



# Presentazione ProgettoPCS2025

Bazzanella Alice,  
Scaglione Francesca,  
Tomatis Elisa.



## Punti principali

Nel nostro progetto abbiamo:

- definito i **solidi geodetici di classe I**;
- scritto su files i risultati ottenuti;
- calcolato il **cammino minimo**;
- costruito i **solidi geodetici di classe II**;
- verificato, per ogni unità logica, il corretto funzionamento utilizzando i GoogleTest.



## Struttura del codice

Cartelle e files creati

Per lavorare al progetto abbiamo suddiviso i files creati all'interno di cartelle:

- **src:** contiene i files con le dichiarazioni di funzioni per la creazione, triangolazione, ottimizzazione ed analisi dei poliedri;
- **src\_test:** contiene il file **Test.hpp** dove all'interno sono implementati i google test richiesti;
- **export\_paraview:** contiene il file **Export\_Paraview.cpp** per l'esportazione su Paraview.



# Definizione dei poliedri

Solidi Geodetici di classe I

Un poliedro geodetico è un poliedro convesso definito da facce triangolari che possiamo rappresentare tramite il **simbolo di Schläfli**  $\{3, q+\}_{b,c}$ .

- **p** rappresenta il numero di vertici del poligono che si osserva guardando ciascuna faccia;
- **q** rappresenta il numero di vertici del poligono che si osserva guardando ciascun vertice ovvero la valenza;
- **b, c** sono parametri utilizzati per la triangolazione delle facce.



## Definizione dei poliedri

Solidi Geodetici di classe I

Per lavorare con i poliedri abbiamo usato la **libreria PolyhedralLibrary** definita in **Utils.hpp**.

Nel file **Polyhedra.cpp** abbiamo costruito i poliedri, ciascuno è caratterizzato da:

- **CelloDsCoordinates**: matrice  $3 \times n$  di coordinate dei vertici normalizzate;
- **Cell1DsExtrema**: matrice  $n \times 2$  contenente gli estremi dei lati;
- **Cell2DsVertices**: vettore di liste con gli id dei vertici delle facce;
- **Cell2DsEdges**: vettore di liste con gli id dei lati di ciascuna faccia;
- **Cell3DsVertices**, **Cell3DsEdges**, **Cell3DsFaces** per il volume finale.



# Inizializzazione della mesh triangolata

Solidi Geodetici di classe I

All'interno del file **Dimension.cpp** ci sono le funzioni per gestire la mesh sia prima che dopo la triangolazione:

- **ComputePolyhedronVEF( $q, b, c$ )**: calcola il numero atteso di vertici, spigoli e facce tramite le formule date. Ci permette di inizializzare correttamente la mesh triangolata;
- **CalculateDuplicated( $q, b, c, dimension$ )**: stima il numero di vertici e spigoli duplicati generati durante la triangolazione.



```
vector<int> ComputePolyhedronVEF(int q,  
    int b, int c)  
{  
    vector<int> result(3); //  
        inizializza un vettore con 3  
        valori, tutti -1  
  
    int T = 0;  
    T = b * b + b * c + c * c;  
    int V = 0;  
    int E = 0;  
    int F = 0;  
  
    if (q == 3) {  
        V = 2 * T + 2;  
        E = 6 * T;  
        F = 4 * T;  
    }
```

```
    else if (q == 4) {  
        V = 4 * T + 2;  
        E = 12 * T;  
        F = 8 * T;  
    }  
    else {  
        V = 10 * T + 2;  
        E = 30 * T;  
        F = 20 * T;  
    }  
    result[0] = V; // Primo elemento:  
        V (vertici)  
    result[1] = E; // Secondo elemento  
        : E (spigoli)  
    result[2] = F; // Terzo elemento:  
        F (facce)  
  
    return result;}
```



# Triangolazione

Solidi Geodetici di classe I

Nel caso  $b = 0$ ,  $c \geq 1$  oppure  $b \geq 1$ ,  $c = 0$  abbiamo scritto il file **Triangulation1.cpp** in cui l'obiettivo é suddividere le facce della mesh in triangoli secondo il livello  $\text{subdivisionLevel} = b + c$ .

Il file contiene le funzioni:

- **FindAddEdge**: verifica se un edge è già stato inserito altrimenti lo aggiunge;
- **triangulateAndStore**: alloca spazio per le strutture e calcola tramite interpolazione i nuovi vertici;
- **PopulateCell3D**: raggruppa le celle 0D, 1D, 2D per la cella 3D. Aggiunge dei flag per distinguere gli elementi interni ed esterni.

