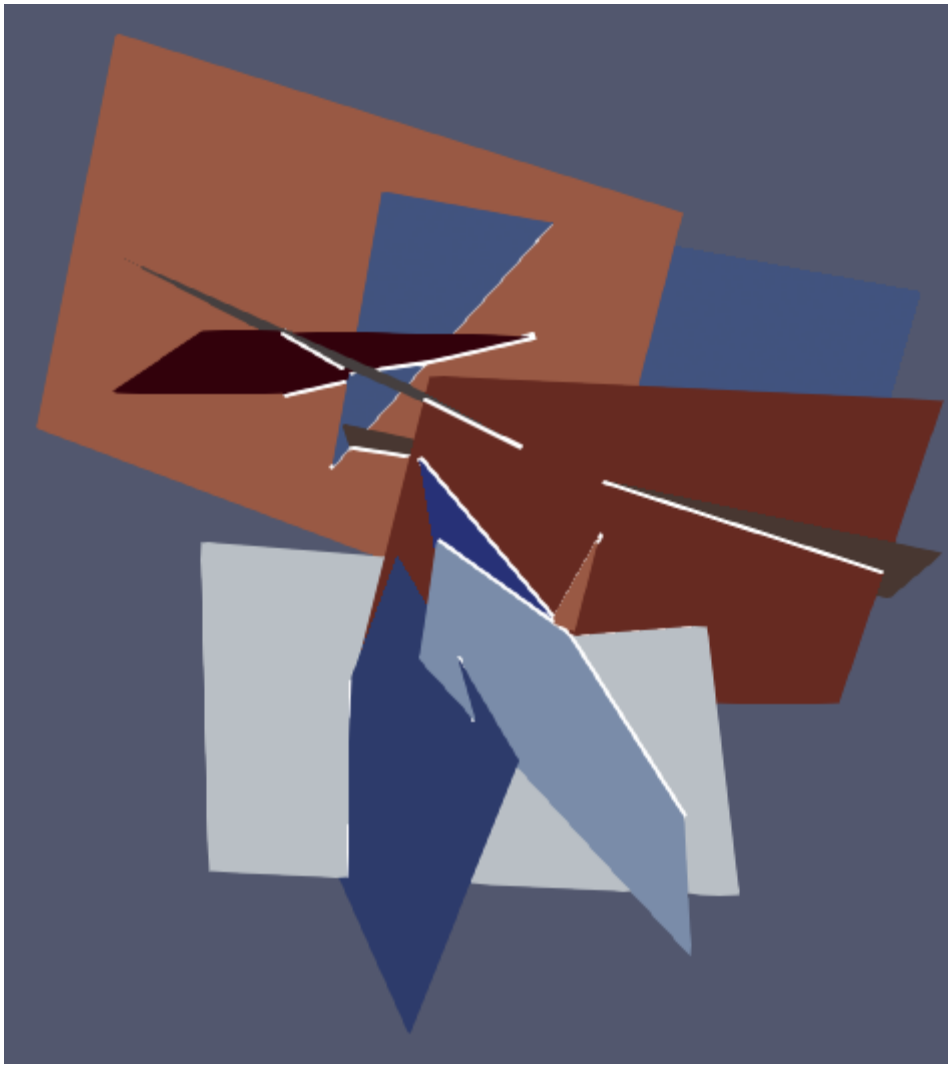

Report Discrete Fracture Network

Santoro Gabriele s283189

INDICE

1 Discrete Fracture Network

Un Discrete Fracture Network (DFN) è un sistema costituito da *N* fratture Fn, n ∈ {1, . . ., N} rappresentate da poligoni planari che si intersecano tra di loro nello spazio tridimensionale.

1.1 Caratteristiche del DFN

- **Fratture**: le fratture in un DFN sono rappresentate da poligoni planari tridimensionali. Ogni frattura è definita da un set di coordinate che identificano i suoi vertici nello spazio tridimensionale.
- **Intersezioni o Tracce**: le intersezioni tra le fratture, chiamate tracce, sono rappresentate da segmenti di linea. Questi segmenti sono definiti dalle coordinate degli estremi delle linee di intersezione.
- **Tracce Passanti e Non-Passanti**:
    - **Traccia Passante**: una traccia è considerata passante per una frattura se i suoi due estremi giacciono sul bordo della frattura;
    - **Traccia Non-Passante**: una traccia è considerata non-passante se almeno uno dei suoi estremi si trova all'interno della frattura.

