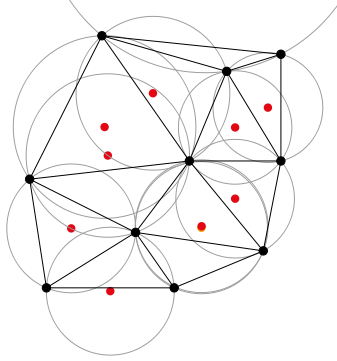
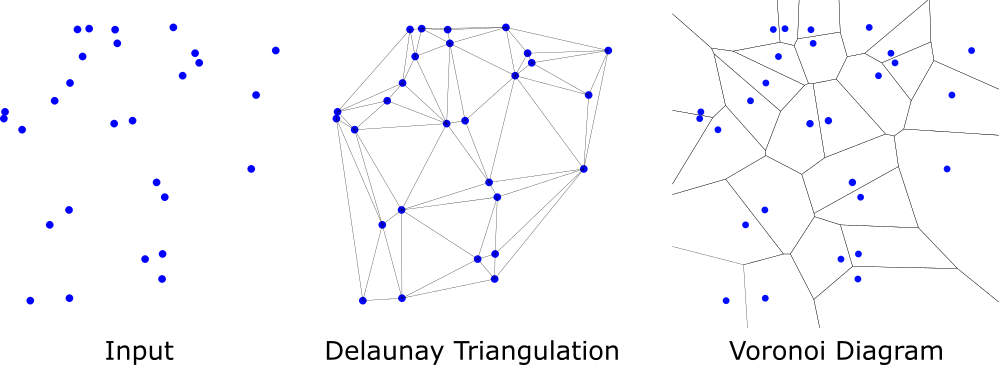
**DELAUNAY TRIGULATION**

**Triangulation** is the decomposition of 2D geometrical planar object **P** into a set of triangles. It involves finding a set of triangles with pairwise non-intersecting interiors whose union geometrically approximates **P**.

<https://www.ics.uci.edu/~eppstein/pubs/BerEpp-CEG-95.pdf>

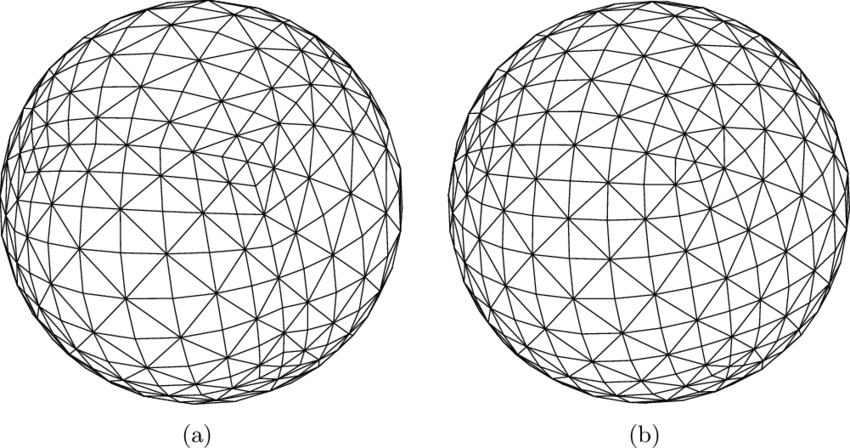
**Delaunay triangulation:** Given a set **input** discrete points in a plane the Delaunay triangulation discretizes such that no point  is inside the circumcircle of any triangle in. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; they tend to avoid sliver triangles.

**Images of triangulation**



Delaunay triangulation with

circumcircles shown

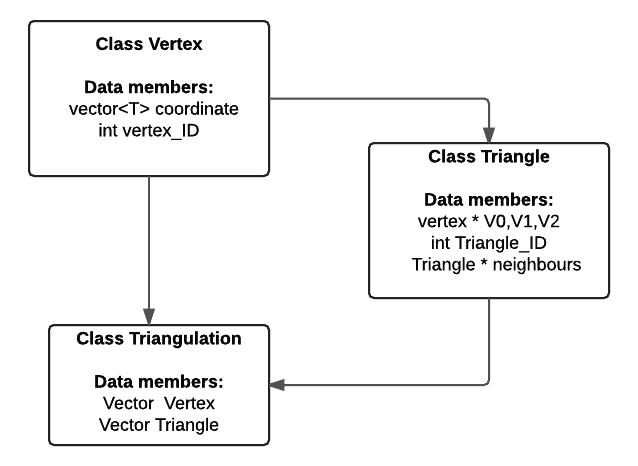


A triangulated sphere

**CODE DESCRIPTION**

The aim of this exercise was to design a software program that encapsulates a triangulation from a “tri” file and “node” file. The code was to be enabled for:

1. file input and output i.e. can read an input tri.file, manipulate and output a tri.file.
2. queries of the form “ given x,y coordinates locate the triangle containing it “
3. performing the integral of a function f(x,y) over a triangulation domain
4. check if a triangulation is Delaunay.
5. Figure 4.1 below illustrates the data structure employed to encapsulate the triangulation.



