Polytech Paris UPMC

Maillages 2D et Surfaciques

Auteurs :
Tupac Quiroga
Alexandre Poulain
Adrian Ahne

Professeur :
Chiara Nardoni

Sommaire

1 Introduction

Notre intérêt pour la implémentation et l'utilisation des maillages en simulation numérique nous a conduit à choisir ce sujet pour notre projet. Un maillage est une représentation ou modélisation d'un domaine à l'aide d'élements géométriques finis. Ici nous travaillerons avec des maillages en dimension 2 et 3. Ils seront ainsi formés de tétraèdres pour les maillages en 3 dimensions et de triangles pour les maillages en 2 dimensions. Nos maillages seront donc constituées de points(vertices), de triangles(triangles), d'arêtes(edges) et de crêtes(ridges). Le format des maillages est le ".mesh" : ceci correspond à une structure précise pour le fichier contenant toutes les données du maillage. L'organisation du fichier .mesh est la suivante :

- Le première ligne contient l'information relative à la version du maillage utilisée : "MeshVersionFormatted"
- La deuxième ligne correspond à la dimension du maillage "Dimension"
- Par la suite le reste des informations relatives aux données géométriques est défini selon la structure suivante :

```
type des elements
nombre de ces elements

...
liste de tout les elements
...
...
```

Dans la liste des élements, on définit ces derniers par leurs coordonnées si ce sont les points ou sinon par le numéro des points si ce sont des triangles ou des arêtes. Pour chacun de ces élements le dernier champs correspond à la réference qui sert par exemple à décrire la couleur souhaitée pour cet élement. A ces fichiers de données ".mesh" nous pouvons y associer un autre fichier contenant la solution d'un calcul. Ce fichier est "même nom que le maillage.sol". Ce fichier précise dans son en-tête sur quel élement s'applique cette solution. Par exemple si cette solution s'applique aux points d'un maillage en 3 dimensions l'en-tête ressemblera à ceci :

```
MeshVersionFormatted 2
Dimension 3
SolAtVertices
```

Le logiciel utilisé pour la visualisation des maillages est medit. Ce logiciel libre à été implémenté par Pascal Frey , directeur de l'institut du calcul et de la modélisation à l'UPMC, et avec l'aide de ces étudiants et colaborateurs. C'est avec l'aide de ces premières précisions sur les structures de données utilisées et sur le logiciel de visualisation que nous pouvons commencer à décrire l'objectif même de ce travail de groupe.

Le but de ce projet étant de calculer la distance de Hausdorff permettant de mesurer l'écart séparant deux maillages. Ce rapport présentera deux points de vue, en effet nous décrirons les méthodes utilisées de manière mathématique et nous expliquerons en détails nos choix pour le codage des différentes fonctions mettant en application ces méthodes. Cherchant ainsi à atteindre cet objectif nous commencerons tous d'abord par l'implémentation et l'explication de plusieurs fonctions "outils" qui nous servirons par la suite. Ce projet continuera par l'explication des différentes méthodes permettant de définir le voisinage d'un point. L'objectif de cette partie est donc de diminuer au maximum le temps de calcul du point le plus proche. Nous terminerons ce rapport par la dernière phase du projet : l'implémentation de la distance de Hausdorff.

2 Structures mesh

Avant de définir la structure principal pour décrire le mesh, on introduit des structure de base.

2.1 Point

La structure *Point* décrit un point dans le mesh donné par ses trois coordonnés et une reference laquelle montre sa position dans le fichier .mesh

```
typedef struct {
  double      c[3],ref;
} Point;
typedef Point * pPoint;
```

2.2 Edge

Pour être capable de définir une arête *Edge* entre deux points on utilise la structure ci-dessous. La reference de cette *Edge* dans le fichier .mesh va enregistrer aussi. QU'EST-CE QUE 'tag' dit ?

```
typedef struct {
  int         v[2],ref,tag;
} Edge;
typedef Edge * pEdge;
```

2.3 Tria

Dans le cours supplémentaire, des nouveaux fonctions et opérations sont calculé avec de l'aide de la structure *Tria* qui décrit une triangle avec ses trois points et encore la reference dans le fichier .mesh

```
typedef struct {
  int         v[3],ref;
} Tria;
typedef Tria * pTria;
```

