Data Structures and Algorithms Design for 2D Triangulation

1. Introduction

In the C++ project, we designed **memory efficient** data structures and algorithms for 2D triangulation.

The project supports:

-File streaming input and output of triangulations
-Given a point, check which triangle contains it
-Check whether a triangulated mesh is Delaunay
-Given a function f(x, y), evaluate the integral of f(x, y) over the domain of the triangulation (by constant value approximation or linear interpolation approximation)

The project also has features that support update for Bowyer Watson algorithm:

-Add a vertex to the point container
-Identify the triangles whose circumcircles enclose the new vertex
-Delete triangles from the triangle container
-Create new triangles connected to the new vertex and add them to the triangle container

2. Project Design

2.1 Data Structure Design

2.1.1 class point

Description:

An instance of class “point” represents a point in 3-dimentional space.

Private Members:

double x, y, z - coordinates of a 3D point

Note: In 2D cases, z coordinates of points are set to the same.

2.1.2 class triangle

Description:

An instance of class “triangle” represents triangles in 3-dimentional space. (also works for 2D case as z coordinates of points will be set to the same)

Private Members:

point* v0, v1, v2 - pointers to vertexes of a triangle

vector<double> attribute - a series of values, represents a triangle's attributes

Note: Because some vertexes are used in more than one triangle, pointers to vertexes are used to avoid each triangle storing its own copy of each of its vertices, thus making the data structure memory efficient.

2.1.3 class triangulation

Description:

An instance of class "triangulation" represents a triangulated mesh.

Private Members:

int no_point - number of points in the mesh

int dimension - 2 or 3

int attribute_per_point - number of attributes per point

map<int, point> point_set - container to store points, key is the index of point

int no_cell - number of cells in the mesh

int vertex_per_cell - number of verteses per cell

int no_attribute - number of attributes per cell

map<int, triangle> triangle_set - container to store cells, key is the index of cell

vector<string> other - other properties of the mesh

Note: map<int, point> is chosen as the container to store points because map has the following properties:

Associative: Each element associates a key to a mapped value, which means index of point is associated to point in this case.

Unique keys: No two elements in the container can have equivalent keys, and indexes of points are also unique.

Ordered: Internally, the elements in a map are always sorted by its key following a specific strict weak ordering criterion, allowing direct iteration.

The reason why map<int, triangle> is chosen as the container to store cells is similar.

2.2 Interface Design

2.2.1 triangulation(ifstream& ifs)

Description:

Construct a "triangulation" instance from a file.

Example Usage:

```
ifstream in("triangulation#0.tri");
triangulation t = triangulation(in);
```

2.2.2 void TriangulationOutput(ofstream& ofs)

Description:

Output the information of triangulation to a file.

Example Usage:

```
ifstream in("triangulation#0.tri");
triangulation t = triangulation(in);
ofstream out("output#0.tri");
t.TriangulationOutput(out);
```

Design:

Part of the code for getting the indexes associated to a triangle's vertexes:

```
for (map<int, triangle>::iterator tsit = triangle_set.begin();
tsit != triangle_set.end();++tsit) {
    int v0_index, v1_index, v2_index;
    map<int, point>::iterator psit;
    for (psit = point_set.begin();psit != point_set.end(); ++psit) {
        if ((*psit).second == (*tsit).second.get_v0()) {
            v0_index = (*psit).first;
        }
    }
    for (psit = point_set.begin();psit != point_set.end(); ++psit) {
        if ((*psit).second == (*tsit).second.get_v1()) {
            v1_index = (*psit).first;
        }
    }
    for (psit = point_set.begin();psit != point_set.end(); ++psit) {
        if ((*psit).second == (*tsit).second.get_v2()) {
            v2_index = (*psit).first;
        }
```

```
    }
}
```

Note: To find the point in points' container representing a triangle's vertex, it's needed to check whether the 2 points are the same, for example:

```
(*psit).second == (*tsit).second.get_v0()
```

Thus the equality operator in class point is overloaded:

```
bool point::operator==(const point &p) {
    return x == p.x && y == p.y && z == p.z;
    //2 "point"s equal when all the corresponding coordinates equal
}
```

2.2.3 int PointInWhichTriangle(point p)

Description:

Return the index of triangle which contains the point, if no triangle contains the point, return -1.