Alla fine si ottiene una nuova mesh chiamata **meshTriangulated**.





```
void FindAddEdge(
    int a, int b,
    PolyhedralMesh& meshTriangulated,
    unsigned int& k2,
    unsigned int k3)
{
    bool found = false;

    for (unsigned int i = 0; i <= k2;
        ++i) {
        if ((meshTriangulated.
            Cell1DsExtrema(i, 0) == a
            &&
            meshTriangulated.
                Cell1DsExtrema(i, 1)
                == b) ||
            (meshTriangulated.Cell1DsExtrema(i,
                0) == b &&
            meshTriangulated.
                Cell1DsExtrema(i, 1)
                == a)) {

            // Edge già presente
            meshTriangulated.
                Cell2DsEdges[k3].
                push_back(i);
            found = true;
            break;}
    }
}
```



# Gestione della mesh triangolata

Solidi Geodetici di classe I

Una volta creata la mesh triangolata bisogna rimuovere i duplicati formati per questo sempre all'interno del file **Dimension.cpp** ci sono le funzioni:

- **RemoveDuplicatedVertices:** tramite la struttura dati **Union-Find** trova tutti i vertici duplicati e aggiorna la mesh con i vertici unificati;
- **RemoveDuplicatedEdges:** con la stessa logica della funzione precedente vengono rimossi gli spigoli duplicati e si aggiornano gli id dei lati.



```
if ((meshTriangulated.  
    Cell0DsCoordinates.col(i) -  
    meshTriangulated.  
        Cell0DsCoordinates.col(j)).  
    norm() < tol) {  
    // Trovato un duplicato:  
    reindirizzamento tramite Union-  
    Find  
    unsigned int root_i = i;  
    while (id_remap[root_i] != root_i)  
        root_i = id_remap[root_i];
```

```
    unsigned int root_j = j;  
    while (id_remap[root_j] != root_j)  
        root_j = id_remap[root_j];  
    if (root_i != root_j)  
        id_remap[root_i] = root_j;  
  
    id_remap[i] = root_j;  
    meshTriangulated.Cell0DsCoordinates  
        .col(i) =  
        meshTriangulated.  
            Cell0DsCoordinates.col(j);
```



# Triangolazione



$$(p \ q \ b \ c) = (3 \ 5 \ 4 \ 0)$$



## Scrittura su files

Solidi Geodetici di classe I

All'interno del main.cpp prendiamo in input una quadrupla di numeri interi  $\{p, q, b, 0\}$  oppure  $\{p, q, 0, b\}$ . Nel caso in cui  $p = 3$  viene restituito il poliedro geodetico di classe I corrispondente con 4 file .txt:

- **CelloDs.txt:** contiene l'id di ogni vertice e le sue coordinate nello spazio;
- **Cell1Ds.txt:** contiene l'id di ogni lato e il vertice di inizio e di fine;
- **Cell2Ds.txt:** contiene gli id di ogni faccia, il numero di vertici e i loro id, i numeri di lati e i loro id;
- **Cell3Ds.txt:** generando un poliedro alla volta contiene il suo id, i vertici, i lati e le facce.



## Duale

Poliedri di Goldberg

A partire da un solido geodetico di classe I, scambiando il ruolo di vertici e facce, è possibile ottenere il suo duale detto **Poliedro di Goldberg generalizzato** indicato con  $\{q+, p\}_{b,c}$ .

Per farlo a partire dalla **meshTriangulated** abbiamo scritto il file **Dual.cpp**.

- **Creazione dei vertici:** vengono calcolati i baricentri delle facce del poliedro originale in quanto vertici del poliedro duale. L'id del vertice è lo stesso della faccia da cui proviene;
- **creazione degli spigoli:** viene creata una mappa che per ogni spigolo originale elenca le facce che lo condividono. Si uniscono i baricentri delle facce condivise;



```
// Mappa Vertice Originale -> Facce che  
    lo Contengono
```

```
map<unsigned int, vector<unsigned int>>  
    buildVertexToFacesMap(const  
        PolyhedralMesh& meshTriangulated) {  
    map<unsigned int, vector<unsigned  
        int>> vertexToFaces;  
    for (unsigned int faceId = 0;  
        faceId < meshTriangulated.  
            Cell2DsId.size(); ++faceId) {  
        }  
    }
```

```
        for (unsigned int  
            vertexOriginalId :  
                meshTriangulated.  
                    Cell2DsVertices[faceId]) {  
            vertexToFaces[  
                vertexOriginalId].  
                push_back(faceId);  
            }  
        }  
    return vertexToFaces;  
}
```



## Duale

Poliedri di Goldberg

- **creazione delle facce:** per ogni vertice originale si trovano le facce incidenti. I baricentri di queste facce generano una faccia del duale. I baricentri vengono raccolti visitando le facce attorno ad un vertice originale.
- vengono gestiti **errori topologici**;
- tutti i vertici vengono proiettati su una **sfera unitaria**.

