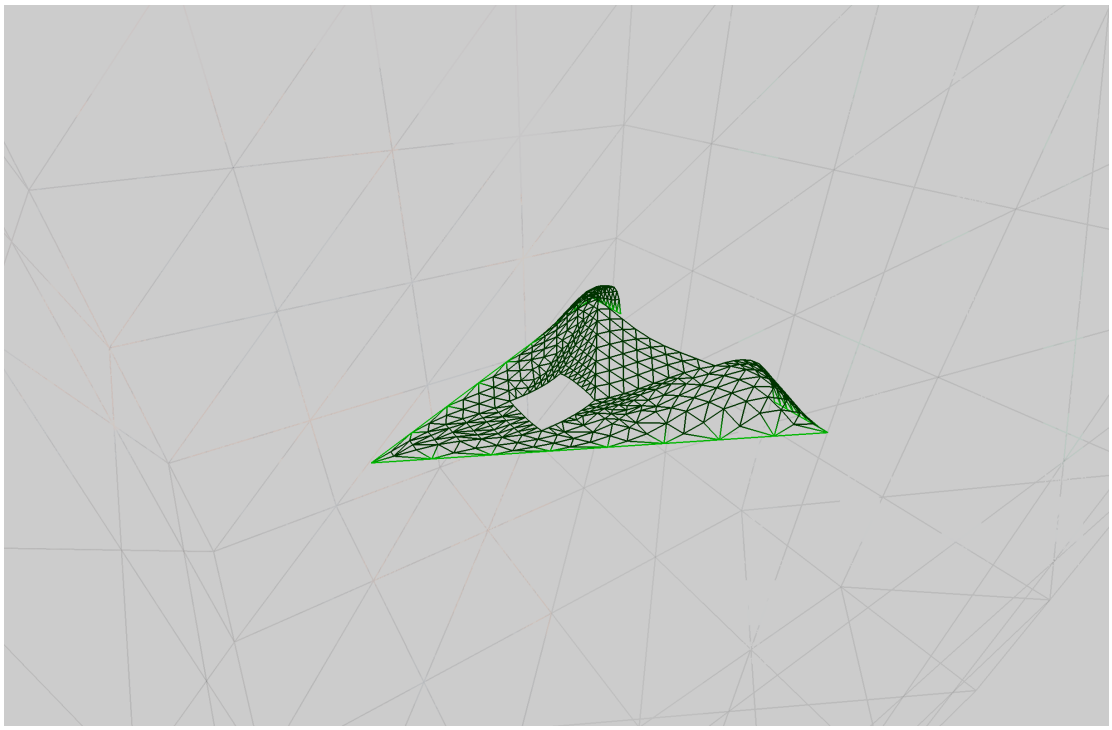

Figure 9: Solution of Poisson's equation with mixed boundary conditions. The domain is identical to that in Figure 2. The green lines indicate a Dirichlet boundary, the remaining boundaries are Neumann boundaries.

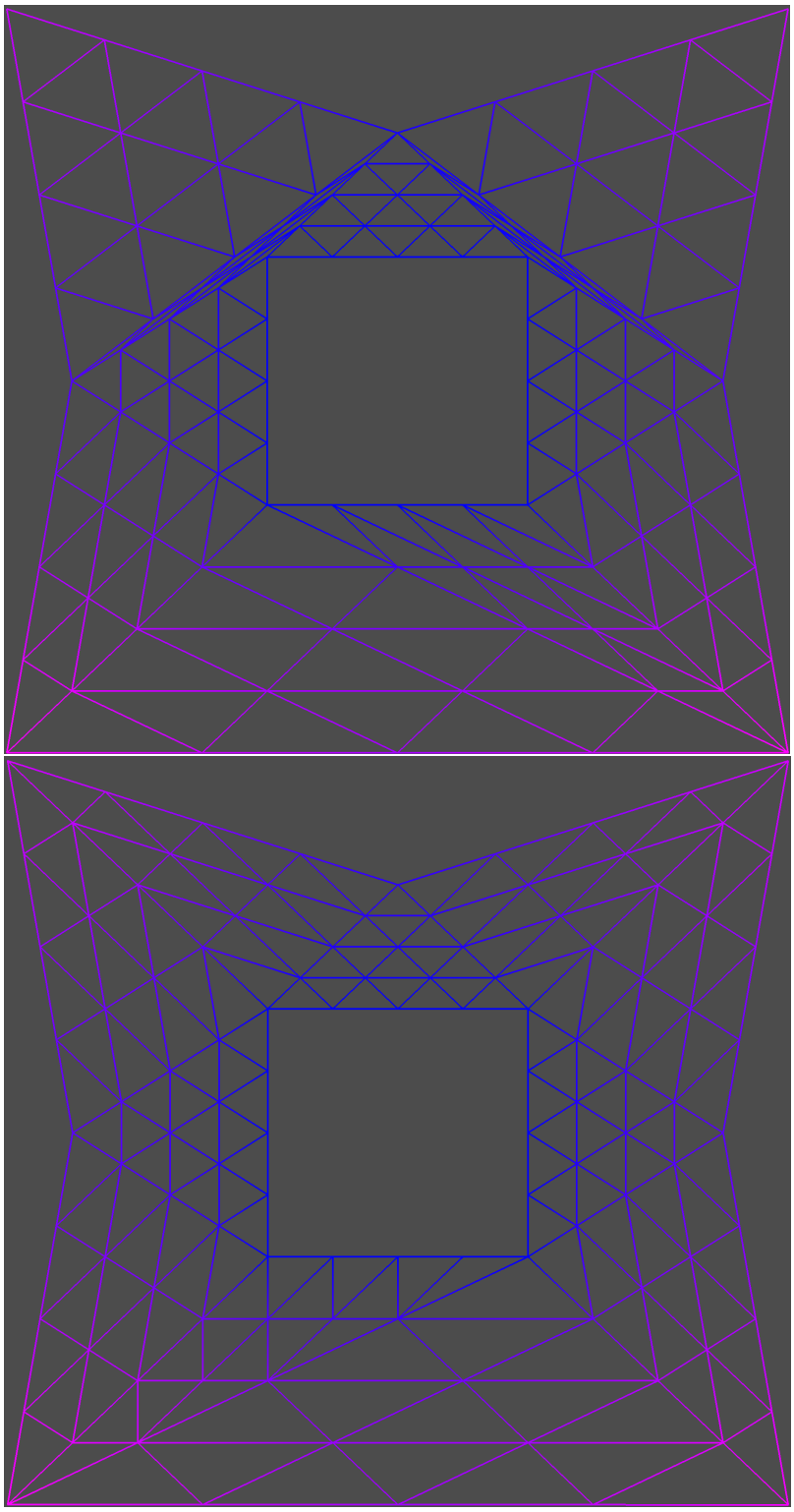

Figure 10: Solution of Poisson's equation with mixed boundary conditions. The interior square satisfies Dirichlet boundary conditions, the outside boundary satisfies Neumann boundary conditions. The bottom mesh has undergone edge swapping. Pink indicates large positive values and blue indicates small positive values.