Parameter(s):

point p - a "point" instance to be tested

Example Usage:

```
ifstream in("triangulation#0.tri");
triangulation t = triangulation(in);
point p0(1.1, 1.1, 0);
cout << t.PointInWhichTriangle(p0) << '\n';
```

Design:

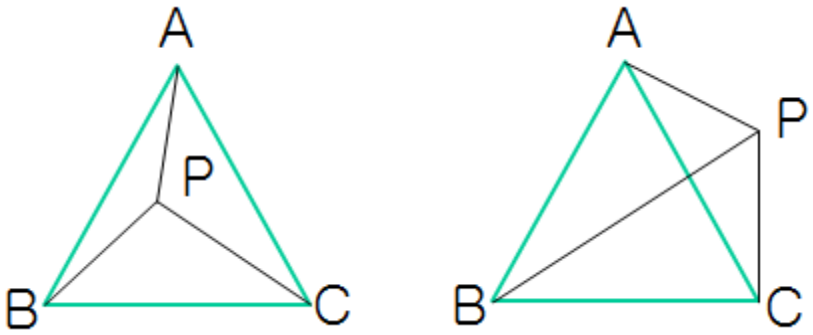
```
int triangulation::PointInWhichTriangle(point p) {

    /*modified version of "find_if" function
    *for each triangle in the container, check whether the triangle contains the point
    *if the triangle contains the point, return the index of the triangle*/
    map<int, triangle>::iterator tsit;
    for (tsit = triangle_set.begin();tsit != triangle_set.end();++tsit) {

        if ((*tsit).second.PointInTriangle(p))//check whether the triangle contains
the point
            return (*tsit).first;
    }
```

```
    /*if all the triangles are checked and the function doesn't return, no triangle
contains the point*/
    return -1;
}
```

Note 1: To check whether the triangle contains the point, like:

```
    (*tsit).second.PointInTriangle(p)
```

The following rules are followed:



If S△PAB + S△PBC + S△PCA > S△ABC, P is outside △ABC.
Else if S△PAB + S△PBC + S△PCA = S△ABC, P is inside △ABC.

The rules above is implemented as the following code:

```
bool triangle::PointInTriangle(point p) {
    if (AreaOfTriangle() < (p.AreaOfTriangle(*v0, *v1) + p.AreaOfTriangle(*v1, *v2) +
p.AreaOfTriangle(*v2, *v0)))
        return 0;
    else
        return 1;
}
```

Note 2: To get the area of triangle in the code above, like:

```
    AreaOfTriangle()
    p.AreaOfTriangle(*v0, *v1)
```

Geometric significance of the vector outer product operation is used:

$$S_{\Delta}ABC = \frac{1}{2}\left|\overrightarrow{AB} \times \overrightarrow{AC}\right|$$

The formula is implemented as the following code:

```
double point::AreaOfTriangle(point v1, point v2) {
    return fabs(0.5 * ((v1.x - x) * (v2.y - y) - (v1.y - y) * (v2.x - x)));
}
```

```
double triangle::AreaOfTriangle() {
    return (*v0).AreaOfTriangle(*v1, *v2);
}
```

2.2.4 bool Delaunay()

Description:

Return 1 if the triangulation is Delaunay, else return 0.

Example Usage:

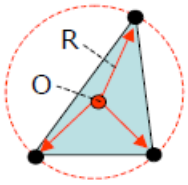
```
    ifstream in("triangulation#0.tri");
    triangulation t = triangulation(in);
    cout << t.Delaunay() << '\n';
```

Design:

```
bool triangulation::Delaunay() {
    /*for each triangle in the container, check whether there is a point in the
cricumcircle
    *if there is one, the triangulation is not Delaunay
    *else the triangulation is Delaunay*/
    for (map<int, triangle>::iterator tsit = triangle_set.begin();
tsit != triangle_set.end();++tsit) {
        triangle t((*tsit).second);
        double r2(t.CircumcircleCenter().SquareOfDistance(t.get_v0()));//calculate
square value of the triangle's circumcircle's radius

        /*modified version of "find_if" function*/
        for (map<int, point>::iterator psit = point_set.begin();
psit != point_set.end();++psit) {
            if (!t.PointInCircumcircle((*psit).second, r2))//check whether the point
is in the cricumcircle
                return 0;
        }
    }
    return 1;
}
```

