Sorbonne Université

C++

Rapport du projet 2

Kevin Ancourt
Louise Ben Salem-Knapp

2019 - 2020

Introduction

Notre objectif est de déterminer rapidement un triangle contenant un point dans un maillage. Pour arriver à ce résultat, notre projet s'est découpé en différentes parties. Tout d'abord, la création d'une classe maillage qui permet de lire un fichier maillage, puis la création de différentes méthodes dans cette classe.

Notre rapport se découpe en deux parties, dans un premier temps l'explication des méthodes de la classe maillage et dans un second temps les tests effectués sur ces méthodes.

Classe maillage

Question 1

Nous avons tout d'abord créé la classe maillage qui se situe dans le fichier maillage.hpp. Cette classe lit un maillage et stocke les différentes informations du maillage. Elle a quatre arguments :

- L'entier nv qui correspond au nombre de sommets dans le maillage.
- L'entier nt qui correspond au nombre de triangles dans le maillage.
- Le vecteur v qui contient les coordonnées des sommets du maillage.
- Le vecteur t qui contient les numéros des trois sommets de chaque triangle du maillage.

Nous avons défini un constructeur de la classe maillage en s'inspirant du constructeur de la classe Mesh2d provenant du fichier EF2d-base.cpp et en l'adaptant à nos choix d'arguments, notamment sur l'utilisation des vecteurs v et t.

```
maillage(const char *filename) {
    ifstream f(filename);
    assert(f); // Teste que le fichier est bien ouvert
    int unused,label;
    double x,y;
    f >> nv >> nt >> unused;
    v.resize(nv);
    t.resize(nt);
    assert(f.good());

    for(int i=0;i<nv;i++){
        f >> x >> y >> label;
        v[i].push_back(x);
        v[i].push_back(y);
        v[i].push_back(label);
        assert(f.good()); }

    for(int k=0;k<nt;++k){
        int v1,v2,v3,useless;
        f >> v1 >> v2 >> v3 >> useless;
        t[k].push_back(v1-1);
        t[k].push_back(v2-1);
        t[k].push_back(v3-1);
        t[k].push_back(useless); }

    cout<< " End Read " << nv << " " << nt << endl; }
```

Constructeur de la classe maillage

Nous avons défini un opérateur d'accès qui permet d'accéder au sommet numéroté localement par i dans le triangle k. Cet opérateur vérifie aussi que les entiers k et i ne prennent pas des valeurs absurdes.

```
int operator()(int k,int i) const {
    assert(k<nt);
    assert(k>=0);
    assert(i>=0);
    assert(i<3);
    return t[k][i]; }
```

Opérateur d'accès de la classe maillage

Nous avons défini un opérateur d'affichage qui affiche les coordonnées des sommets du maillage et les trois sommets de chaque triangle du maillage.

```
inline ostream& operator << (ostream& o, const maillage &Th){

    // Affiche les coordonnées de chaque sommet
    for(int i=0; i<Th.nv ; i++){
        for(int j=0; j<3 ; j++){
            o << Th.v[i][j] << " "; }
        o << endl; }
    o << endl;

    // Affiche les triangles
    for(int i=0; i<Th.nt; i++){
        o << "triangle numéro " << i << " ";
        for(int j=0; j<4; j++){
            o <<  Th.t[i][j] << " "; }
        o << endl; }
    o << endl;
    return o; }
```

Opérateur d'affichage de la classe maillage

Méthodes de la classe

Question 2

Nous avons dans un premier temps créé une méthode qui construit le tableau des triangles adjacents par une arête à chaque triangle du maillage considéré. Pour avoir les triangles adjacents à l'arête opposée au sommet que l'on considère, nous avons défini pour chaque sommet l'arête ne le contenant pas.
On dit que deux arêtes (i,j) et (k,l) appartiennent à la même classe d'équivalence si min(i,j) = min(k,l). La méthode triangle_adj utilise deux tableaux :

- Le tableau d'entiers next dont la taille est le majorant du nombre d'arêtes. Ce tableau permet de parcourir une classe d'équivalence. Le coefficient next[j] est le numéro de l'arête qui précède l'arête j dans sa classe d'équivalence. Si l'arête j n'est précédée d'aucune arête, on définit next[j] = -1.

- Le tableau d'entiers head dont la taille est le nombre de sommets du maillage. Ce tableau permet de récupérer le numéro d'arête le plus grand de sa classe. Le coefficient head[j] est le plus grand numéro d'arête parmi celles de la classe d'équivalence de l'arête associée au sommet j. Si la classe associée à l'arête du sommet j est vide, on définit head[j] = -1, c'est pourquoi le vecteur head est initialisé à -1.