2.4 Mesh

Après avoir introduit les structure de l'aide, on peut se consacrer à étudier la structure principal *mesh* dans laquelle tous les données de la fichier .mesh sont enregistré. *np* donne le nombre de points dans le mesh, *na* le nombre des arêtes, *nt* le nombre des triangles , ...*ver* est la version du mesh et *dim* est la dimension. *namein* est le nom de fichier input, *nameout* est le nom de fichier output. *point*, *edge*, *tria* sont les données des points, arêtes et triangles du fichiers. Les autres variables dans la structures vont introduire plus tard et expliqué.

```
typedef struct {
  int        np,na,nt,nr,nn,ver,dim,mark;
  char      *namein,*nameout;
  double    *sol,o[3],rad;
  pPoint     point;
  pNormal    Normal;
  pEdge      edge;
  pTria      tria;
  pTriaNorm  triaNorm;
  int *adja;
} Mesh;
typedef Mesh * pMesh;
```

3 Fonctions basics

3.1 Translation

Les fonctions de translation permettent la translation en 2 ou 3 dimension du maillage. Cette opération est facile à programmer. Considerons le cas en dimension 2 (équivalent en dimension 3) :
Le but est de déplacer le maillage de x_{trans} dans la direction de x et de y_{trans} dans la direction de y. Par conséquent, pour chaque point du maillage, la coordonnée doit être déplacer par x_{trans} et son y-coordonnée par y_{trans}. On obtient alors pour les nouveaux points :

$$x = x + x_{trans}$$
$$y = y + y_{trans}$$

pour tous les points dans le maillage.

3.2 Rotation

Les fonctions de rotation permettent la rotation en 2 ou 3 dimensions d'un maillage. Ces rotations sont données en radian. Le principe est simple puisque pour chaque point du maillage on calcule ses nouvelles coordonnées.

Ainsi pour la rotation 2D d'un angle α, les nouvelles coordonnées des points notés x_r et y_r sont données par :

$$x_r = xcos(\alpha) - ysin(\alpha)$$
$$y_r = ycos(\alpha) + xsin(\alpha)$$

Pour la rotation en 3 dimensions, on considère 3 angles de rotation θ_x, θ_y et θ_z. De même les nouvelles coordonnées des points sont notées x_r, y_r et z_r. Néanmoins pour la rotation 3D, on doit distinguer les 3 rotations possibles :

- Rotation selon l'axe x d'un angle θ_x, on calcule alors les nouvelles coordonnées
  $y_r = ycos(\theta_x) - zsin(\theta_x)$
  $z_r = ysin(\theta_x) + zcos(\theta_x)$
- Rotation selon l'axe y d'un angle θ_y
  $x_r = zsin(\theta_y) + xcos(\theta_y)$
  $z_r = zcos(\theta_y) - xsin(\theta_y)$
- Rotation selon l'axe z d'un angle θ_z
  $x_r = xcos(\theta_z) - ysin(\theta_z)$
  $y_r = xsin(\theta_z) + ycos(\theta_z)$

En fonction des 3 angles possibles de rotation, on applique les nouvelles coordonnées de 2 points complémentaires. Notez qu'on peut donner plusieurs angles de rotations à la fonction, c'est à dire que si on souhaite une rotation de θ_x, θ_y et θ_z, la fonction va faire en tout 6 calculs, 2 pour chaque angle.

3.3 Normale

La fonction de calcul des normales de chacun des triangles qui composent le maillage se différencie des autres fonctions basiques par le fait qu'elle ne modifie pas le maillage mais ajoute une information à celui-ci. Ainsi nous

avons créé un structure *pTriaNorm* qui contient un point de type *double n[3]* et un poids normalisé qui est proportionnel à la surface du triangle. La normale d'un triangle est donc un vecteur composé d'un point du triangle considéré et d'un autre point calculé par la fonction. Dans la nomenclature que nous avons imaginé, la normale d'indice i est associée au triangle d'indice i comme il suit.

```
typedef struct {
  int         v[3],ref;
} Tria;
typedef Tria * pTria;
```