Note 1: To get triangle's circumcircle center, like:

```
    t.CircumcircleCenter()
```



The following order 3 matrix equation is solved:

$$\begin{pmatrix} X_0^2+Y_0^2 \\ X_1^2+Y_1^2 \\ X_2^2+Y_2^2 \end{pmatrix} = \begin{pmatrix} X_0 & Y_0 & 1 \\ X_1 & Y_1 & 1 \\ X_2 & Y_2 & 1 \end{pmatrix} \begin{pmatrix} 2O_x \\ 2O_y \\ R^2-O_x^2-O_y^2 \end{pmatrix}$$

The solutions are implemented as the following code:

```
point triangle::CircumcircleCenter() {
    double v0_x(v0->get_x()), v0_y(v0->get_y());
    double v1_x(v1->get_x()), v1_y(v1->get_y());
    double v2_x(v2->get_x()), v2_y(v2->get_y());
    return point
    (
        ((v1_y - v0_y) * (v2_y * v2_y - v0_y * v0_y + v2_x * v2_x - v0_x * v0_x)
            - (v2_y - v0_y) * (v1_y * v1_y - v0_y * v0_y + v1_x * v1_x - v0_x * v0_x))
        / (2.0 * ((v2_x - v0_x) * (v1_y - v0_y) - (v1_x - v0_x) * (v2_y - v0_y))),
        ((v1_x - v0_x) * (v2_x * v2_x - v0_x * v0_x + v2_y * v2_y - v0_y * v0_y)
            - (v2_x - v0_x) * (v1_x * v1_x - v0_x * v0_x + v1_y * v1_y - v0_y * v0_y))
        / (2.0 * ((v2_y - v0_y) * (v1_x - v0_x) - (v1_y - v0_y) * (v2_x - v0_x))),
v0->get_z()
    );
}
```

Note 2: To check whether the point is in triangle's circumcircle, like:

```
t.PointInCircumcircle((*psit).second, r2)
```

It is implemented as the following code:

```
bool triangle::PointInCircumcircle(point p, double r2) {
    return (CircumcircleCenter().SquareOfDistance(p) <= r2 ? 1 : 0);
    //if the point is in the triangle's circumcircle, the distance between the point
and circumcircle center is less than radius
}
```

Because knowing the relationship between distances is sufficient, only square of distance is calculated, implemented as the following code:

```
double point::SquareOfDistance(point p) {
    return (x - p.x) * (x - p.x) + (y - p.y) * (y - p.y) + (z - p.z) * (z - p.z);
}
```

2.2.5 double ConstantValueApproximation()

double LinearInterpolationApproximation()

Description:

Given a function f(x, y), evaluate the integral of f(x, y) over the domain of the triangulation by constant value approximation or linear interpolation approximation.

Example Usage:

```
    ifstream in("triangulation#0.tri");
    triangulation t = triangulation(in);
    cout << t.ConstantValueApproximation() << '\n';
    cout << t.LinearInterpolationApproximation() << '\n';
```

Design:

```
double triangulation::ConstantValueApproximation() {
    double i;
    vector<double> I;

    /*evaluate constant value approximation to the integral of f(x,y) over each
triangle*/
    for (map<int, triangle>::iterator tsit = triangle_set.begin();
tsit != triangle_set.end();++tsit) {
        triangle t((*tsit).second);
        i = t.AreaOfTriangle() * fxy(t.CircumcircleCenter().get_x(),
t.CircumcircleCenter().get_y());
        I.push_back(i);
    }

    /*calculate the sum*/
    double sum(0.);
    for (vector<double>::iterator it = I.begin();it != I.end();++it) {
        sum += *it;
    }
    return sum;
}
```

```
double triangulation::LinearInterpolationApproximation() {
    double i;
    vector<double> I;

    /*evaluate linear interpolation approximation to the integral of f(x,y) over each
triangle*/

    for (map<int, triangle>::iterator tsit = triangle_set.begin();tsit !=
triangle_set.end();++tsit) {
        triangle t((*tsit).second);
        i = (fxy(t.get_v0().get_x(), t.get_v0().get_y())
            + fxy(t.get_v1().get_x(), t.get_v1().get_y())
            + fxy(t.get_v2().get_x(), t.get_v2().get_y()));
        I.push_back(i);
    }

    /*calculate the sum*/
    double sum(0.);
    for (vector<double>::iterator it = I.begin();it != I.end();++it) {
        sum += *it;
    }
    return sum / 3.;
}
```