Alla fine si ottiene una nuova mesh chiamata **meshDual**.

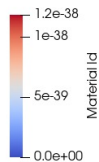
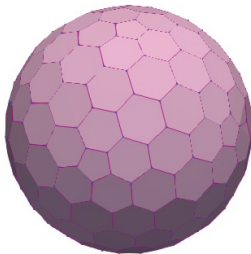




```
void ProjectMeshToUnitSphere(
    PolyhedralMesh& meshTriangulated) {
    for (int i = 0; i <
        meshTriangulated.
            Cell0DsCoordinates.cols(); ++i)
    {
        Vector3d vertexCoords =
            meshTriangulated.
                Cell0DsCoordinates.col(i);
        double norm = vertexCoords.norm
            (); // Equivalente a std::
               sqrt(vertexCoords.
                  squaredNorm()));
        if (norm < 1e-12) {
            cerr << "Warning: Vertice "
                << i << " troppo
                    vicino all'origine. Non
                        proiettato." << endl;
            continue; // Salta la
                     proiezione per questo
                     vertice
        }
        meshTriangulated.
            Cell0DsCoordinates.col(i) =
                vertexCoords / norm;}
    }
```



# Duale



$$(p \ q \ b \ c) = (5 \ 3 \ 4 \ 0)$$





## Cammino minimo

Prendendo in input una 6-tupla di numeri interi ( $p, q, b, c, id\_vertice\_1, id\_vertice\_2$ ) abbiamo trovato un cammino minimo che unisce i vertici contrassegnati da  $id\_vertice\_1$  e  $id\_vertice\_2$ . Per farlo abbiamo scritto il file **CamminoMinimo.cpp**:

- la funzione **calculateDistanceById** calcola la distanza euclidea fra due vertici della mesh dato il loro ID;
- nella struttura **ShortestPathResult** viene salvato il numero di lati del cammino e la lunghezza totale del cammino;
- la funzione **calculateAdjacencyMatrix** crea la matrice di adiacenza dei vertici della mesh;



```
double calculateDistanceById(const
    PolyhedralMesh& mesh, unsigned int
    id1, unsigned int id2) {

    VectorXd p1 = mesh.
        Cell0DsCoordinates.col(id1);
    VectorXd p2 = mesh.
        Cell0DsCoordinates.col(id2);

    return (p1 - p2).norm(); // Calcola
        la norma (distanza euclidea)
}
```

```
// Funzione per calcolare la matrice di
    adiacenza
for (unsigned int i = 0; i < mesh.
    Cell1DsId.size(); ++i) {
    unsigned int v1 = mesh.
        Cell1DsExtrema(i, 0);
    unsigned int v2 = mesh.
        Cell2DsExtrema(i, 1);

    adjMatrix(v1, v2) = 1;
    adjMatrix(v2, v1) = 1;
}
```



- La funzione **findShortestPathDijkstra** implementa l'algoritmo per trovare il cammino più breve fra due vertici. Vengono visitati i vertici vicini ed aggiornate le distanze ed i predecessori quando viene trovato un cammino più corto.

```
// Variabili Dijkstra
vector<double> dist(numVertices,
    numeric_limits<double>::infinity())

vector<unsigned int> predVertex(
    numVertices, -1)

vector<unsigned int> predEdge(
    numVertices, -1)

vector<bool> visited(numVertices, false)

using QueueElem = pair<double, unsigned
    int>;
priority_queue<QueueElem, vector<
    QueueElem>, greater<>> pq;
```

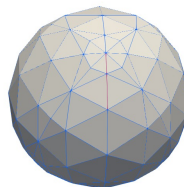


# Cammino minimo

```
appuser@THIS_IS_PCS_DOCKER:~/Data/ProgettoPCS2025/Debug$ ./polyhedra 3 4 3 3 10 18
Modalità: Generazione mesh e calcolo cammino minimo.
Hai inserito: p=3, q=4, b=3, c=3
Cammino minimo da vertice ID: 10 a vertice ID: 18

--- Cammino Minimo ---
Numero di lati nel cammino: 2
Lunghezza totale del cammino: 0.544331
```

