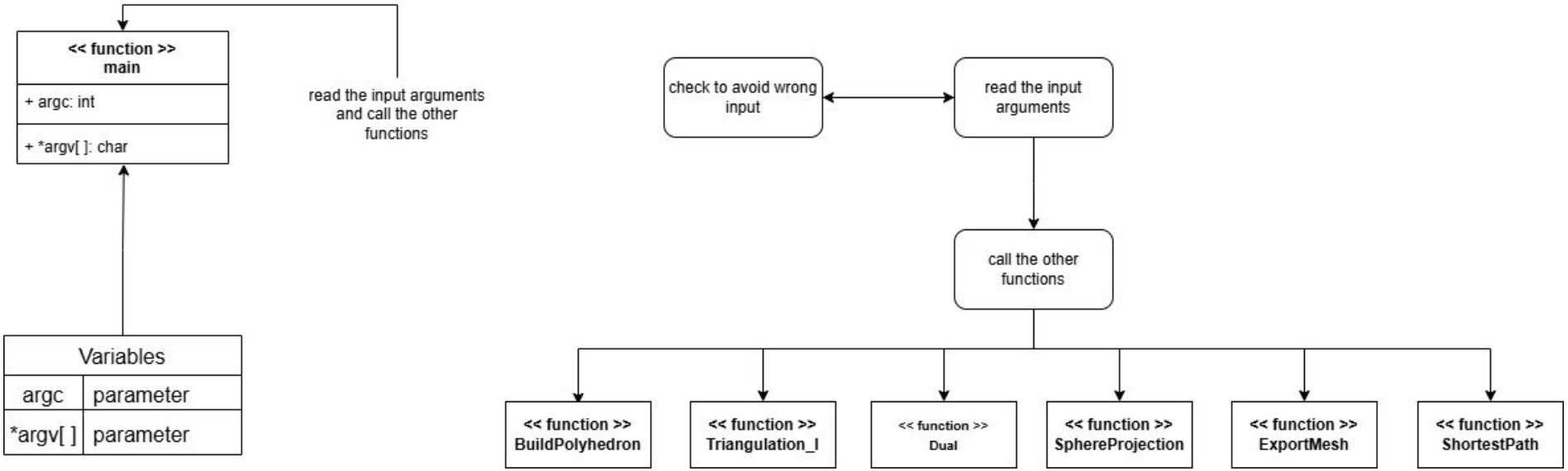
Progetto PCS 2025

Abbrugiati Filippo
Donadio Gabriele
Giraudo Alessandro

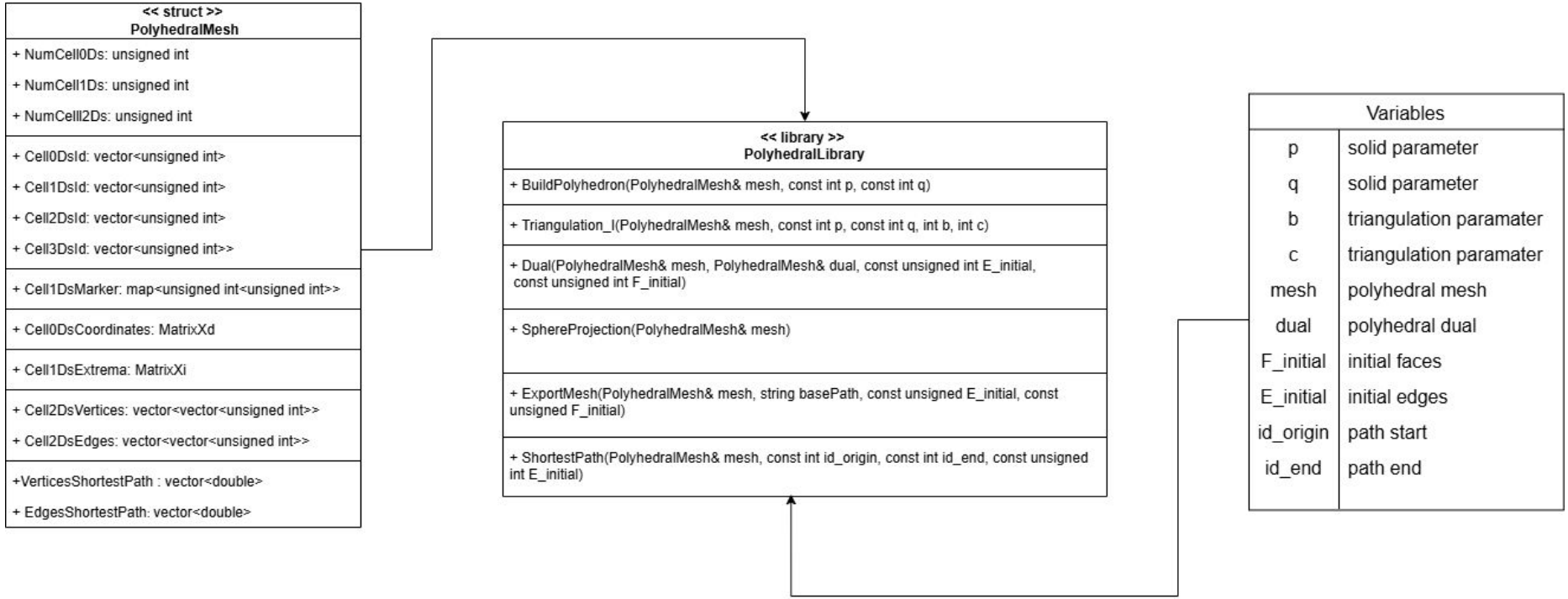
Politecnico di Torino

main.cpp

main

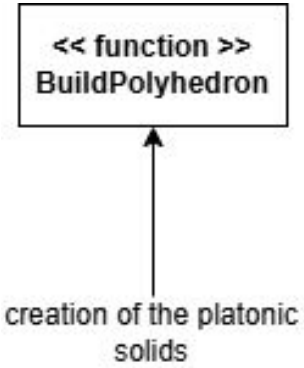


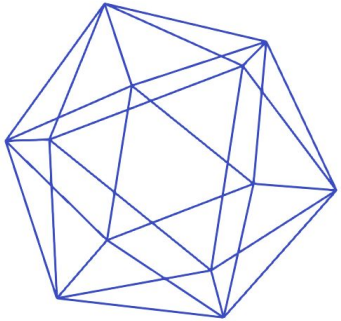
PolyhedralMesh.hpp and Utils.hpp



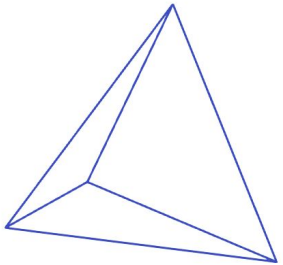
Utils.cpp

BuildPolyhedron

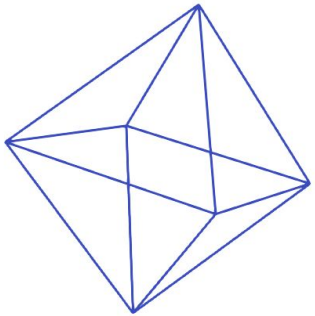




Icosahedron{3,5}

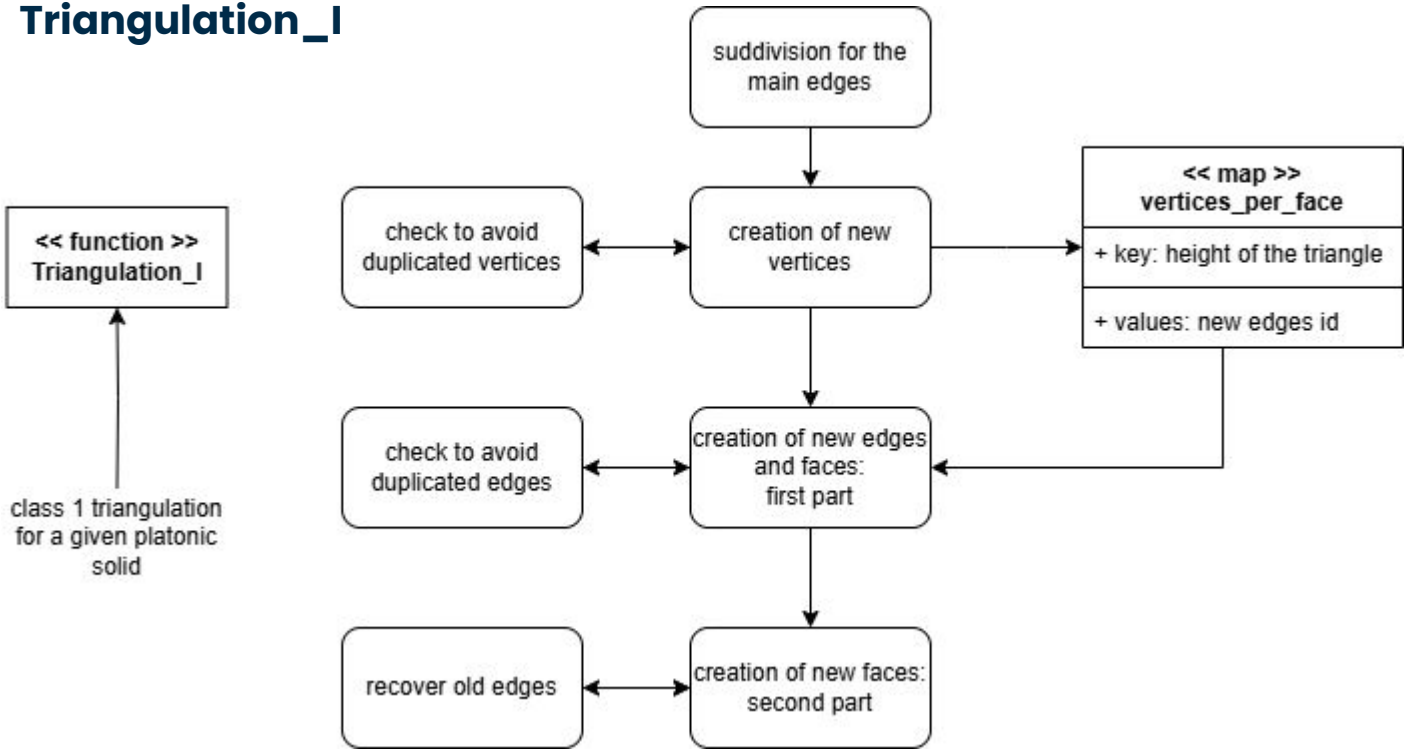


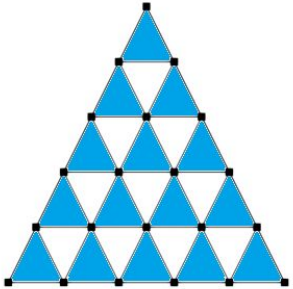
Tetrahedron {3,3}



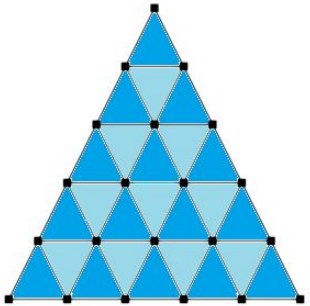
Octahedron {3,4}

Triangulation_I





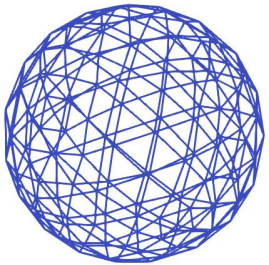
creation of new edges and faces: part 1



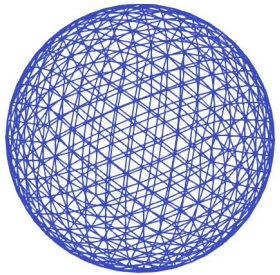
creation of new faces: part 2

```
for(unsigned int face = 0; face < mesh.NumCell2Ds; face++)
{

    // salvare le 3 direzioni del triangolo
    Eigen::Matrix3d matrix_edges;

    for (auto& vec : id_vertices_suddivisione) {
        vec.clear();
    }
    for (auto& vec : id_vertices_suddivisione) {
    vec.resize(b-1);
    }
    vertices_per_face.clear();

    for(unsigned int vertex = 0; vertex < 3; vertex++)
    {
        // DIVIDO IL LATO
        unsigned int vertex_origin_ID = mesh.Cell2DsVertices[face][vertex];