Note: In both functions, the first loop is parallelizable, because it's **SIMD** case. Evaluating approximation to the integral of f(x, y) over each triangle is similar, and the sum is not affected by adding order. (However should take simultaneous writing issue into account)

3.Test

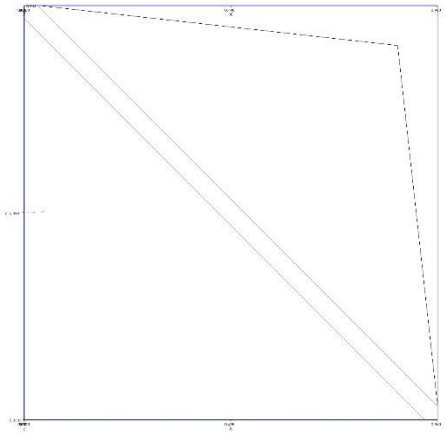
To test whether the project works, 2 simple and typical triangulations are used.

Function f(x, y) to be integrated over the domain of the triangulation is defined as:

```
double triangulation::fxy(double x, double y) {
    return x * y;
}
```

The first one is Delaunay triangulation containing 4 points and 2 triangles, ***triangulation#0.tri***:

```
4 3 0
0 0 0 0
1 1 0 0
2 0 1 0
3 0.9 0.9 0
2 3 0
0 0 1 2
1 1 2 3
```



Execute the following code:

```
int main() {
    ifstream in("triangulation#0.tri");
    triangulation t = triangulation(in);

    point p0(1.1, 1.1, 0);//P0 is not in any triangle
    cout << t.PointInWhichTriangle(p0) << '\n';//should return -1
    point p1(0.1, 0.1, 0);//P1 is in triangle #0
    cout << t.PointInWhichTriangle(p1) << '\n';//should return 0
    point p2(0.6, 0.6, 0);//P2 is in triangle #1
    cout << t.PointInWhichTriangle(p2) << '\n';//should return 1
    cout << t.Delaunay() << '\n';//should return 1
    cout << t.ConstantValueApproximation() << '\n';
    cout << t.LinearInterpolationApproximation() << '\n';

    ofstream out("output#0.tri");
    t.TriangulationOutput(out);
}
```

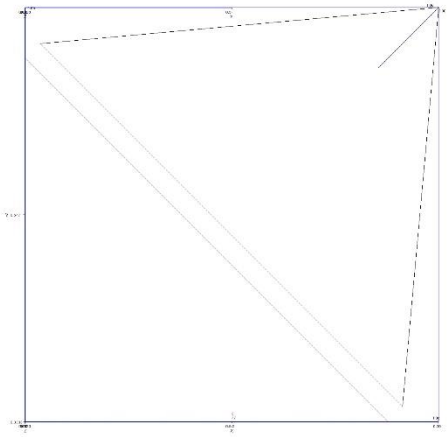
The output is as expected:

```
-1
0
1
1
0.185063
0.27
```

Check for identical results by doing ***diff triangulation#0.tri output#0.tri*** and get no difference at all.

To further test function ***Delaunay()***, use the second triangulation containing 4 points and 2 triangles, which is non-Delaunay, ***triangulation#1.tri***:

```
4 3 0
0 0 0 0
1 1 0 0
2 0 1 0
3 1.1 1.1 0
2 3 0
0 0 1 2
1 1 2 3
```



The output for Delaunay test is 0, as expected.