**Figure 4.1** Flow diagram illustrating triangulation data structure.

The following design rules were set in order to ensure code practicality

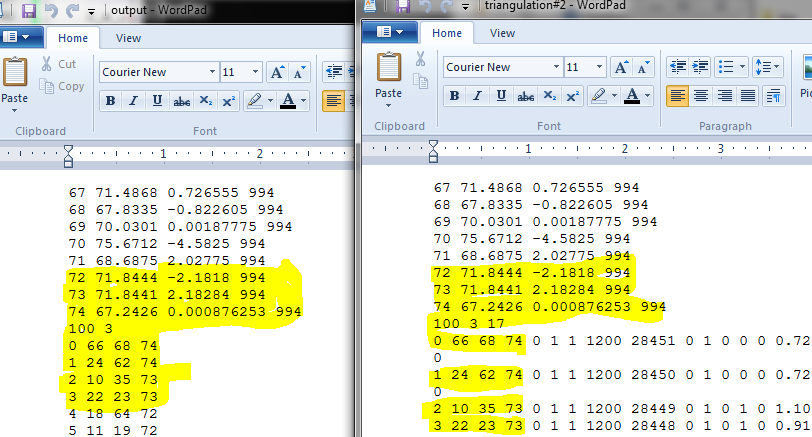
**Design rule 1:** Assume the user is fairly conversant with aspects of the data structure, as well as with the interfacing files. Therefore all vertex IDs and Tri IDs can only take the integer type, the coordinate points on the other hand are template type, allowing for compile time flexibility.

**Design rule 2:** Allow the user to decide which integral function to use i.e. linear interpolation or constant time approximation.

**TESTS**

A set of tests were designed to check the code compiled successfully and the functions were correct. These tests are described below and can be found in the main.cpp

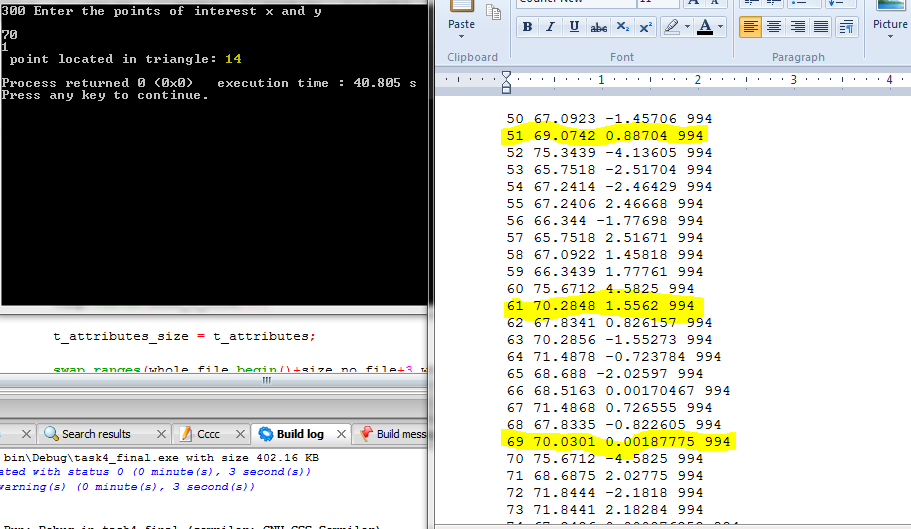
**Test1 – FILE I/O:** In order to check the correctness of the read and the screenshot below demonstrates the file I/O works.



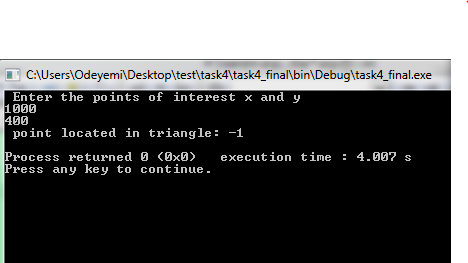
I did not read the attributes, the client (Phil Sewell ) said it was not necessary. Hence a file comparison could not be done using code.

**Output.tri Triangulation#2.tri**

**Test2 – Finding which triangle holds the points x,y In order to find the triangle holding the set of points, I employed the use of the std find\_if ( ) algorithm**. Which took iterators pointing to the first triangle and last triangle in a vector of triangle objects. The function was passed a functor **is\_point\_in\_tri** which checked each individual triangle in the vector and returned a bool. The screenshots provided demonstrate the computation was correct.



