Scalable Coursework 2

Cross-Platform Software Design

Data Structures, Algorithms and Design Coursework 2

Edward Percy – 4304249

The following documentation will outline scenarios and support evidence that the work required by the client works as intended by the brief and includes options for scalability and upgrades for the future. The data structures can load the clients existing .tri and .node files and save them. Furthermore, several functions are available to conduct circumcircle checks, Delaunay checks, integration of functions and runtime points tests. A discussion on how the code might be made parallelable and additional automated Catch 2.0 tests for proof and assurance to the client that all functions will work as intended.

Table of Contents

1. Data Structure

The program consists of three main classes; the mesh class, which is at the top of the hierarchy, the vertices class and finally the cell class which has a child class triangle. The reason for this is so that new shapes such as hexahedrons or tetrahedrons can easily be implemented around the current design structure, allowing for cells with more vertices per cell to be loaded. Other classes include the matrix class which is used primarily for the calculation of the circumcentre and radius.

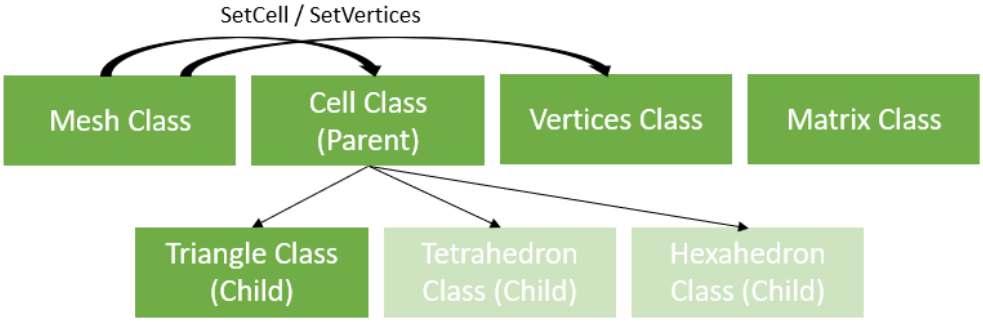

Figure 1 – Flowchart to show the classes data structure in the program

2. Delaunay Triangulation

To generate a Delaunay triangulation a few key algorithms are required, these are outlined below, and an explanation of the approaches taken, and the tests carried out is outlined.

2.1 Algorithms required

2.1.1 In which triangle does a new vertex fall?

The first algorithm required is a check to see which triangle ID a new vertex added falls within. The approach taken to do carry this out was brute force (Linear search). A for loop was used to run through all the triangles loaded previously into the vector of triangle objects (vector<Triangle>). Each triangle was then tested using the .isPointInside function which took X and Y parameters which were the coordinates of the point to test.

2.1.1.1 Algorithm improvements?

A Linear search is not the fastest method and it might be wise in the future to implement some sort of binary search. This could be implemented by ordering the triangles in the vector of triangles in such as way that the triangles with lower X values are at the start and higher values at the end. This could then be searched using a binary search to look for triangles close to the points X value (being tested). This would speed up the algorithm and not require all triangles to be searched, speeding up processing time and saving resources.

2.1.2 Testing

All tests within the program were conducted using Catch2.0 as this is a lightweight and portable header allowing for automated tests on aspects of the code. Below is a screenshot outlining the test taken to check the isPointInside test:

```
TEST_CASE( "PointinTriangleCheck", "[PointCheck]" ) {
    Mesh M = Mesh("../resources/triangulation_files/triangulation#1.tri");
    REQUIRE(M.isPointContained(70,-2) == 2229);
}
```

Figure 2 – TEST_CASE (Point in Triangle Check)

Further tests could be carried out to check more points but due to the time budget only a few could be done.

2.1.2 From start triangle search adjacent triangles

The next algorithm which is needed for Delaunay is a check of adjacent triangles starting at the initial triangle found above. This was not fully implemented due to time constraints but surround functions and classes have been taken into consideration so this may be added in the future with ease.

First the Point check will be called below:

```
isPointContained(point.getx(),point.gety());
```

To find the adjacent triangles a linear search of the vector<Triangles> list could be done to find triangles with the same vertex ID as the initial triangle.