Considérons maintenant les 3 points du triangle pour lequel nous souhaitons calculer la normale : P1, P2 et P3. Notons les vecteurs $\vec{V} = P1P2$ et $\vec{W} = P1P3$, la normale au triangle est donné par le produit vectoriel $\vec{n} = \vec{V} \wedge \vec{W}$. Nous devons maintenant considérer les 3 points du triangles par ses coordonnées pour pouvoir implémenter cette fonction. On peut donc calculer le vecteur normal n :

$$\begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} \wedge \begin{pmatrix} W_x \\ W_y \\ W_z \end{pmatrix} = \begin{pmatrix} P2_x - P1_x \\ P2_y - P1_y \\ P2_z - P1_z \end{pmatrix} \wedge \begin{pmatrix} P3_x - P1_x \\ P3_y - P1_y \\ P3_z - P1_z \end{pmatrix}$$

$$\begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \begin{pmatrix} (P2_y - P1_y)(P3_z - P1_z) - (P2_z - P1_z)(P3_y - P1_y) \\ (P2_z - P1_z)(P3_x - P1_x) - (P2_x - P1_x)(P3_z - P1_z) \\ (P2_x - P1_x)(P3_y - P1_y) - (P2_y - P1_y)(P3_x - P1_x) \end{pmatrix}$$

En appliquant ainsi ce calcul sur l'ensemble des triangles du maillage, on peut facilement visualiser le champs de normales comme sur la figure 1 à la surface du maillage grace à medit en appuyant sur la touche N.

FIGURE 1 – Normales (en rose) d'un maillage 3D

3.4 Superposition

3.5 Courbure

La fonction courbure est la fonction qui calcule les courbures au niveau des points et génère un fichier .sol associé au mesh contenant la solution de ce calcul aux points. Le calcul de la courbure au niveau des points pour un maillage en deux dimension correspond à l'angle entre deux arêtes. La fonction parcours donc l'ensemble des points et teste si un point est à l'intersection de deux arêtes, si oui on calcule la soustraction entre 180 degrés et l'angle formé par les deux arêtes. On obtient ainsi la courbure des arêtes d'un maillage en deux dimension. Sur la figure suivante on peut observer le résultat de la fonction courbure2D.

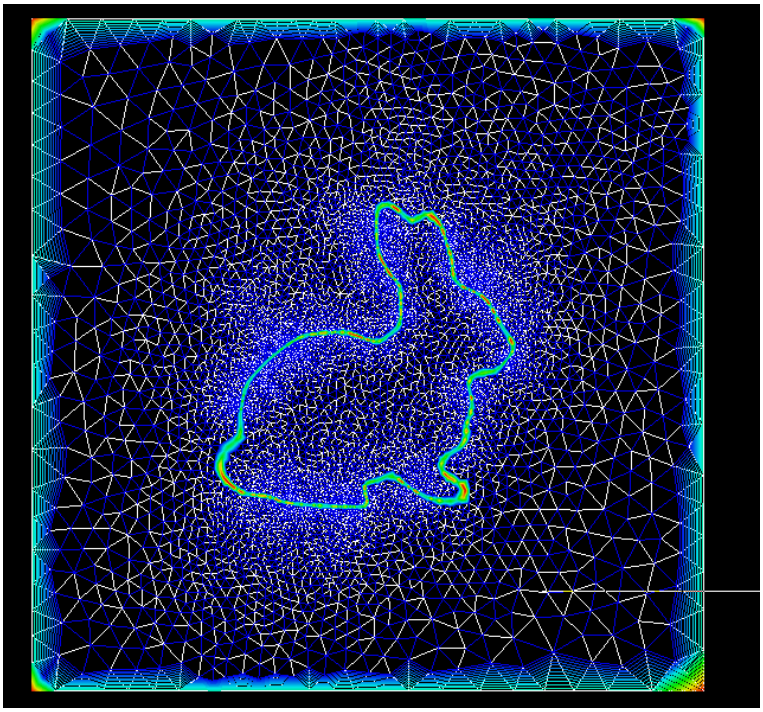

FIGURE 2 – Résultat de la fonction courbure 2D

Nous observons la courbure des arêtes sur cette image, en effet plus la courbure est forte plus le point apparaît en rouge et si la courbure est faible le point apparaît en vert. Pour un maillage en trois dimension, le calcul de la courbure s'effectue de la manière suivante. Si on prend un point du maillage, il appartient aux triangles qui l'entoure. On effectue la somme des angles des triangles auquel le point appartient puis on retranche à 180° (angle plat) ce résultat : on obtient ainsi une valeur correspondant à la courbure. En effectuant ce calcul à l'ensemble des points on obtient pour un tore maillé finement la figure suivante :