        unsigned int vertex_end_ID = mesh.Cell2DsVertices[face][(vertex+1)%3];

        double x_origin = mesh.Cell0DsCoordinates(vertex_origin_ID, 0);
        double y_origin = mesh.Cell0DsCoordinates(vertex_origin_ID, 1);
        double z_origin = mesh.Cell0DsCoordinates(vertex_origin_ID, 2);

        double x_end = mesh.Cell0DsCoordinates(vertex_end_ID, 0);
        double y_end = mesh.Cell0DsCoordinates(vertex_end_ID, 1);
        double z_end = mesh.Cell0DsCoordinates(vertex_end_ID, 2);

        Eigen::Vector3d vector_edge;
        vector_edge << x_end - x_origin,
                       y_end - y_origin,
                       z_end - z_origin;
        vector_edge /= b; // vettore direzione normalizzato
        matrix_edges.row(vertex) = vector_edge;
        // Creo i punti della triangolazione sui lati principali
        // Cerco il lato di ID Cell2DsEdges[face][vertex] nel vettore associato al marker 2
        auto iter_2DMark = find(mesh.Cell1DsMarker[2].begin(), mesh.Cell1DsMarker[2].end(), mesh.Cell2DsEdges[face][vertex]);
        if (iter_2DMark != mesh.Cell1DsMarker[2].end()){
            for(unsigned int i = 1; i < b; i++)
            {
                double x_sudd = x_origin + vector_edge(0)*i;
                double y_sudd = y_origin + vector_edge(1)*i;
                double z_sudd = z_origin + vector_edge(2)*i;