1.2 Rappresentazione

La rappresentazione grafica di un DFN prevede la visualizzazione delle fratture come superfici piane e delle tracce come linee di intersezione tra queste superfici; questa rappresentazione aiuta a comprendere come le fratture si interconnettono.

2 Struttura C++

Segue la descrizione della struttura in linguaggio C++ utilizzata:

2.1 Strutture dati

2.1.1 Vertex:

```
struct Vertex {
   double x, y, z;
   bool operator<(const Vertex& other) const {
      if (x != other.x) return x < other.x;
      if (y != other.y) return y < other.y;
      return z < other.z;
   }
};
```

Rappresenta un punto nello spazio 3D con coordinate x, y, z.

2.1.2 Fracture:

```
struct Fracture {
   int id;
   vector<Vertex> vertices;
};
```

Rappresenta una frattura identificata da un id e una lista di vertici che definiscono i suoi contorni.

2.1.3 Trace:

```
struct Trace {
   int id;
   int fractureId1;
   int fractureId2;
   bool Tips1;
   bool Tips2;
   vector<Vertex> intersectionPoints;
};
```

Rappresenta un'intersezione (traccia) tra due fratture, con un id, gli ID delle fratture coinvolte, due booleani per indicare se l'intersezione attraversa le fratture e i punti di intersezione.

3 Descrizione delle funzioni

Di seguito è fatta una descrizione dettagliata della struttura delle funzioni che compongono l'algoritmo:

3.1 Funzioni di utilità

3.1.1 segmentsIntersect2D e segmentsIntersect2D_vista_x:

```
bool segmentsIntersect2D(const Vertex& p1, const Vertex& p2, const
Vertex& q1, const Vertex& q2, Vertex& intersection) {

   double s1_x = p2.x - p1.x;

   double s1_y = p2.y - p1.y;

   double s2_x = q2.x - q1.x;

   double s2_y = q2.y - q1.y;


   double s = (-s1_y * (p1.x - q1.x) + s1_x * (p1.y - q1.y)) / (-s2_x *
s1_y + s1_x * s2_y);

   double t = ( s2_x * (p1.y - q1.y) - s2_y * (p1.x - q1.x)) / (-s2_x * s1_y
+ s1_x * s2_y);


   if (s >= 0 && s <= 1 && t >= 0 && t <= 1) {

      intersection.x = p1.x + (t * s1_x);

      intersection.y = p1.y + (t * s1_y);

      intersection.z = p1.z + (t * (p2.z - p1.z));

      return true;

   }


   return false;

}


bool segmentsIntersect2D_vista_x(const Vertex& p1, const Vertex&
p2, const Vertex& q1, const Vertex& q2, Vertex& intersection) {

   double s1_y = p2.y - p1.y;

   double s1_z = p2.z - p1.z;

   double s2_y = q2.y - q1.y;

   double s2_z = q2.z - q1.z;


   double s = (-s1_z * (p1.y - q1.y) + s1_y * (p1.z - q1.z)) / (-s2_y *
s1_z + s1_y * s2_z);

   double t = ( s2_y * (p1.z - q1.z) - s2_z * (p1.y - q1.y)) / (-s2_y * s1_z
+ s1_y * s2_z);
```

```
    if (s >= 0 && s <= 1 && t >= 0 && t <= 1) {
        intersection.x = q1.x + (s * (q2.x - q1.x));
        intersection.y = p1.y + (t * s1_y);
        intersection.z = p1.z + (t * s1_z);
        return true;
    }

    return false;
}
```

Queste funzioni verificano se due segmenti si intersecano rispettivamente nel piano XY e YZ e calcolano il punto di intersezione. Se i segmenti si intersecano, le funzioni calcolano il punto di intersezione e lo salvano in "intersection".

3.1.2 pointInFractures_x e pointInFractures_z:

```
set<Vertex> pointInFractures_x(const set<Vertex>& intersections,
const Fracture& f) {
    set<Vertex> result;
    double x_min = 100, x_max = 0;
    for (size_t n = 0; n < f.vertices.size(); n++) {
        if (f.vertices[n].x > x_max) {
            x_max = f.vertices[n].x;
        }
        if (f.vertices[n].x < x_min) {
            x_min = f.vertices[n].x;
        }
    }
    for (const Vertex& point : intersections) {
        if ((point.x >= x_min && point.x <= x_max)) {
            result.insert(point);
        }
    }
    return result;
}

set<Vertex> pointInFractures_z(const set<Vertex>& intersections,
const Fracture& f) {
    set<Vertex> result;
```

```
    double z_min = 100, z_max = 0;
    for (size_t n = 0; n < f.vertices.size(); n++) {
        if (f.vertices[n].z > z_max) {
            z_max = f.vertices[n].z;
        }
        if (f.vertices[n].z < z_min) {
            z_min = f.vertices[n].z;
        }
    }
    for (const Vertex& point : intersections) {
        if ((point.z >= z_min && point.z <= z_max)) {
            result.insert(point);
        }
    }
    return result;
}
```