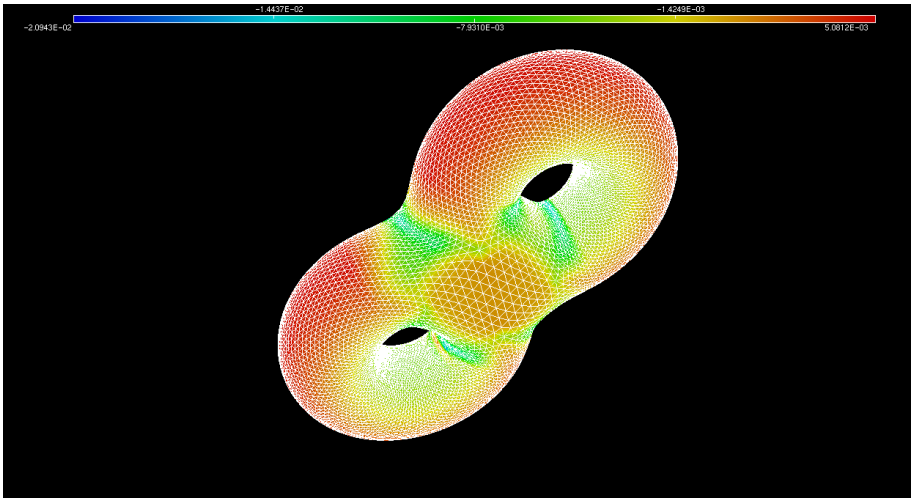

FIGURE 3 – Résultat de la fonction courbure 3D

On observe en rouge les points ou la courbure est forte et en vert les points ou la courbure est forte mais dans l'autre sens de courbure. Les calculs des courbures restent assez simple mathématiquement mais restent assez lourd car le parcours de tous les points reste nécessaire.

4 bucket

L'objectif principal de cette partie est d'expliquer comment nous arrivons à subdiviser un maillage spatialement. La nécessité de partitionner le maillage s'impose lorsque par exemple un calcul nécessite un voisinage. Au lieu de parcourir l'ensemble des points, parcourir uniquement les zones intéressantes nous permet de diminuer grandement le temps et la loureur de certains calculs. La technique décrite ici est celle du bucket : il s'agit de subdiviser le maillage en plusieurs sous-domaines de même taille. La boite dans laquelle est contenu le maillage est donc sous-divisée en plusieurs petites boites : chacune d'entre elle est associée à une clé. Lorsqu'un point "tombe" dans une des petites boites on lui associe la clé correspondant à la boite. Ainsi on peut pour certains calculs au lieu de regarder l'ensemble des points, on peut tester juste les points contenus dans les boites aux alentours. Le bucket permet donc de subdiviser spatialement le maillage vis-à-vis des points. Nous définirons dans notre programme la structure du bucket comme suit :

```
/* Structure bucket */
```

```
typedef struct {
  int size ;
  int *head ;
  int *link ;
} Bucket;
```

Dans cette définition, "size" correspond au nombre de sous-divisions souhaité."head" et "link" sont deux tableaux d'indice de points que nous décrirons par la suite. Cependant, précisons que la taille de head est de la taille du nombre de sous-divisions c'est-à-dire size^3 multiplié par la taille d'un entier (int) et la taille de "link" est égal aux nombres de points multiplié par la taille d'un nombre entier (int). Pour chaque point du maillage nous allons calculer les trois entiers i,j et k en fonction des coordonnées du point. **N, c'est quoi ? Le nombre de points ?**

```
i =  max (0,(int)(N * p->c[0]) -1);
k = max (0,(int)(N * p->c[1]) -1);
j =  max (0,(int)(N * p->c[2]) -1);
```

à l'aide de ces entiers compris entre 1 et size-1, nous allons pouvoir calculer la key de manière unique.

```
key = (k* size + j)*size + i ;
```