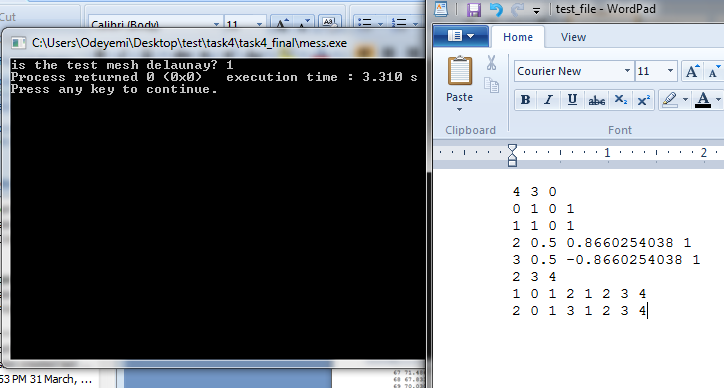
**Note:** Triangle **14 (in test file : triangulation#2.tri) had vertices 51,61 and 69** whose coordinates are highlighted. This is evidence the function works.



Testing for points outside of the domain of the triangulation . **(in test file : triangulation#2.tri)**

-1 represents no triangle.

**Test3 – Checking if a triangulation is Delaunay –** The check for Delaunay was achieved by checking the vertices of each triangle’s neighbor did not fall within the circumcircle. This approach was preferred over searching the whole array because it took a computational time of **O (n ) rather than O (n2 ).** It should be noted that all the triangulations files provided were checked and returned to be delaunay. To validate the correctness of the function, I used a test.tri file in which all triangles in the mesh were equilaterals. This mesh returned as a true delauanay. The figures below show the checks.



I ran into some rounding error problems which made some triangles that were illegal appear to be Delaunay. This was beyond the scope of the exercise and will be investigated in the future.

**Computing the Integral -** Computing the integral over the domain of the triangulation was achieved two ways as suggested in the project description. The functions *mesh\_calc\_f\_integral1* and *mesh\_calc\_f\_integral2* achieved this. Both took as their argument a functor representing the function which would be integrated over the domain.

Other functionalities were declared in the t\_mesh class such as – create\_a \_delaunay – these will implemented in the future and so their interface was only declared.

**PARALLELIZING THE CODE USING OPEMNMP**

This task involved parallelizing the code using openMP. The first step was to identify which of the functions and code components that could be parallelized. Table 5.1 below shows the list of functions parallelized and the time measured for each compared against the non-parallerized versions. The test conditions for each function were the same i.e. the same tri file was used to compare for the openmp and non-openMP codes. I ran the code on node002. The test code involved reading from the tri file to the triangulation data structure (in my case class t\_mesh ) and then calling the corresponding parallelized member function of which the time taken to compute the parallelized member function is measured and displayed on the screen. Further investigation was carried out on the number of threads and these are also shown in the table.

**Table 5.1 – Average time taken for different openmp settings for triangulation#2.tri file.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Parallelized member function** | **Average Time taken without openmp** | **Time taken when number of threads isn’t specified** | **Time taken when N=2 threads are specified to be used.** | **Time taken when N=4 threads are specified to be used.** | **Time taken when N=10 threads are specified to be used.** |
| **Find\_neighbour()** | **8.725 x 10-4 seconds** | 3.339 x 10-3 seconds | **5.80 x 10-4 seconds** | 3.112 x 10-3 seconds | 3.15 x10-3 seconds |
| **Is\_delaunay()** | **1.111 x 10-3 seconds** | 3.167 x 10-3 seconds | **8.76 x 10-4 seconds** | 3.14 x 10-3 seconds | 3.14 x 10-3 seconds |
| **Print\_neighbours()** | **2.4473 x 10-3 seconds** | 2.220 x 10-2 seconds | **3.59 x 10-3 seconds** | 1.7 x 10-2 seconds | 1.75 x 10-2 seconds |

Table 5.1 shows that speed up is achieved when the number of threads N=2, for the *find\_neighbour ()* and *is\_delaunay ()* member functions. No improvement is shown in the computational time for parallelized cases compared to the non-parallelized case for the *print\_neighbour()* , this is because the function prints to the screen. It was observed that the order of which the data is printed is random from the non-parallelized case. This demonstrates how each threads executes at a different time is from the others i.e. the order of work completion is different.

Increasing the number of threads degraded the computational time ( for N >2 ), this is because the size of the data set involved in the task. Parallel programming is only beneficial when used on large data sets.

I ran the same test on a different node (node 001) and found the above observation to be the same however the overall time was slightly shifted because it is the busier node.