These triangles could then be deleted from the vector<Triangles> list using the delete function.

Finally, new triangles could be defined using the following function below by connecting the new point to all existing nearby nodes to create the triangles.

```
Mesh::SetTriangle(int id,double V1,double V2,double V3,string attributes)
```

A final Circumcircle check described below could be used to check if it was successful.

2.1.3 Circumcircle test

The next test briefly noted above is the circumcircle test. This is used to see if a point is within the circumcircle of a triangle shown in figure 3. This check is also carried out using a linear brute force check by searching through each triangle in the vector<Triangles> list and running the CalcCircumcircle() test. From here we can compare the distance between Ox, Oy and the point with the radius and if it is < radius it is within the circumcircle.

There are two algorithms tested for calculating the circumcircle, one uses matrices and required the implementation of a Matrix class whereas the other using midpoints, slopes and solves for x and y. The matrix version is slightly quicker from testing however the midpoint

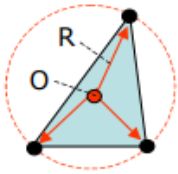

Figure 3 – Circumcircle representation

version appeared more reliable. Further testing into this will help determine which is best for the final version. Currently it has been kept using the matrix version.

```
TEST_CASE( "CircumcirclesCheck", "[CircumcirclesCheck]" ) {
    Mesh M = Mesh("../resources/triangulation_files/triangulation#2.tri");
    Vector p = Vector(65.674, 1.48582, 994);
    REQUIRE(M.CircumcirclesCheck(p) == 4);
```

Figure 4 – Test case to show circumcircle test carried out

2.1.3.1 Parallel thoughts

The Circumcircle check above could easily be made parallelisable as the check looks through all triangles in the vector<Triangles> list, so a different thread/worker could easily be given a different triangle to check as they are not dependent on each other. Each triangle is an object with its own data and no share data. If more time was given a test case using OpenMP on Jenna would have been tested but the 25-hour budget given by the client did not suffice this as necessary as it ran as reasonable speeds without.

Other algorithms which could be parallelised include the DelaunayCheck and the constant/linear integration algorithms discussed later.

The -O4 flag was also used when compiling to speed up and automatically optimise the code. This worked quite effectively.

2.1.4 Delaunay check on mesh

The final Algorithm mentioned above is the Delaunay check on the entire mesh to see if the triangulation obeyed the Delaunay principle. This was implemented using the following function in the Mesh class:

```
bool Mesh::DelaunayCheck(){
```

It returns a bool to tell the program if the mesh is Delaunay or not and prints the points which caused issues for debugging purposes.

The check works by looping through all vertices in the vector<Vertices> list and cross-checking every point against every triangle using the algorithm described above (CircumcircleCheck()).

2.1.4.1 Testing

Figure 5 below shows the test case used to check the Delaunay algorithm

```
TEST_CASE( "Delaunay Check", "[DelaunayCheck]" ) {
    Mesh M1 = Mesh("../resources/triangulation_files/triangulation#1.tri");
    Mesh M2 = Mesh("../resources/triangulation_files/triangulation#2.tri");
    Mesh M3 = Mesh("../resources/triangulation_files/triangulation#3.tri");
    Mesh M4 = Mesh("../resources/triangulation_files/triangulation#4.tri");

    REQUIRE(M1.DelaunayCheck() == false);
    REQUIRE(M2.DelaunayCheck() == false);
    REQUIRE(M3.DelaunayCheck() == true);
    REQUIRE(M4.DelaunayCheck() == false);
```

Figure 5 – Delaunay Check on the entire mesh (Test on tri#1,2,3 and 4)

The above test shows that we expect the Delaunay check to fail on all the test file given except triangulation#3.tri which we expect to be fully Delaunay.