Une fois cette clé calculé nous allons placer l'indice du point dans le tableau "head" à l'indice "key", soit :

```
 bucket->head[key] = point ;
```

Si cette case du tableau est vide, alors il s'agit du premier point associé à cette boite du partitionnement. En revanche, si cette case n'est pas vide, on place l'élément présent à cette case du tableau dans le tableau "link" de la manière suivante.

```
 bucket->link[Point] = bucket->head[key] ;
```

Puis on met à jour le tableau "head" avec le nouveau point. A la fin de ce processus sur tout les points, on obtient donc le partitionnement voulu. A présent, on peut utiliser ce bucket pour trouver les points d'un maillage A les plus proche d'un point n'appartenant pas à ce maillage. Pour cela il suffit de calculer la clé comme précédemment en utilisant les coordonnées du point, puis tester si cette clé dans le tableau "head" et récuperer ainsi les points

les plus proches. Si aucun point ne se trouve dans le tableau "head" avec cette clé il suffit de faire varier légerement les indices i,j et k pour tester les alentours et obtenir les points les plus proches.

5 hash

5.1 coordonnées barycentriques

Pour décrire une relation entre un point et un triangle on introduit les coordonnées barycentriques, qui nous donnent une possibilité efficient de le faire. Donné un triangle T avec ses trois points A, B, C non alignés.

Théorème. Pour tout point $P \in \mathbb{R}^3$, il existe un et un seul triple de nombres $(\lambda_1, \lambda_2, \lambda_3)$ vérifiant

$$\begin{cases} \lambda_1 + \lambda_2 + \lambda_3 = 1 \\ P = \lambda_1 A + \lambda_2 B + \lambda_3 C \end{cases}$$

Les nombres $(\lambda_1, \lambda_2, \lambda_3)$ vérifiant (1) sont les *coordonnées barycentriques* de P par rapport au triangle ABC, compare figure 4.

Il est possible aussi de déterminer la position d'un point quelconque au triangle. En plus on calcule les coordonnées barycentriques $(\lambda_1, \lambda_2, \lambda_3)$ d'un point P et regarde leurs signes. Voir figure 5, si par exemple les coordonnées barycentriques ont des valeurs suivantes : $\lambda_1 > 0$, $\lambda_2 < 0$, $\lambda_1 > 0$, donc on sait que le point P se trouve dans le domaine à l'extérieur du triangle à l'arête $\overrightarrow{AC}$.

Comment on peut déterminer les coordonnées barycentriques
Les trois points A, B, C définissent un plan dans l'espace. Choissi un des points et regarde tous les autres location sur le plan comme relative à ce point. *(Pick on of the points and consider all other locations on the plan as relative to that point).* Donc on choissit A et les deux arêtes qui touchent A, alors $(C - A)$ et $(B - A)$. Maintenant on est capable d'exprimer tous les points P dans l'espace, juste par commencer en A et aller quelque distance vers la direction $(C - A)$ et après vers la direction $(B - A)$. Alors on peut

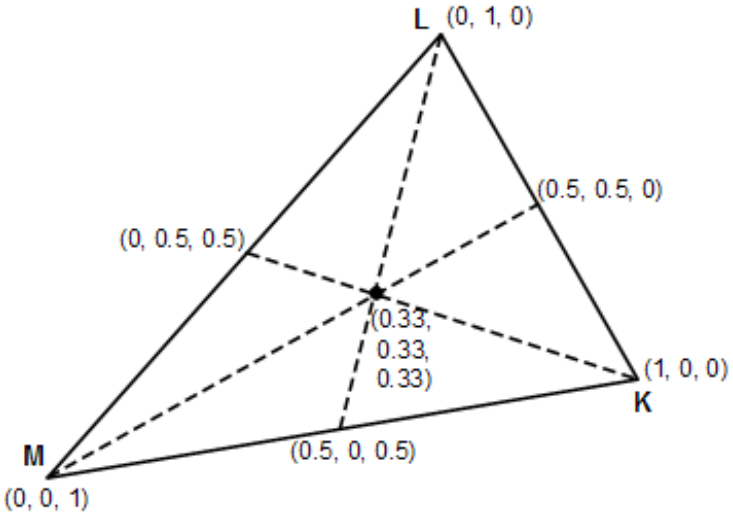

FIGURE 4 – Exemple : Coordonnées barycentriques pour un triangle avec des points KML