Output del cammino minimo



Cammino minimo su paraview

3 4 3 3 da 10 a 18



# Triangolazione

## Solidi Geodetici di classe II

Per la seconda parte del progetto abbiamo costruito i **solidi geodetici di classe II** per  $b = c$ . Per la nuova triangolazione abbiamo scritto il file **Triangulation2.cpp** in cui:

- la funzione **get\_cyclic\_normalized** data una sequenza di id degli spigoli la normalizza per farla iniziare dal valore minimo in modo da poter confrontare due facce;
- la funzione **NormalizeFaceEdges** confronta una sequenza con la sua inversa e restituisce quella lessicograficamente minima;
- la funzione **FindNearBarycenter** trova il baricentro di una faccia adiacente ad un dato spigolo usando la mappa **edgeToFacesMap**;



# Triangolazione

Solidi Geodetici di classe II

La funzione principale per la triangolazione è **triangulateAndStore2**:

- viene inizializzata la mesh triangolata;
- se uno spigolo è di bordo vengono creati 2 sottotriangoli usando il baricentro e il punto medio dello spigolo;
- se lo spigolo è interno si costruiscono 2 triangoli usando il baricentro della faccia corrente, il baricentro della faccia adiacente e i vertici dello spigolo.

Le facce, gli spigoli e i vertici vengono aggiunti solo se non ancora presenti usando le funzioni **FindAddFace**, **FindAddVertice**, **FindAddEdge2**.





```
void FindAddFace(const vector<unsigned
    int>& new_face_vertices,
const vector<unsigned int>&
    new_face_edges, PolyhedralMesh&
    meshTriangulated, unsigned int& k3)
{
vector<unsigned int>
    normalized_new_edges =
    NormalizeFaceEdges(new_face_edges);

bool found = false;
for (unsigned int i = 0; i <
    meshTriangulated.Cell2DsId.size();
    ++i) { vector<unsigned int>
    normalized_existing_edges =
    NormalizeFaceEdges(meshTriangulated
        .Cell2DsEdges[i]);
}
```

```
    if (normalized_new_edges ==
        normalized_existing_edges) {
        found = true;
        break;}
}
if (!found){
meshTriangulated.Cell2DsVertices[k3] =
    new_face_vertices;
meshTriangulated.Cell2DsEdges[k3] =
    new_face_edges;
meshTriangulated.Cell2DsId[k3] = k3;
k3++;
}
}
```



## Google Test

Nel file **test.hpp** abbiamo verificato il corretto funzionamento di ogni unità logica usando i Google Test all'interno del namespace **PolyhedraTest**:

- **TestTriangulationTetrahedron**: verifica la correttezza della funzione di triangolazione del tetraedo;
- **TestOrderedEdges**: controlla il corretto ordinamento degli spigoli nelle facce;
- **DualTest**: verifica la corretta costruzione del poliedro duale;
- **ShortestPath**: verifica la correttezza del cammino minimo su un tetraedo triangolato.



```
\\ TestNotNullArea
for (size_t i = 0; i < meshTriangulated
    .Cell2DsVertices.size(); ++i) {
    const auto& tri = meshTriangulated.
        Cell2DsVertices[i];
    ASSERT_EQ(tri.size(), 3);

    Vector3d A = meshTriangulated.
        Cell0DsCoordinates.col(tri[0]);
    Vector3d B = meshTriangulated.
        Cell0DsCoordinates.col(tri[1]);
    Vector3d C = meshTriangulated.
        Cell0DsCoordinates.col(tri[2]);

    double area = 0.5 * ((B - A).cross(C -
        A)).norm();
    EXPECT_GT(area, eps) << "Triangolo con
        area nulla" << i;
}
```

```
\\ TestNotNullEdges
double eps = numeric_limits<double>::
    epsilon();
for (unsigned int i = 0; i <
    meshTriangulated.Cell1DsExtrema.
        rows(); ++i) {
    int vStart = meshTriangulated.
        Cell1DsExtrema(i, 0);
    int vEnd = meshTriangulated.
        Cell1DsExtrema(i, 1);
    Vector3d startPoint = meshTriangulated.
        Cell0DsCoordinates.col(vStart);
    Vector3d endPoint = meshTriangulated.
        Cell0DsCoordinates.col(vEnd);
    double length = (endPoint - startPoint)
        .norm();

    EXPECT_GT(length, eps)
}
```

# ProgettoPCS2025

---