Figure 6 below shows the debugging output from the console so that the specific points and triangles which they intercept can be seen.

```
Point(169.95,-5.84473e-007) in circumcircle triangleID(983)    PointID: 309
Point(-130.768,-8.50104) in circumcircle triangleID(1175)    PointID: 446
Point(-125.085,-9.74949) in circumcircle triangleID(1712)    PointID: 448
Point(173.232,-4.6017) in circumcircle triangleID(1904)    PointID: 514

Point(67.0922,1.45818) in circumcircle triangleID(1)    PointID: 58
Point(67.2426,0.000876253) in circumcircle triangleID(80)    PointID: 74


Point(90.8803,1.60727) in circumcircle triangleID(5)    PointID: 21
Point(96.9,0) in circumcircle triangleID(65)    PointID: 24
Point(93.1,0) in circumcircle triangleID(89)    PointID: 26
Point(99.1197,1.60727) in circumcircle triangleID(88)    PointID: 27
Point(96.7119,1.3317) in circumcircle triangleID(65)    PointID: 60
Point(93.2649,1.37586) in circumcircle triangleID(89)    PointID: 61
```

Figure 6 – Output from console for debugging to show PointIDs and specific triangles intercepted

The first four Points are from triangulation test file one. It shows that Points with ID 309, 449, 448 and 514 fail because they intercept the circumcircle of triangle IDs 983, 1175, 1712 and 1904. Below is a list of the findings from the tests:

Triangulation#1	Non-Delaunay	Failed PointIDs: 309,446,448,514
Triangulation#2	Non-Delaunay	Failed PointIDs: 58,74
Triangulation#3	Delaunay!	
Triangulation#2	Non-Delaunay	Failed PointIDs: 21,24,26,27,60,61

Due to precision errors this would need fully testing further with a test .tri file with known Failed PointIDs. This would confirm that the algorithms described above are functioning correctly.

3. File Streaming

3.1 Loading

Another main algorithm in the program is the ability to load and save .tri and .node files into the mesh and subsequent classes/objects. Figure 7 below shows the code used .tri load the .tri and .node files. It has been implemented with scalability in mind (discussed below).

```
while(file>>LinesToInterate>>NumCols>>NumProperties) {

    if (iterator==0){ //IF first line (READING VECTORS)
        dimensions = NumCols; //Saves the dimesntion for later saving if required
        for (int i(0);i<LinesToInterate;i++){ //Iterates through all the lines (no_points)

            //Checks the dimensions
            if (NumCols == 2) (file>>id>>X>>Y) && getline(file, attributes);
            else if (NumCols == 3) (file>>id>>X>>Y>>Z) && getline(file, attributes);
            else throw "ERROR: Invalid dimensions entered";
            //Sets the vertices with any attributes
            SetVertices(id,X,Y,Z,NumCols ,attributes);

        }
    }
```

```
    else{ //IF second line or further for scalabiluty? (READING TRIANGLES / OTHER SHAPES)
        NoArguments = NumCols; //Sets the number of arguments/parameters for saving the file later
        for (int i(0);i<LinesToInterate;i++){ //Loops through all cells/triangles in memory
            file>>id; //Gets the ID (First column)
            for (int v_per_cell(0);v_per_cell < NumCols ;v_per_cell++){ //Loops through the number of
                                                                        //Means that other shapes wit
                file>>temp;
                C_Vertices.push_back(temp); //Stores the vectors temporarily before stored in the cel
            }
            getline(file, attributes);  //Get the rest of the line (parameters) future versions could

            //checks the type of shape
            if (NumCols == 3) SetTriangle(id,C_Vertices[0],C_Vertices[1],C_Vertices[2],attributes); /
            else throw("ERROR: 3 vertices shapes only supported currently");
            C_Vertices.clear();

            //else if **** WRITE OTHER SHAPES HERE ****
            //else if (NumCols == 4) SetTetrahedron; (EXAMPLE)
            //else if (NumCols == 8) SetHexahedron; (EXAMPLE for future code)

        }
    }
    iterator +=1;
}
```

Figure 7 – Screenshot showing the algorithm to load .tri & .node files

The code has been implemented so that extra shapes can be added. First a parent cell class was used with generic parameters and variables for all shaped so that tetrahedrons, hexahedron or other types of cells the client wants can be implemented. It also stores the parameters as a string and splits them up later in the Cell class allowing as many parameters or extra ones in the future to be added. Furthermore, a check to see if the dimensions are 2 or 3 has been implemented so both. node and .tri files can be loaded.

```
TEST_CASE( ".Tri File loaded", "[LoadMesh]" ) {
    Mesh M = Mesh();
    M.LoadMesh("../resources/triangulation_files/triangulation#1.tri");

    REQUIRE( M.NumberVertices() == 1467);
    REQUIRE( M.NumberTriangles() == 2620);
}

TEST_CASE( ".node File loaded", "[LoadNodes]" ) {
    Mesh M = Mesh();
    M.LoadMesh("../resources/vertex_files/vertices#1.node");

    REQUIRE( M.NumberVertices() == 22);
```