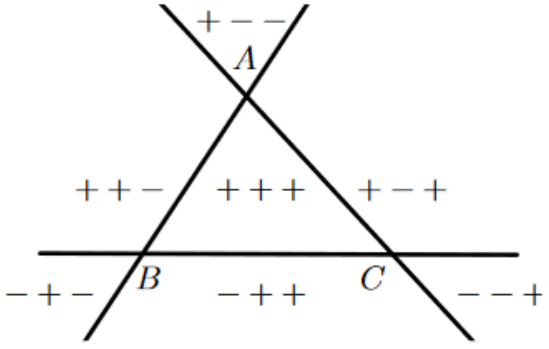

FIGURE 5 – Exemple : Domaine dans lequel se trouve un point, réferant au triangle, dépendant aux signes de coordonnées barycentriques en dimension 3.

décrire tous les points P par

$$P = A + u * (C - A) + v * (B - A) \tag{1}$$

Remarque que si u ou $v < 0$, on est allé dans la fause direction et on doit être à l'extérieur du triangle. La même, si u ou $v > 1$, on est allé trop loin dans une direction et on est à l'extérieur du triangle aussi. Et par consequence, si $u + v > 1$, alors on a dépassé *(crossed ?)* l'arête BC en sortant *(leaving)* le triangle encore.

Pour calculer les coordonnées barycentriques u, v et $(1 - u - v)$ on reprend la formule 1, on soustrait A des deux côtes

$$(P - A) = u * (C - A) + v * (B - A) \tag{2}$$

et on substitute, alors on obtient la formule

$$v2 = u * v0 + v * v1 \tag{3}$$

avec $v0 = (C - A)$, $v1 = (B - A)$, $v2 = (P - A)$.
Dans cette equations il y a deux variables inconnu u et v, donc on a besoin de deux equations pour les résoudre. C'est pourquoi on utilise le produit scalaire $\langle ., . \rangle$ une fois avec $v0$ et une fois avec $v1$ et donc

$$\begin{aligned} \langle v2, v0 \rangle &= \langle u * v0 + v * v1, v0 \rangle \\ \langle v2, v1 \rangle &= \langle u * v0 + v * v1, v1 \rangle \end{aligned} \tag{4}$$

et donc

$$\begin{aligned} \langle v2, v0 \rangle &= u * \langle v0, v0 \rangle + v * \langle v1, v0 \rangle \\ \langle v2, v1 \rangle &= u * \langle v0, v1 \rangle + v * \langle v1, v1 \rangle \end{aligned} \tag{5}$$

Et maintenant on a obtenu deux équations avec deux variables inconnu qui on peut résoudre facilement. Finalement on obtient :

$$\begin{aligned} u &= \frac{\langle v1, v1 \rangle * \langle v2, v0 \rangle - \langle v1, v0 \rangle * \langle v2, v1 \rangle}{\langle v0, v0 \rangle * \langle v1, v1 \rangle - \langle v0, v1 \rangle * \langle v1, v0 \rangle} \\ v &= \frac{\langle v0, v0 \rangle * \langle v2, v1 \rangle - \langle v0, v1 \rangle * \langle v2, v0 \rangle}{\langle v0, v0 \rangle * \langle v1, v1 \rangle - \langle v0, v1 \rangle * \langle v1, v0 \rangle} \end{aligned} \tag{6}$$

Le troisième coordonné barycentrique se calcule par $1 - u - v$.
SOURCES ! ! !

5.2 Hash des arêtes du maillage

Lorsqu'on travaille sur un maillage, on a souvent besoin de savoir la relation qu'on les triangles voisins entre eux. En effet si l'oeil humain parvient à savoir immédiatement quels sont les voisins d'un triangle, la machine doit elle parcourir l'ensemble du maillage pour avoir cette information. La table de hash stocke, pour chacun des triangles du maillage, ses 3 voisins (dans le cas d'un maillage conforme) ainsi que la manière dont chacun de ses triangles adjacent "voit" le triangle. Considérons le cas de la figure 6 ci-dessous.