                bool found = false;
                unsigned int ind = 0;

                // recupero gli id dei vertici della suddivisone dei lati principali
                unsigned int id_found;
                while(not found){
                    if(abs(mesh.Cell0DsCoordinates(ind,0)-x_sudd) < 1e-12 && abs(mesh.Cell0DsCoordinates(ind,1)-y_sudd)
                        < 1e-12 && abs(mesh.Cell0DsCoordinates(ind,2)-z_sudd) < 1e-12) {
                        id_found = mesh.Cell0DsId[ind];
                        found = true;
                    }
                    ind++;
                }
                id_vertices_suddivisione[vertex][i-1] = id_found;
```

```
            }
        }
        else{
            for(unsigned int i = 1; i < b; i++)
            {
                mesh.Cell0DsCoordinates(n, 0) = x_origin + vector_edge(0)*i;
                mesh.Cell0DsCoordinates(n, 1) = y_origin + vector_edge(1)*i;
                mesh.Cell0DsCoordinates(n, 2) = z_origin + vector_edge(2)*i;
                mesh.Cell0DsId.push_back(n);
                id_vertices_suddivisione[vertex][i-1] = n;
                n++;
            }
            mesh.Cell1DsMarker[2].push_back(mesh.Cell2DsEdges[face][vertex]);
        }

    }

    // popolo il livello 0 della mappa con tutti i vertici della faccia