L'avantage de cette méthode est sa faible complexité. En effet, sa complexité est en $O(nt)$. L'algorithme est présenté à la page suivante.

Nous avons ensuite défini la méthode suivante :

```
1 int return_triangle_adj( int * t_adj, int num_t, int i ) {
2     assert(num_t<nt);
3     assert(num_t>=0);
4     assert(i>=0);
5     assert(i<3);
6     return t_adj[ 3 * num_t + i]; }
```

Cette méthode permet d'accéder au triangle adjacent du triangle num_t opposé au sommet i en accédant au tableau t_adj que nous avons créé avec la méthode triangles_adj avec une complexité constante en $O(1)$.

```
int * triangles_adj(const maillage & Th) {
    int * t_adj = new int [Th.nt*3];
    int m = Th.nv;
    int nn = Th.nt*3; // majorant du nombre d'arêtes
    int ne = 0; // nombre d'arêtes
    std::vector<int> head(m,-1),next(nn),v(nn),ta(nn);
    for(int t=0 ;t<Th.nt;t++){
        for(int e=0 ;e<3 ;e++){
        // Initialise à -1 pour traiter le cas où on ne trouve
        // pas de triangle adjacent (triangle du bord)
            t_adj[3*t+e]=-1;

        // On récupère les sommets opposés au sommet considéré
            int i = Th(t, (e+2)%3);
            int j = Th(t, (e+1)%3);
            if( j<i) std::swap(i,j);

            int ke=ne;
            bool exist = false;
            // Permet de ne pas redéfinir une arête déjà traitée
            for ( int p=head[i] ; p>=0 ; p=next[p]){
                // boucle sur les éléments d'une classe
                if( v[p] == j ) {
                    ke = p;
                    exist=true; // si l'arête a déjà été traitée
                                // on ne la traite pas à nouveau
                    break;}
                }

            // Si l'arête n'a pas encore été traitée
            if( !exist ) {
                next[ne]= head[i];
                head[i]= ne;
                v[ne]=j;
                ta[ne] = 3*t+e; // tableau de tous les triangles
                ne++; }
            // Si l'arête a déjà été traitée cela signifie qu'on
            // a trouvé un triangle adjacent
            if( exist ) {
                int tt=ta[ke], ee= tt%3 ;
                tt /= 3;
                t_adj[3*t+e]=tt;
                t_adj[3*tt+ee]=t; }
        }
    }
    return t_adj; }
```

Algorithme de recherche des triangles adjacents

Question 3

Nous avons ensuite implémenté l'algorithme 1 décrit dans l'énoncé du projet. L'objectif de cet algorithme est de déterminer dans quel triangle du maillage se trouve un point choisi p. Cet algorithme part d'un triangle k choisi par l'utilisateur et calcule l'aire signée du triangle formé par (ai,bi,p), pour i = 0,1,2. Il fait ensuite une boucle jusqu'à ce que la taille du tableau triangle_aire_neg soit nulle, c'est-à-dire jusqu'à ce que les trois aires formées par (ai,bi,p) soient positives.

Lorsqu'un des triangles (ai,bi,p) a une aire négative, nous stockons le numéro du triangle adjacent par l'arête qui correspond à cette aire négative. Nous utilisons ce triangle adjacent comme nouveau triangle de référence de la boucle.

Lorsque deux aires sont négatives, nous stockons les numéros des deux triangles adjacents par les arêtes qui correspondent aux aires négatives. Nous choisissons aléatoirement un des deux triangles adjacents stockés au préalable comme nouveau triangle de référence de la boucle.

Dans le cas où le point n'appartient à aucun triangle, l'algorithme renvoie la valeur -1 afin de pouvoir différencier cette valeur des numéros des triangles qui sont tous positif. L'algorithme est présenté à la page suivante.
La complexité de cet algorithme est majorée par $O(\sqrt{nt})$.

```
int Algorithme_1 (int k, const R2 & p){

    int * tadj = (*this).triangles_adj((*this));
    float mes;
    vector<int> triangle_aire_neg(1) ;
    int g = k;

    while (triangle_aire_neg.size() != 0) {
        triangle_aire_neg.resize(0);

        for (int i=0; i<3; ++i){

            R2 A(v[t[g][i]][0],v[t[g][i]][1]), B(v[t[g][(i+1)
    %3]][0],v[t[g][(i+1)%3]][1]);

            mes = det(A,B,p); //Calcul de l'aire du triangle ABp

            if ( mes < 0 ){
                // On stocke quand les aires sont négatives
                triangle_aire_neg.push_back(return_triangle_adj(
    tadj,g,(i+2)%3)); }
        }

        if(triangle_aire_neg.size()==1){
            g=triangle_aire_neg[0];
            if (g==-1) {
                // Cas où le sommet n'appartient à aucun
                // triangle
                break; }
        }

        else if(triangle_aire_neg.size()==2){
            std::bernoulli_distribution R(0.5);
            g=triangle_aire_neg[R(gen)];
            if (g==-1) {
                // Cas où le sommet n'appartient à aucun
                // triangle
                break; }
        }
    }

    delete tadj;
    return g; }
```

