Problem 1. Algorithm

(a) Describe the representation of a convex polyhedron. Show the representation of a tetrahedron.

Since a convex polyhedron can be interpreted as a planar graph, we can use a doubly-connected edge list to represent a convex polyhedron. One example of the representation of a tetrahedron by using a doubly-connected edge list is shown in Fig. 1.

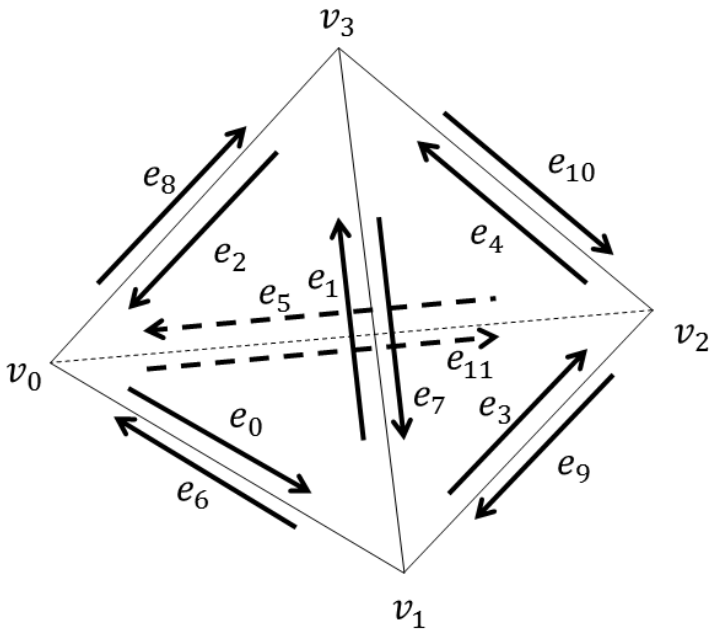

Figure 1. One example of the representation of a tetrahedron.

Around the each vertex, the half edge is ordered by the clockwise order. In the case of Fig. 1, edges e_0, e_{11}, e_8 is the clockwise order around the vertex v_0, for instance. The entire ordered list is shown in the Table. 1.

Table 1. The ordered list of edges around each vertex

Vertex	Ordered list of edges
v_0	e_0, e_{11}, e_8
v_1	e_1, e_3, e_6
v_2	e_4, e_5, e_9
v_3	e_2, e_{10}, e_7

(b) Explain how to represent the conflict graph by adding pointers to this representation.

The conflict graph is represented by adding the pointers of the visible faces to each unadded vertices and the pointers of the visible vertices to each hull faces. In this manner, each vertex can look up all the visible faces by just scanning these pointers, and each face can look up all the visible vertices by just scanning these pointers. The declaration of the classes Vertex and Face should be modified as shown in Fig. 2 and Fig. 3.

```
class Vertex {
 public:
  vector<Face*> visibleFaces;

  ...(snip)...
};
```

Figure 2. The declaration of class Vertex

```
class Face {
 public:
  vector<Vertex*> visibleVertices;
  ...(snip)...
};
```

Figure 3. The declaration of class Face

(c) Which vertices, edges, and faces are removed when a vertex is added to the hull?

When a vertex p_r is added, we want to remove all the visible faces from p_r. We also want to remove all the edges that constitute those faces except the horizon edges. For vertices, we do not explicitly remove the visible vertices, because those vertices are implicitly removed from the hull when the corresponding edges are removed.

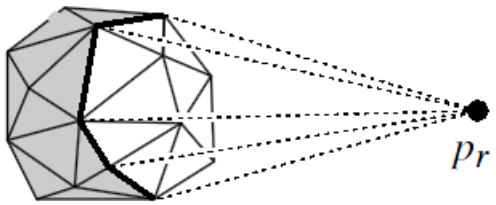

Figure 4. All the visible faces are to be removed when a vertex p_r is added.

To find the visible faces, we just need to look up the conflict graph. More precisely, $p_r \rightarrow$*visibleFaces* stores all the faces which are visible from p_r.

(d) How can one horizon edge be found? How can be the next one in clockwise order by found?

One horizon edge can be found by traversing all the constituting edges of the visible faces until one horizon edge is found. For each edge, we check whether the face of its twin edge is visible or not. If it is not visible, then this edge is a horizon edge, so we stop traversing.

```
FineOneHorizonEdge(vertex p_r)
1  For each face f_i ∈ p_r->visiblefaces
2    e = f_i->edge
3    g = e
4    do
5      If g->twin->face is not visible then
6        return g
7      Else if
8      g = g->twin->next
9    until g == e
10 End for
```

Figure 5. The pseudo code to find one horizon edge

Once we find the first horizon edge, we can get the entire horizon edges by the following process. Let p_r be the newly added vertex, e_0 be the first horizon edge we found, and g_0 be its twin edge. Then we traverse all the edges which go out from the tail of e_0 until we find the next horizon edge. The reason why this process can find the next horizon edge is very straightforward. If a vertex v is on the horizon, then there will be two edges outgoing from the vertex. This implies that if we keep traversing the next edges starting from a horizon edge e_0, we will eventually find the next horizon edge as shown in Fig. 6.