    vertices_per_face[0].reserve(b+1);
    vertices_per_face[0].push_back(mesh.Cell2DsVertices[face][0]);
    for(const auto& iter : id_vertices_suddivisione[0]) {
        vertices_per_face[0].push_back(iter);
    }
    vertices_per_face[0].push_back(mesh.Cell2DsVertices[face][1]);

    // trovo i vertici interni della triangolazione
    // itero sull'altezza
    for(unsigned int j = 0; j < b-1 ; j++)
    {
        vertices_per_face[j+1].reserve(b-j);
        vertices_per_face[j+1].push_back(id_vertices_suddivisione[2][b-2-j]);
        // itero sulle righe
        for(unsigned int k = b-2-j; k > 0; k--)
        {
            mesh.Cell0DsCoordinates(n, 0) = mesh.Cell0DsCoordinates(id_vertices_suddivisione[1][j], 0) - matrix_edges(0, 0)*k;
            mesh.Cell0DsCoordinates(n, 1) = mesh.Cell0DsCoordinates(id_vertices_suddivisione[1][j], 1) - matrix_edges(0, 1)*k;
            mesh.Cell0DsCoordinates(n, 2) = mesh.Cell0DsCoordinates(id_vertices_suddivisione[1][j], 2) - matrix_edges(0, 2)*k;
            mesh.Cell0DsId.push_back(n);
            vertices_per_face[j+1].push_back(n);
            n++;
        }

        vertices_per_face[j+1].push_back(id_vertices_suddivisione[1][j]);
    }

    vertices_per_face[b].reserve(1);
    vertices_per_face[b].push_back(mesh.Cell2DsVertices[face][2]);
```