Queste funzioni filtrano i punti di intersezione calcolati per determinare se si trovano all'interno delle fratture considerate. La prima funzione lavora nel piano XY, mentre la seconda nel piano YZ. Il risultato è un insieme di punti di intersezione validi per ciascun piano.

3.2 Funzioni principali

3.2.1 calculateIntersectionPoints:

```
vector<Vertex> calculateIntersectionPoints(const Fracture& f1, const Fracture& f2) {
    set<Vertex> uniqueIntersections;
    Vertex intersection;
    for (size_t i = 0; i < f1.vertices.size(); ++i) {
//resto del codice
}
```

Questa funzione è fondamentale per calcolare tutti i punti di intersezione tra due fratture. Utilizza un set per memorizzare i punti di intersezione, assicurandosi di eliminare eventuali duplicati. Verifica le intersezioni nei piani XY e YZ e filtra i punti validi all'interno delle fratture.

3.2.2 printIntersection:

```
bool printIntersection(const Fracture& f1, const Fracture& f2) {
    vector<Vertex> intersectionPoints = calculateIntersectionPoints(f1, f2);
    if (!intersectionPoints.empty()) {
        cout << "Fracture " << f1.id << " intersects with Fracture " << f2.id << endl;
        cout << "Intersection Points:" << endl;
```

```
        for (const Vertex& point : intersectionPoints) {
            cout << "(" << point.x << ", " << point.y << ", " << point.z << ")" << endl;
        }
        return true;
    }
    cout << "Fracture " << f1.id << " does not intersect with Fracture " << f2.id << endl;
    return false;
}
```

Questa funzione è progettata per stampare i punti di intersezione tra due fratture. Utilizza la funzione "calculateIntersectionPoints" per ottenere i punti di intersezione e li stampa. Se non ci sono intersezioni, indica che non ci sono punti di intersezione.

3.2.3 **pointOnSegment**:

```
bool pointOnSegment(const Vertex& point1, const Vertex& point2,
const Vertex& point3) {
    double APx = point1.x - point2.x;
//resto del codice...
```

Questa funzione verifica se un punto specifico si trova su un segmento 3D definito da due estremi. Calcola i vettori tra il punto e gli estremi del segmento e utilizza il prodotto vettoriale per verificare il loro allineamento. Inoltre, controlla se il punto si trova effettivamente all'interno dei limiti del segmento utilizzando il prodotto scalare.

3.2.4 **calculateTips**:

```
bool calculateTips(const Trace& t, const Fracture& f){
    int count = 0;
    for(size_t i=0; i<t.intersectionPoints.size(); i++){
        for(size_t j=0; j<f.vertices.size(); j++){
            Vertex point1 = t.intersectionPoints[i];
            Vertex point2 = f.vertices[j];
            Vertex point3 = f.vertices[(j+1)% f.vertices.size()];
            if(pointOnSegment(point1, point2, point3)){
                count++;
            }
        }
    }
    if(count>=2){
        return true;
    }
    return false;
```

```
}
```

Questa funzione determina se un'intersezione tra una traccia e una frattura ha due o più punti di intersezione validi (ossia "tips"). Utilizza “pointOnSegment” per verificare se i punti di intersezione si trovano sui segmenti della frattura.

3.2.5 **calculateTraces**:

```
vector<Trace> calculateTraces(const vector<Fracture>& fractures) {
   int numFractures = fractures.size();
   vector<Trace> Traces;
   int ids = 0;
   for (int i = 0; i < numFractures - 1; i++) {
//resto del codice
}
```

Questa funzione calcola tutte le tracce di intersezione tra un insieme di fratture. Per ogni coppia di fratture verifica se si intersecano utilizzando “printIntersection” e se verificato, crea una traccia con i dettagli dell'intersezione, compresi i punti di intersezione e la presenza di eventuali "tips". Infine, raccoglie tutte le tracce in un vettore che viene restituito al termine.