Figure 8 – Test case for loading files

To test the functionality a test case was created to load the triangulation files and .node files and check if the number of vertices and triangles was as expected. Below are the results used in the console for debugging:

```
Number of triangles loaded: 2620
Number of vertices loaded: 1467

Number of triangles loaded: 0
Number of vertices loaded: 22
```

Figure 9 – Console output to show triangles and vertices loaded

3.2 Saving

The saving algorithm is very similar to the loading but in reverse. It works by writing the initialising lines shown below:

```
file << Vertices.size() << " " << dimensions << " " << 0; //Writes the initialising
line to define the arguments for the vertices#
if (Triangles.size() > 0) file << "\n" << Triangles.size() << " " << 3 << " " << 17;
```

it then loops through all the Vertices and triangles in the vector lists and writes them to the out file. Checks on the dimensions and number of arguments are carried out so the correct amount are saved.

Throughout the code error checking is implemented in the form of throw. One check which was not implemented is to see if the file name is .tri or .node and only accept them. With more time this would be added as well as options for scalability by allowing other files. Maybe .Tet files for tetrahedrons?

3.2.1 Testing

Tests like loading are carried out in the automated tests. The test case loads the .tri file into the objects and then saves it, next it re-loads the new file and carries out the NumVertices and NumTriangles test.

```
TEST_CASE( ".Tri File Saved", "[SaveMesh]" ) {
    Mesh M = Mesh("../resources/triangulation_files/triangulation#1.tri");
    M.SaveMesh("triangulation#1_copy.node");

    Mesh MCopy = Mesh("triangulation#1_copy.node");
    REQUIRE( MCopy.NumberVertices() == 1467);
    REQUIRE( MCopy.NumberTriangles() == 2620);

}

TEST_CASE( ".node File Saved", "[SaveNodes]" ) {
    Mesh M = Mesh("../resources/vertex_files/vertices#1.node");
    M.SaveMesh("vertices#1_copy.node");

    Mesh MCopy = Mesh("vertices#1_copy.node");
    REQUIRE( MCopy.NumberVertices() == 22);
```

Figure 10 – Test case for saving .node and .tri files

Furthermore, tests to check for invalid files (file could not be opened), invalid dimensions and an invalid number of vertices per cell were implemented where the test case requires the throw exception. Figure 11 below shows in detail what is meant by this.

```
TEST_CASE( "Invalid dimension", "[InvalidDimensions]" ) {
    Mesh M = Mesh();
    REQUIRE_THROWS_WITH(M.LoadMesh("../resources/triangulation#5.tri"),"ERROR: Invalid dimensions entered");

}

TEST_CASE( "Invalid file", "[InvalidFile]" ) {
    Mesh M = Mesh();
    REQUIRE_THROWS_WITH(M.LoadMesh("triangulation#6.tri"),"ERROR: File could not be opened");

}


TEST_CASE( "Invalid Number of Vertices per cell", "[InvalidVpC]" ) {
    Mesh M = Mesh();
    REQUIRE_THROWS_WITH(M.LoadMesh("../resources/triangulation#7.tri"),"ERROR: 3 vertices shapes only supported currently");
```