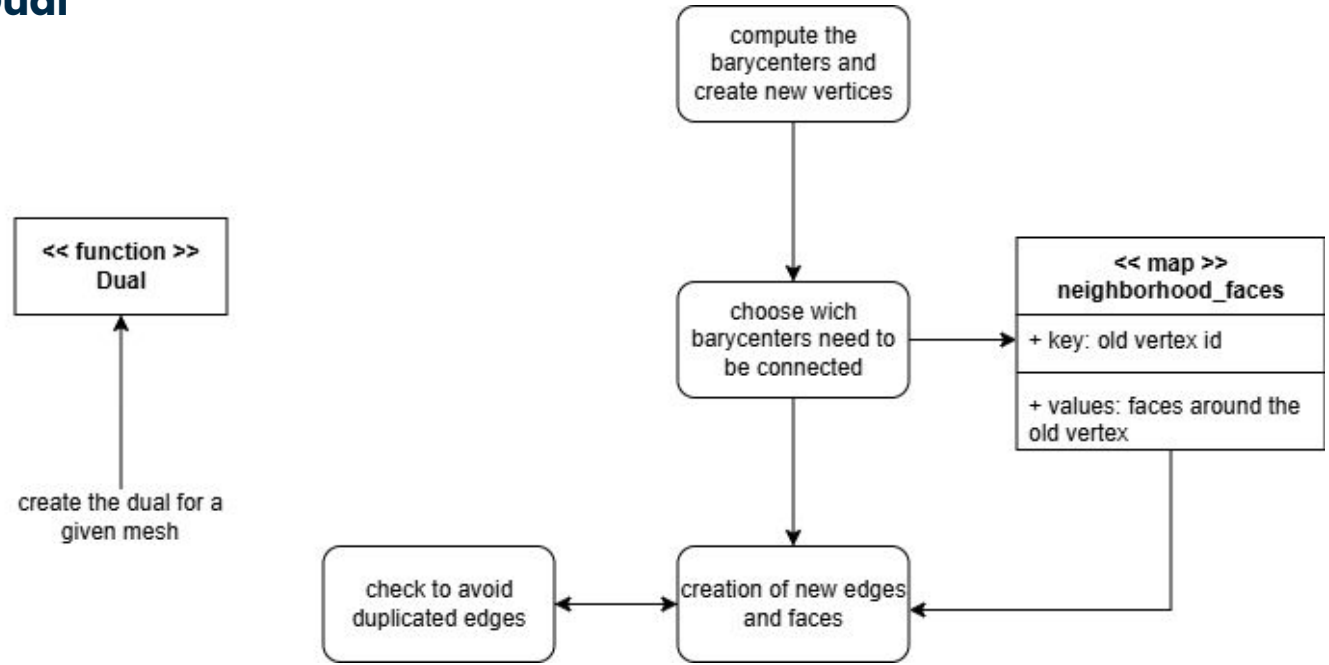


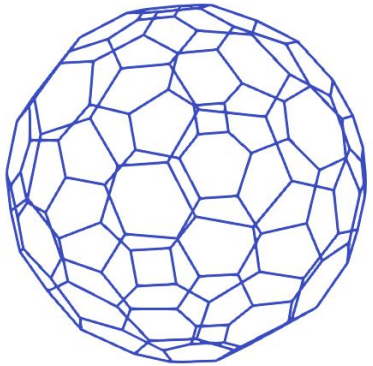
(p,q,b,c)=(3,4,6,0)



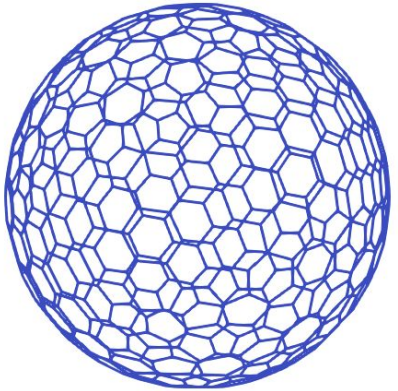
(p,q,b,c)=(3,5,0,8)

Dual



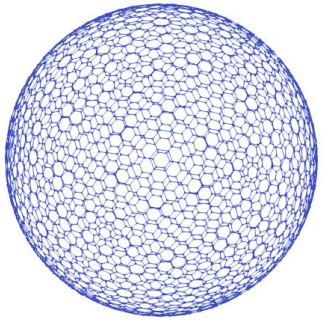


(p,q,b,c)=(4,3,4,0)