3.2.6 **readDFNFile**:

```
vector<Fracture> readDFNFile(const string& filename) {
   ifstream inputFile(filename);
   if (!inputFile.is_open()) {
      cerr << "Error: Unable to open input file." << endl;
      return {};
   }
//resto del codice…
```

Questa funzione legge i dati delle fratture da un file di input ignorando tutte le righe di file che iniziano col carattere “#”. Apre il file specificato e legge il numero di fratture. Per ogni frattura, legge il numero di vertici e le coordinate X, Y, Z, costruendo un oggetto “Fracture” che viene aggiunto a un vettore di fratture. Questa funzione è essenziale per inizializzare i dati necessari per l'analisi delle intersezioni.

3.3 Funzione main

Di seguito il codice senza la parte dei GoogleTest:

```
int main() {
   string filename = "dfn.txt";
   vector<Fracture> fractures = readDFNFile(filename);
   ofstream outputFile("intersections.txt");
   if (!outputFile.is_open()) {
      cerr << "Error: Unable to open output file." << endl;
```

```
        return 1;
    }
    for (size_t i = 0; i < fractures.size(); ++i) {
        for (size_t j = i + 1; j < fractures.size(); ++j) {
            const Fracture& f1 = fractures[i];
            const Fracture& f2 = fractures[j];
            vector<Vertex> intersectionPoints = calculateIntersectionPoints(f1, f2);
            if (!intersectionPoints.empty()) {
                outputFile << "Fracture " << f1.id << " intersects with Fracture " <<
f2.id << endl;
                outputFile << "Intersection Points:" << endl;
                for (const Vertex& point : intersectionPoints) {
                    outputFile << "(" << point.x << ", " << point.y << ", " << point.z <<
")" << endl;
                }
                outputFile << endl;
            }
        }
    }
    outputFile.close();
    return 0;
}
```

La funzione “main” legge i dati dalle fratture dal file di input utilizzando “readDFNFile”, calcola le intersezioni tra tutte le coppie di fratture utilizzando “printIntersection” e scrive i risultati nel file Esegue la lettura del file, calcola le intersezioni tra le fratture e salva i risultati in un file di output.

4 GoogleTest

Si usano i test per verificare il funzionamento delle funzioni.

4.1 Test segmentsIntersect2D

La funzione “segmentsIntersect2D” verifica se due segmenti 2D si intersecano e, se verificato, calcola il punto di intersezione. Nello specifico, nel primo caso i segmenti si intersecano e controllano il punto di intersezione mentre, nel secondo caso i segmenti non si intersecano.

4.2 Test pointOnSegment

La funzione “pointOnSegment” verifica se un punto giace su un segmento definito da altri due punti. Nello specifico abbiamo due test, nel primo viene scelto un punto che è sul segmento, mentre nel secondo viene scelto un punto che non è sul segmento.

4.3 Test calculateIntersectionPoints

La funzione “calculateIntersectionPoints” calcola i punti di intersezione tra due fratture che sono rappresentate come segmenti. Nel primo test viene considerato il caso in cui le fratture si intersecano e controllano il punto di intersezione. Nel secondo test viene considerato il caso in cui le fratture non si intersecano.

5 Conclusioni

In conclusione, questo codice legge un file di input leggendone per ogni frattura i vertici e le loro coordinare X, Y, Z, costruendo in seguito un oggetto “Fracture” il quale viene aggiunto a un vettore di fratture. Successivamente, tramite le funzioni sopracitate (par. 3), crea un file di output dove saranno presenti il numero di tracce presenti tra le fratture determinate, per ogni traccia vengono indicate le due fratture che prevedono in comune la traccia con rispettive coordinate dei punti di intersezione e infine, le tracce vengono suddivise in passanti e non passanti. In quelle passanti viene indicato l’id della frattura per la quale la traccia è passante. Tutto questo viene fatto per 6 file di input e ognuno prevede una quantità diversa di fratture.

5.1 Esempio

Di seguito viene riportato il file di output che viene creato partendo dal file di input “FR3”:

```
# Number of Traces
2
# TraceId; FractureId1; FractureId2; X1; Y1; Z1; X2; Y2; Z2
0; 0; 1; 8.0000000000000004e-01; 0.0000000000000000e+00; 0.0000000000000000e+00; 8.0000000000000004e-01; 1.0000000000000000e+00; 0.0000000000000000e+00
1; 0; 2; 0.0000000000000000e+00; 5.0000000000000000e-01; 0.0000000000000000e+00; 3.1618370000000001e-01; 5.0000000000000000e-01; 0.0000000000000000e+00
#Passanti
#TraceId; FractureId
0; 0
0; 1
#Non Passanti
#TraceId
1
```