Algorithme 1

Question 4

Enfin, nous avons défini l'algorithme Q4 qui détermine pour chaque sommet p d'un maillage Th2, le triangle dans le maillage Th qui contient p. Cet algorithme appelle l'algorithme 1 pour chaque sommet du maillage Th2 en partant du triangle numéro 1. Nous avons fait le choix de partir du triangle 1 car c'est un triangle qui appartient à tous les maillages, même les plus grossiers.

```
1  int * Q4 ( const maillage & Th2) {
2
3      int *tab = new int [Th2.nv];
4
5      for (int i=0; i<Th2.nv; i++){
6          R2 p(Th2.v[i][0],Th2.v[i][1]);
7          tab[i] = (*this).Algorithme_1(1,p);
8          }
9
10     return tab;
11 }
```

La complexité de cet algorithme est en $O(\text{Th2.nv}*\sqrt{Th.nt})$.

Procédure de validation

Nous allons tester notre classe maillage et toutes ses méthodes sur un premier cas simple, celui du cercle de rayon 0.9 avec 10 points sur le cercle.

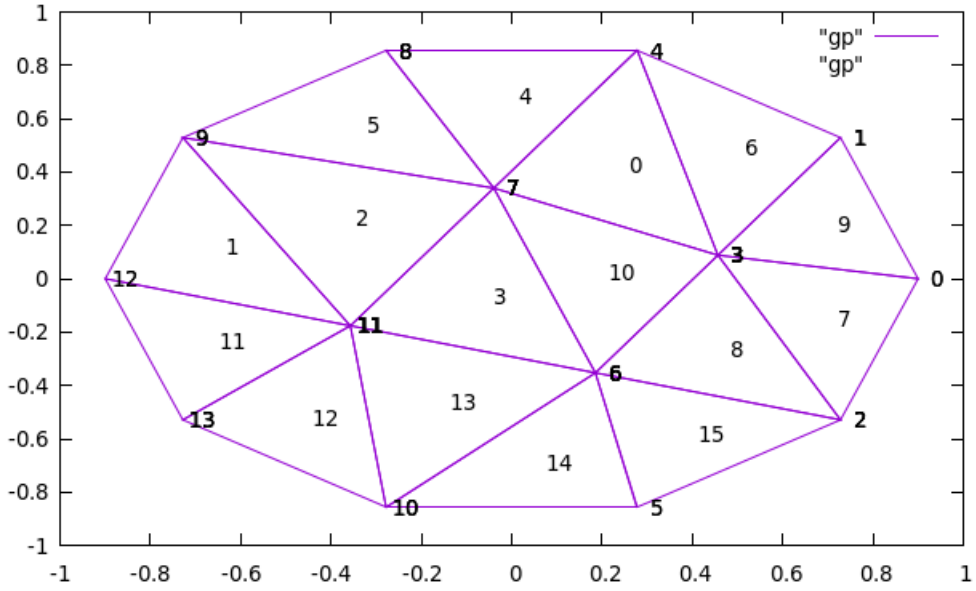

FIGURE 1 – Maillage du cercle de centre (0,0) et de rayon 0.9

Nous avons créé un fichier main.cpp qui permet de tester la classe et ses méthodes. Chaque méthode est testée séparément dans une fonction qui est appelée dans le main. La première méthode testée est le constructeur de la classe et l'affichage du maillage. On obtient le résultat suivant dans le terminal.

```
sommet numéro 0 de coordonnées : 0.9 0
sommet numéro 1 de coordonnées : 0.728115 0.529007
sommet numéro 2 de coordonnées : 0.728115 -0.529007
sommet numéro 3 de coordonnées : 0.456763 0.0881678
sommet numéro 4 de coordonnées : 0.278115 0.855951
sommet numéro 5 de coordonnées : 0.278115 -0.855951
sommet numéro 6 de coordonnées : 0.18541 -0.352671
sommet numéro 7 de coordonnées : -0.0395898 0.339808
sommet numéro 8 de coordonnées : -0.278115 0.855951
sommet numéro 9 de coordonnées : -0.728115 0.529007
sommet numéro 10 de coordonnées : -0.278115 -0.855951
sommet numéro 11 de coordonnées : -0.357295 -0.176336
sommet numéro 12 de coordonnées : -0.9 1.10218e-16
sommet numéro 13 de coordonnées : -0.728115 -0.529007
```

```
triangle numéro 0 de sommets : 7 3 4
triangle numéro 1 de sommets : 11 9 12
triangle numéro 2 de sommets : 11 7 9
triangle numéro 3 de sommets : 7 11 6
triangle numéro 4 de sommets : 8 7 4
triangle numéro 5 de sommets : 8 9 7
triangle numéro 6 de sommets : 1 4 3
triangle numéro 7 de sommets : 3 2 0
triangle numéro 8 de sommets : 3 6 2
triangle numéro 9 de sommets : 1 3 0
triangle numéro 10 de sommets : 7 6 3
triangle numéro 11 de sommets : 13 11 12
triangle numéro 12 de sommets : 13 10 11
triangle numéro 13 de sommets : 11 10 6
triangle numéro 14 de sommets : 6 10 5
triangle numéro 15 de sommets : 2 6 5
```