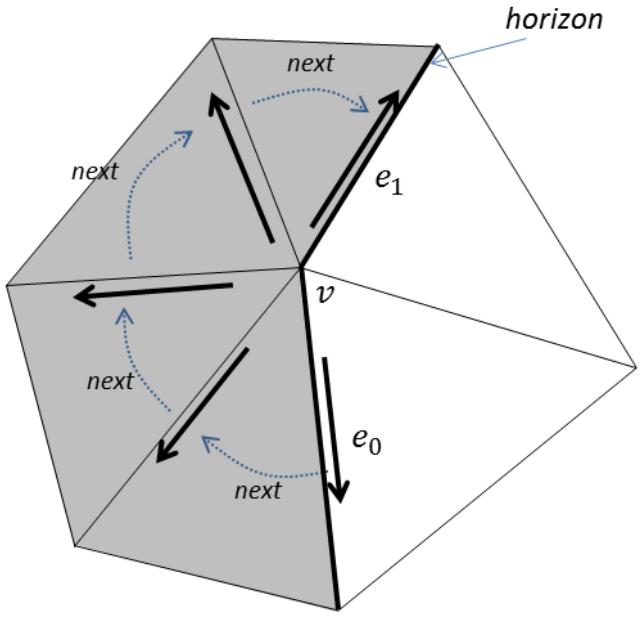

Figure 6. The next horizon edge e_1 can be found by traversing the next edges starting from the first horizon edge e_0.

The pseudo code of these steps are shown in Fig. 7.

```
FindNextHorizonEdge(vertex p_r, edge e_0)
1  g_0 = e_0->twin
2  e = e_0
3  while (true)
4     if e->twin->face is visible from p_r then
5        return e;
6     e = e->next
7  end while
```

Figure 7. The pseudo code to find the next horizon edge

(e) Which vertices, edges, and faces are added to the hull? Which are removed?

When a vertex p_r is added, we add edges that connect p_r with all the vertices on the horizon, and add the corresponding faces as shown in Fig. 8. When we add these edges and faces, we also have to maintain the representation of the convex hull, especially the clockwise order of edges around each vertex. Let p_i and p_j be the vertices on one of the horizon edges, h_{i1} be the edge from p_i to p_j, h_{i2} be the edge from p_j to p_i, e_{i1} be the edge from p_i to p_r, e_{i2} be the edge from p_r to p_i, e_{j1} be the edge from p_j to p_r, e_{j2} be the edge from p_r to p_j, and f_i be the face surrounded by the edge h_{i1}, e_{j1}, and e_{i2}. Then, the order of the edges will be updated as follows:

- $e_{i1} \rightarrow next = h_{i1}$
- $e_{j2} \rightarrow next = e_{i2}$
- $h_{i2} \rightarrow next = e_{j1}$

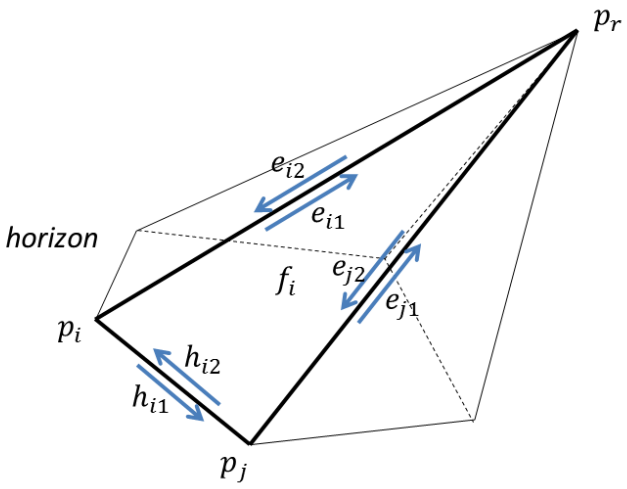

Figure 8. After all the visible faces are removed, edges that connect horizon with the newly added vertex p_r and the corresponding faces are to be added, and these edges are to be ordered in counter-clockwise order around p_r.

For vertices, since all the vertices are added to the arrangement at the beginning, we do not need to add the vertex any more.

(f) How does the conflict graph change?