Figure 11 – Test case for invalidation errors

4. Integration of function

Another requirement of the client was the ability to integrate a function given at compile-time over the domain of the mesh. Two approximations were implemented, Linear and Constant in the form of templates so that they could be passed a function chosen by the client. Figure 12 shows how the template takes a function and the type of approximation as parameters.

```
template <typename T>
double integrate(T func, int ApproxType) { //Allows input of functions
    //I wanted to pass the Approx type as a templated function reeference but was not able to
    //a templated function to a template and I ran out of time so had to use int
    double res,temp = 0;
    //loops through all triangles to get integration over entire domain of the mesh
    for (int i = 0; i < Triangles.size(); i++) {
        //checks the type of approximation the user wants
        if (ApproxType == 0) {
            temp =  abs(ConstantApprox(func, Triangles[i])); //Constant approximation
            if(isnan(temp)==false) res += temp;
        }
        else if (ApproxType == 1){
            temp = abs(LinearApprox(func, Triangles[i])); //linear approximation
            if(isnan(temp)==false) res += temp; //Check for infinite values /0 etc
        }
    }
    return res;
}
```

Figure 12 – Templated integration function

4.1 Constant Approximation

For the Constant approximation the formula from Appendix 2 was used: A * F(Ox,Oy) to approximate the area. This was iterated for all triangles and the result was summed.

```
//constant approximation check
template<typename T>
double ConstantApprox(T func, Triangle &triangle) {
    //formula in appendix of sheet given by client

    //A * F(Ox,Oy) for all triangles
    Vector O = triangle.getCircumcentre();
    return (triangle.getarea() *  func(O.getx(),O.gety()));
}
```

Figure 13 – Templated Constant approximation function

4.2 Linear Approximation

Figure 14 shows the same approximation but using a linear approach where the A/3 approximation was used for the area Ai (up to the client which method they choose)

```
template<typename T>
double LinearApprox(T func, Triangle &triangle) {
    vector<int> p = triangle.getVertices();
    //Formula given in appendix by client
    //A/3 (F0(x,y) + F1(x,y) + F2(x,y))
    double res = ((
                (triangle.getarea() / 3) * func(Vertices[p[0]].getx(),Vertices[p[0]].gety()) +
                (triangle.getarea() / 3) * func(Vertices[p[1]].getx(),Vertices[p[1]].gety()) +
                (triangle.getarea() / 3) * func(Vertices[p[2]].getx(),Vertices[p[2]].gety())
                )
    );
    return res;
}
```

Figure 14 – Templated Linear Approximation function

4.3 Testing

Using Catch 2.0 the described algorithms were tested. A simple function x*3 + y*3 was chosen to test the functionality of the algorithm.

```
double function (double x, double y){
        return x*3 + y*3;
}

TEST_CASE( "Integrate Function over mesh (Constant)", "[ConstantIntegration]" ) {
    Mesh M = Mesh("../resources/triangulation_files/triangulation#1.tri");
    for (int i(0);i < M.NumberTriangles();i++){
        M.CalcCircumcircle(i);
    }
    double I = M.integrate(function, 0);
    cout << "Constant Approx: " << I << endl;
    REQUIRE(I == Approx(2920231.515));
}
TEST_CASE( "Integrate Function over mesh (Linear)", "[LinearIntegration]" ) {
    Mesh M = Mesh("../resources/triangulation_files/triangulation#1.tri");
    double I = M.integrate(function, 1);
    cout << "Linear Approx: " << I << endl;
    REQUIRE(I == Approx(2920218.269));
```

Figure 15 – Test case for linear and constant integration

The results from the console are shown below:

```
Linear Approx: 2.92022e+006
```

```
Constant Approx: 2.92023e+006
```

5. Additional Tests

As previously described all tests were carried out using Catch2.0 and the tests were split up into four test files. Mesh_Class_Test.cpp, Matrix_Class_Test.cpp, Triangle_Class_Test.cpp and Vector_Class_Test.cpp. The tests varied and briefly noted below:

- Initiating objects
- The main functions described above
- Inversing matrix, multiplying 3x3 and 1x3 matrix
- Vector equality test, subtracting vectors
- Triangle Parameter testing and circumcircle tests

6. Jenna

All code demonstrated above has been tested above on both Windows OS and Jenna (Debian), compiled using the following code:

```
g++ -O4 ../src/*.cpp ../tests/*.cpp -o Mesh_Loader
```

The screenshot below in Figure shows that all automated tests passed

```
===============================================================================
All tests passed (42 assertions in 21 test cases)
```

Figure 16 – Screenshot from console to show tests passing

7. Contact

Email: eeyep2@nottingham.ac.uk