Le fichier de maillage a bien été lu et les informations pertinentes ont bien été extraites et affectées dans les variables correspondantes de la classe.

Pour tester l'algorithme de la question 2 qui consiste à trouver le triangle adjacent à un triangle donné, nous allons d'abord afficher l'ensemble des triangles adjacents à chaque triangle d'un maillage donné puis nous allons appeler notre algorithme renvoyant un triangle adjacent au triangle donné opposé au sommet choisi.

```
Le triangle 0 a pour triangles adjacents : 6 4 10
Le triangle 1 a pour triangles adjacents : -1 11 2
Le triangle 2 a pour triangles adjacents : 5 1 3
Le triangle 3 a pour triangles adjacents : 13 10 2
Le triangle 4 a pour triangles adjacents : 0 -1 5
Le triangle 5 a pour triangles adjacents : 2 4 -1
Le triangle 6 a pour triangles adjacents : 0 9 -1
Le triangle 7 a pour triangles adjacents : -1 9 8
Le triangle 8 a pour triangles adjacents : 15 7 10
Le triangle 9 a pour triangles adjacents : 7 -1 6
Le triangle 10 a pour triangles adjacents : 8 0 3
Le triangle 11 a pour triangles adjacents : 1 -1 12
Le triangle 12 a pour triangles adjacents : 13 11 -1
Le triangle 13 a pour triangles adjacents : 14 3 12
Le triangle 14 a pour triangles adjacents : -1 15 13
Le triangle 15 a pour triangles adjacents : 14 -1 8
```

Lorsqu'un -1 est affiché cela signifie qu'il n'y a pas de triangle adjacent opposé au sommet considéré. Dans un cercle, par exemple, un triangle du bord du maillage ne possède que deux triangles adjacents.

Nous pouvons maintenant tester notre algorithme qui permet d'accéder au triangle adjacent d'un triangle donné opposé à un sommet donné numéroté localement entre 0 et 2. Pour faire ce test, nous allons considérer le triangle 1 et son sommet numéro 2. Ce sommet est le sommet numéro 12 si on se réfère à la lecture du maillage page 10. Nous devons obtenir le triangle numéro 2. Lorsque nous lançons l'algorithme test correspondant, nous obtenons bien :

```
1 Le triangle adjacent au triangle 1 opposé
2 au sommet numéro 2 est : 2
```

Nous avons testé ensuite l'algorithme 1 en partant du triangle 0 avec le point de $\mathbb{R}^2$ p(-0.6,-0.3) qui appartient au triangle 11. Nous obtenons :

```
1 Le point p (-0.6 -0.3) appartient au triangle : 11
```

Enfin, nous avons testé notre algorithme Q4 avec différents maillages. Tout d'abord avec le maillage que nous avons utilisé jusqu'ici pour tester nos méthodes et avec un autre maillage simple, puis avec des maillages plus complexes pour tester les performances de notre algorithme.
Nous utilisons dans un premier temps les deux maillages ci-dessous. Celui de gauche est le maillage Th2 du cercle de rayon 0.9 avec 10 points sur le bord utilisé jusqu'à présent et celui de droite est le maillage Th du même cercle qui contient 20 points sur le bord.