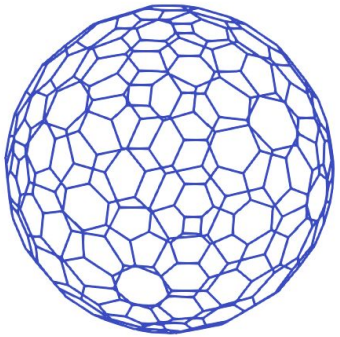


(p,q,b,c)=(5,3,5,0)

```
// Visitiamo le facce del vicinato evitando le ripetizioni e salviamo gli ID dei vertici della faccia
for(unsigned int face = 0; face < neighborhood_faces[vertex].size(); face++)
{
    vector<unsigned int> edges_face = mesh.Cell2DsEdges[iter_face];

    bool found = false;
    for(const auto& iter_face_ad: neighborhood_faces[vertex])
    {
        if(found)
            break;
        if(iter_face_ad == iter_face || iter_face_ad == id_past)
            continue;

        vector<unsigned int> edges_face_ad = mesh.Cell2DsEdges[iter_face_ad];

        // Controlliamo se un lato della faccia è in comune, per determinare la faccia adiacente
        for(const auto& it: edges_face_ad)
        {
            auto iter_edges = find(edges_face.begin(), edges_face.end(), it);
            if(iter_edges!=edges_face.end())
            {
                found = true;
                new_face_ad = iter_face_ad;
                if(face < neighborhood_faces[vertex].size()-1)
                    vertices.push_back(new_face_ad-F_initial);
            }
        }
    }

    bool find = false;
    unsigned int edge_0 = m;
    unsigned int vert_0 = iter_face-F_initial;
    unsigned int vert_1 = new_face_ad-F_initial;
    for(unsigned int iter = 0; iter < dual.Cell1DsId.size(); iter++)
    {
        // cerchiamo se esiste il lato con estremi vert_0 e vert_1
        if((dual.Cell1DsExtrema(iter, 0) == vert_0 && dual.Cell1DsExtrema(iter , 1) == vert_1) ||
           (dual.Cell1DsExtrema(iter, 0) == vert_1 && dual.Cell1DsExtrema(iter, 1) == vert_0))
        {
            edge_0 = dual.Cell1DsId[iter];
            find = true;
            break;
        }
    }

    if(not find)
    {
        dual.Cell1DsId.push_back(edge_0);
        // baricentro di faccia iter_face è iter_face-F_initial
        dual.Cell1DsExtrema(m, 0) = vert_0;
        dual.Cell1DsExtrema(m, 1) = vert_1;
        edge_0 = m;
        m++;
    }
    edges.push_back(edge_0);
    id_past = iter_face;
    iter_face = new_face_ad;
}
```

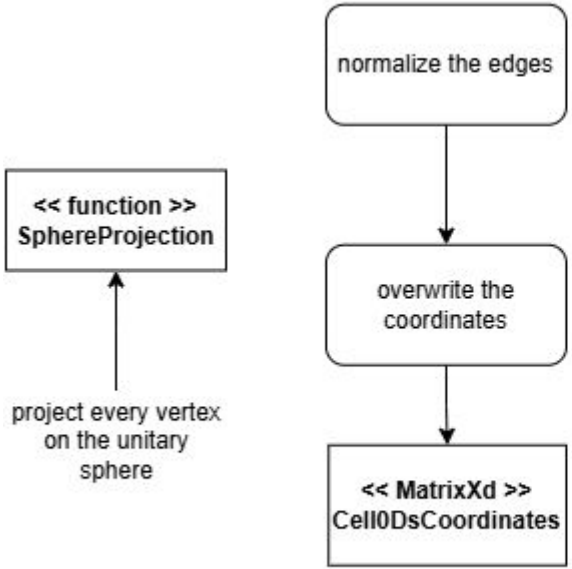


(p,q,b,c)=(5,3,12,0)