First, we list up all the vertices that are visible from faces that are adjacent to the horizon edges. These vertices are to be used for the visibility test for the newly added faces later. Since we also want to remove the newly added vertex, we exclude this vertex from the list. Next, we remove all the nodes that correspond to the visible faces, and remove all the outgoing edges from those nodes. Since we maintain the conflict graph by the pointers as I explained in (b), we do not explicitly store the edges of the conflict graph. Instead, we remove all the visible faces from the arrangement and from the pointers of the vertices that are visible from those visible faces. Then, we want to remove the newly added vertex from the conflict graph, but we do not do anything for this, because the newly added vertex does not have any edges any more. Lastly, for all the newly added faces, we check if they are visible from the vertices that were listed up in the first step. If a face is visible from a vertex, the vertex is added to the pointers of the face, and the face is added to the pointers of the vertex. The pseudo code of these steps is shown in Fig. 9.

```
UpdateConflictGraph
1  List up all the vertices that are visible from faces that are adjacent to
the horizon edges.
2  For each visible face f_i
3    For each vertex v_j ∈ f_i->visibleVertices
4      v_j->removeFace(f_i)
5    End for
6    arr->removeFace(f_i)
7  End for
8  For each vertex v_i ∈ list retrieved in line 1
9    For each newly added face f_i
10     If f_i is visible from v_i Then
11       v_i->visibleFaces.push_back(f_i)
12       f_i->visibleVertices.push_back(v_i)
13   End for
14 End for
```

Figure 9. The pseudo code of updating the conflict graph

Problem 2. Implementation

Please type the following command to compile and execute my implementation:

```
$ make

$ ./ps3-nishida-1
```

One of the test data that I used to test is as follows:

```
$ ./ps3-nishida-1
```

```
10 0 0 0 1 1 1 5 5 10 5 5 -5 10 10 0 8 8 -1 0 10 0 7 3 1 5 5 -10 10 0 0
```

Then, the program outputs the following line.

```
8 9 0 8 2 0 9 9 8 4 2 9 4 0 2 6 2 4 6 4 8 6 8 0 6
```

The resulting convex hull is shown in Fig. 10, which was visualized by *ParaView*.

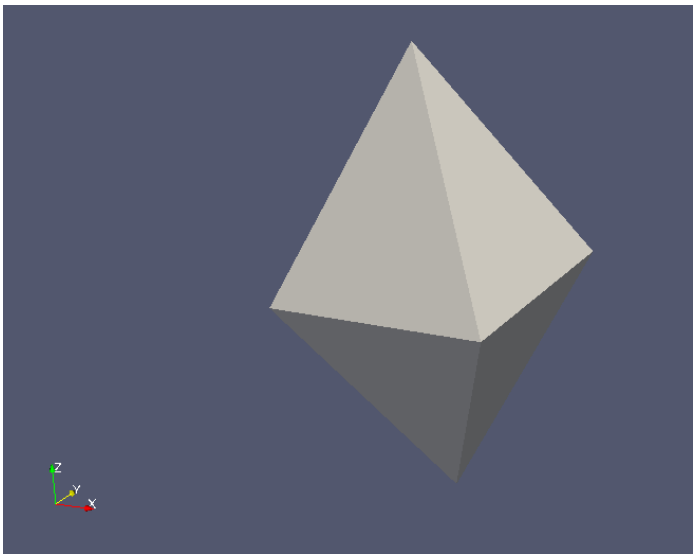

Figure 10. The resulting convex hull is visualized by *ParaView*.

Problem 3. Delaunay Triangulation

Please type the following command to compile and execute my implementation:

```
$ make

$ ./ps3-nishida-2
```

One of the test data that I used to test is as follows:

```
$ ./ps3-nishida-2

6 0 0 5 -5 10 5 5 5 10 15 0 10
```

Then, the program outputs the following line.

```
5 1 3 0 2 3 1 3 5 0 2 4 3 3 4 5
```

The resulting Delaunay triangulation is shown in Fig. 11.

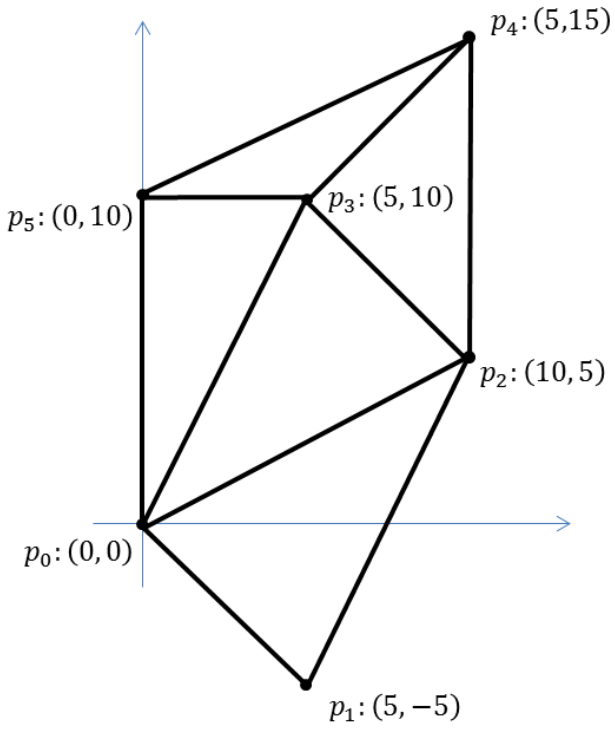

Figure 11. The result of the Delaunay triangulation of the test data.