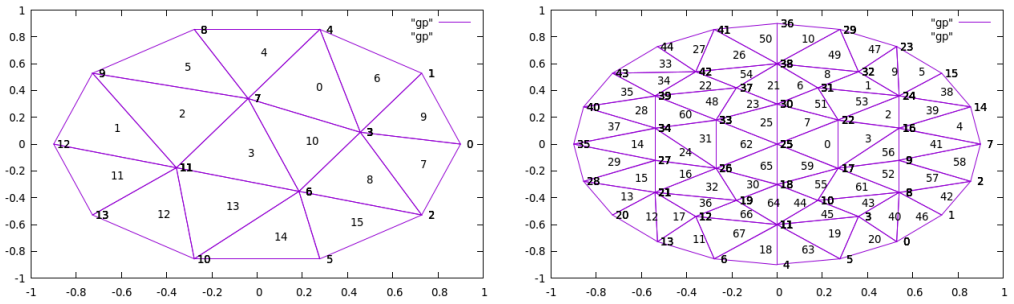

FIGURE 2 – Maillages du cercle de centre (0,0) et de rayon 0.9

Nous cherchons à déterminer dans quel triangle de Th2 se trouvent les sommets de Th. Notre algorithme Q4 nous renvoie le résultat suivant qui correspond bien à ce que nous pouvons observer sur les maillages :

```
Le sommet 0 est inclus dans le triangle 4
Le sommet 1 est inclus dans le triangle 5
Le sommet 2 est inclus dans le triangle 42
Le sommet 3 est inclus dans le triangle 3
Le sommet 4 est inclus dans le triangle 49
Le sommet 5 est inclus dans le triangle 20
Le sommet 6 est inclus dans le triangle 55
Le sommet 7 est inclus dans le triangle 21
Le sommet 8 est inclus dans le triangle 26
Le sommet 9 est inclus dans le triangle 34
Le sommet 10 est inclus dans le triangle 67
Le sommet 11 est inclus dans le triangle 16
Le sommet 12 est inclus dans le triangle 37
Le sommet 13 est inclus dans le triangle 12
```

Temps de calcul

Après avoir testé si nos méthodes fonctionnaient bien, nous nous sommes intéressés au temps de calcul pour la dernière méthode car cette méthode utilise toutes les autres méthodes que nous avons créées auparavant et c'est la plus coûteuse en temps de calcul et en complexité. Pour appliquer l'algorithme Q4 nous avons besoin de deux maillages : le maillage Th2 sera le maillage le plus grossier et le maillage Th sera le plus fin. En faisant ce choix nous considérons ainsi un nombre de sommets plus faible mais un nombre de triangles plus élevé. Nous avons mesuré le temps d'exécution pour différents maillages du cercle de centre (0,0) et de rayon 0.9 allant de maillages grossiers à des maillages beaucoup plus fins :

- Si nous considérons tout d'abord le maillage Th ayant 20 points sur le bord et le maillage Th 2 qui a 10 points sur le bord, nous obtenons un temps de 0.000572 s.
- Pour Th ayant 100 points sur le bord et Th2 ayant 50 points sur le bord, nous obtenons un temps de 0.139 s.
- Pour Th ayant 150 points sur le bord et Th2 ayant 100 points sur le bord, nous obtenons un temps de 1.32781 s.
- Pour Th ayant 300 points sur le bord et Th2 ayant 150 points sur le bord, nous obtenons un temps de 11.8885 s.
- Pour Th ayant 500 points sur le bord et Th2 ayant 300 points sur le bord, nous obtenons un temps de 133.358 s.
- Pour Th ayant 1000 points sur le bord et Th2 ayant 750 points sur le bord, nous obtenons un temps de 3070.07 s.

Nous avons donc observé que le temps est d'autant plus important que les maillages sont fins, ce qui est cohérent au vu de la complexité de notre algorithme qui dépend du nombre de triangles de Th et du nombre de sommets de Th2.

Si on inverse les deux maillages, on obtient des temps de calcul du même ordre. Et de la même manière, le temps de calcul augmente avec la finesse du maillage.

Conclusion

Nous avons pu mettre en place un algorithme de recherche rapide de triangle contenant un point dans un maillage en utilisant plusieurs méthodes de la classe maillage que nous avons créée. Notre algorithme a une complexité en $O(\text{Th2.nv}*\sqrt{Th.nt})$ et les temps de calcul sont proportionnels à la finesse des maillages utilisés.

Nous tenons à remercier Monsieur Hecht pour les cours de C++.