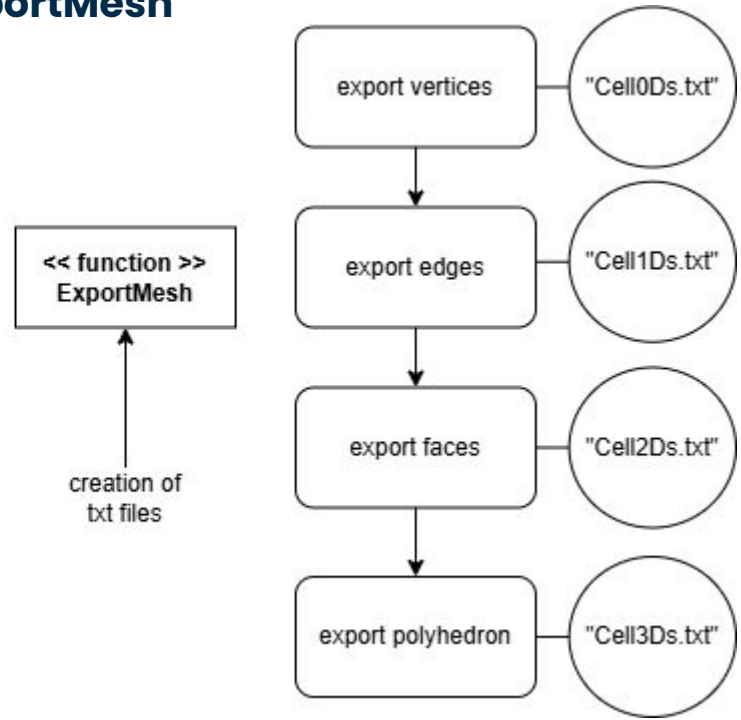


(p,q,b,c)=(4,3,0,6)

SphereProjection

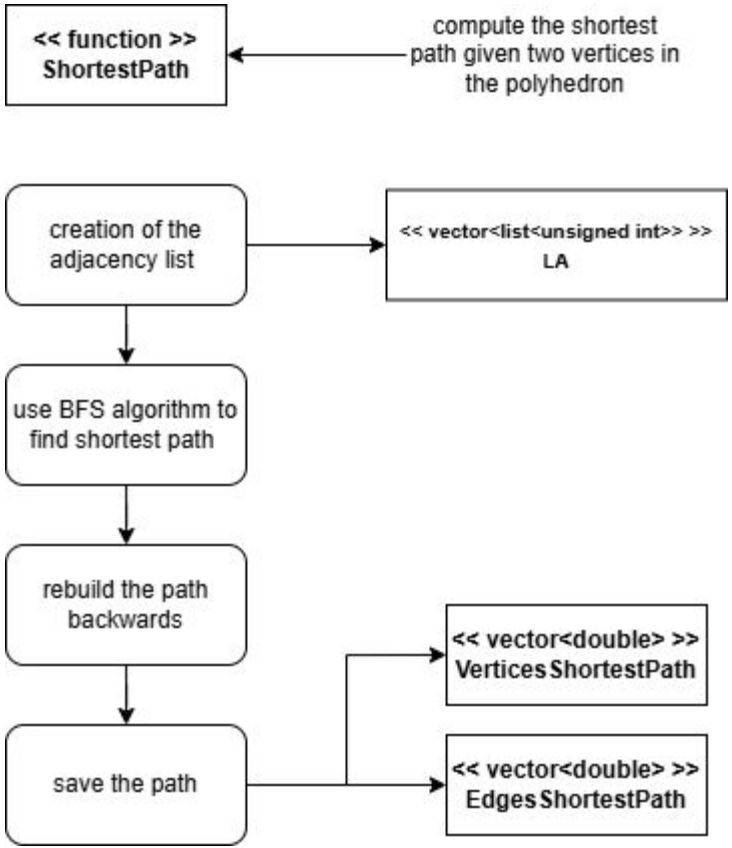


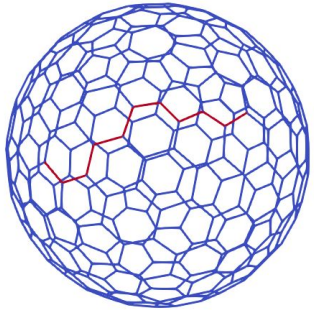
ExportMesh



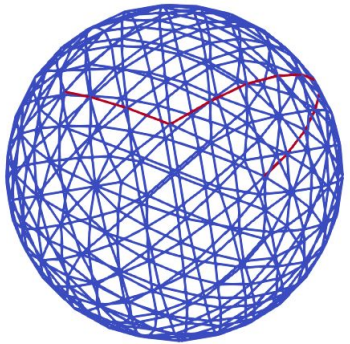
ShortestPath

```cpp
// BFS
vector<bool> reached(n);
// vettore dei predecessori per la ricostruzione del cammino minimo
vector<unsigned int> pred(n);
queue<unsigned int> Q;
for(unsigned int i = 0; i < n; i++)
{
    reached[i] = false;
    pred[i] = -1;
}
Q.push(id_origin);
pred[id_origin] = id_origin;
while(not Q.empty() && not reached[id_end])
{
    unsigned int u = Q.front();
    Q.pop();
    reached[u] = true;
    for(const auto& w: LA[u])
    {
        if(not reached[w])
        {
            reached[w] = true;
            Q.push(w);
            pred[w] = u;
        }
    }
}

// ricostruisco il percorso a ritroso
// lista per poter fare push_front
list<unsigned int> path;
path.push_front(id_end);
unsigned int new_id_end = id_end;
bool found = false;
while(not found)
{
    path.push_front(pred[new_id_end]);
    new_id_end = pred[new_id_end];

    if(new_id_end == id_origin)
        found = true;
}
```





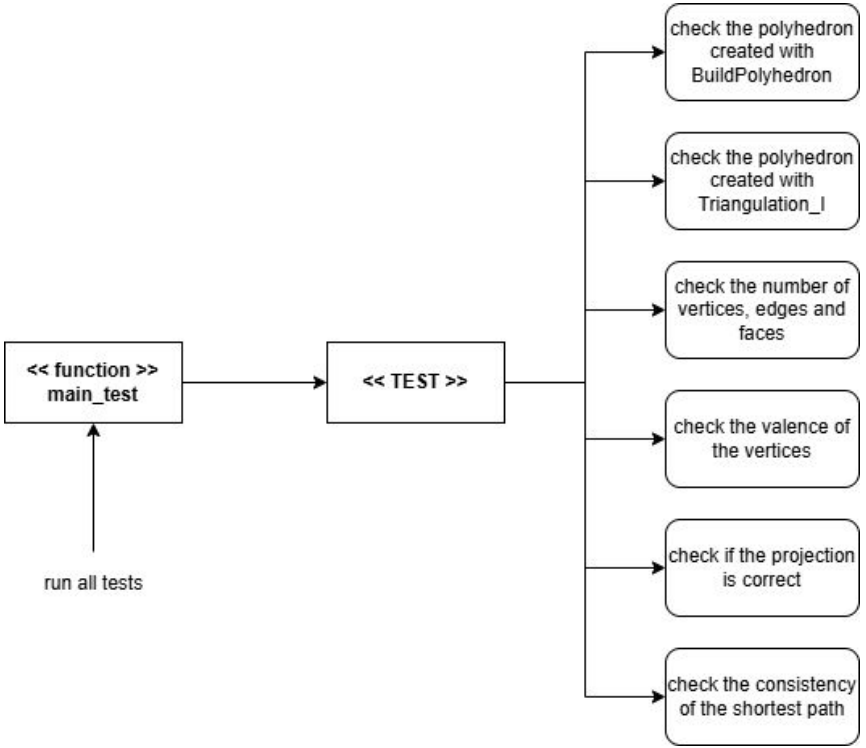
(5,3,4,0,2,100)



(3,5,0,5,2,170)

main_test.cpp and TestPolyhedron.cpp

TEST




```cpp
// test sulla coerenza dei segmenti e dei vertici dello shortest path
TEST(PolyhedralMeshTest, CheckShortestPath)
{
    PolyhedralMesh mesh;
    BuildPolyhedron(mesh, 3, 4);
    Triangulation_I(mesh, 3, 4, 5, 0);
    unsigned int E_initial = 12;
    ShortestPath(mesh, 0, 42, E_initial);
    bool ok = true;
    unsigned int NumVertices = 0;
    unsigned int NumEdges = 0;
    vector<unsigned int> vertices;
    vector<unsigned int> edges;
    vertices.reserve(mesh.NumCell0Ds);
    edges.reserve(mesh.NumCell0Ds); // ci dovranno essere vertices.size() - 1 edges
    for(unsigned int i=0; i<mesh.NumCell0Ds; i++)
    {
        if(mesh.VerticesShortestPath[i]==1)
        {
            NumVertices++;
            vertices.push_back(i);
        }
    }
    for(unsigned int j=0; j<mesh.NumCell1Ds-E_initial; j++)
    {
        if(mesh.EdgesShortestPath[j]==1)
        {
            NumEdges++;
            edges.push_back(j+E_initial);
        }
    }
    if(NumVertices!=(NumEdges+1))
        ok = false;

    // se è già falso da prima, inutile fare altri controlli
    if(ok)
    {
        for(const auto& vertex: vertices)
        {
            if(ok)
            {
                unsigned int count = 0;

                // verifichiamo in quanti lati di edges compare l'estremo vertex
                for(const auto& edge: edges)
                {
                    if(mesh.Cell1DsExtrema(edge, 0) == vertex || mesh.Cell1DsExtrema(edge, 1) == vertex)
                        count++;
                }

                if(count < 1 && (vertex == 0 || vertex == 42))
                    ok = false;
                else if(count < 2 && (vertex != 0 && vertex != 42))
                    ok = false;
            }
        }
    }

    ASSERT_EQ(ok,true);
}
```