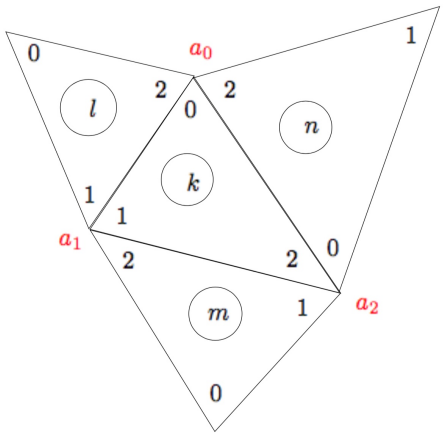

FIGURE 6 – Illustration du hash avec 4 triangles

Ici on a 4 triangles k, l, m et n. Les entiers 0, 1 et 2 correspondent à la numérotation locale des sommets du triangle et a_0, a_1 et a_2 sont les sommets du triangles k en numérotation globale. L'objectif concret est donc de pouvoir accéder au voisins du triangles k de sorte que (3k+0) ait pour voisin (3m+0), (3k+1) ait pour voisin (3n+1) et (3k+2) ait pour voisin (3l+2). Remarquez qu'on a utilisé comme convention qu'on attribue la numérotation des arrêtes en prenant le sommet opposé.

La structure du hachage que nous avons utilisé et qui permet donc de stocker astucieusement les arrêtes (na,nb) et leurs relations est la suivante :

```
typedef struct {
        int ia, ib, adj1, adj2, nxt;
```

```
} Hedge;
typedef Hedge * pHedge;
```

On associe donc un objet de cette structure à chacune des arêtes formées des points na et nb. Les entiers ia et ib permettent de stocker les 2 points d'une arête avec la convention $ia = min(na, nb)$ et $ib = max(na, nb)$. Les entiers adj1 et adj2 permettent de recueillir les 2 relations d'adjacences de chaque arrête. Enfin nxt va permettre de chaîner les objets Hedge et contiennent donc un indice. En effet, les objets Hedge sont rangés dans un tableau global de type

```
Hedge *tab;
```

A chaque arête (na,nb) est associée une clé de hachage qui se calcule simplement à partir des extrémités na et nb et qui permet ainsi de retrouver facilement l'objet Hedge correspondant. Nous avons choisi l'heuristique suivante :

$$\text{clé} = (KA * min(na, nb) + KB * max(na, nb))\%np$$
$$\text{avec np} = \text{le nombre de points du maillage, } KA = 7 \text{ et } KB = 11$$

Cette convention de calcul de clé permet statistiquement d'éviter les collisions. Le champs nxt permet ainsi de gérer le cas d'une collision puisque si lors d'une recherche, le champ Hedge ne contient pas (na,nb), on pourra alors facilement voir le l'autre objet qui a la même clé chaîné par nxt.

Le remplissage du tableau global de type Hedge se fait donc grâce à 2 boucles for imbriquées, l'une qui parcourt l'ensemble des triangles et l'autre qui pour chaque triangle parcourt les 3 points de chaque triangle. A chaque itération sont calculées les arêtes et les relations d'adjacence qui sont rangées dans la table de hachage à l'indice créé par le calcul de la clé.

5.3 Création des relations d'adjacence

Tous les objets Hedge que nous avons créés précédemment nous permettent maintenant de calculer simplement et rapidement les relations d'adjacence. En effet, si nous avions du les calculer de manière "brute", on aurait du faire une double boucle for, le calcul aurait ainsi une complexité quadratique ce qui n'est concrètement pas imaginable au vu du temps de calcul que cela représenterait. Dans le cas de l'utilisation des objets Hedge, le temps de

calcul est quasi linéaire (en effet comme nous l'avons dit plus tôt la probabilité d'avoir des conflits est très faible).

La création des relations d'adjacences consiste simplement en une classification différente des éléments calculés dans la fonction de hash vue plus haut. En effet cette fonction revoie un tableau de type

```
int adja[3*mesh->nt+1]
```

Cette fonction se compose donc de 2 boucles for (comme pour la génération de la table de hash), la première parcourt tous les triangles et la dixième parcourt les 3 points de chaque triangle. A chaque itération, on s'intéresse uniquement aux champs adj1 et adj2 des objets Hedge. Aussi le champs adj1 de cet objet est la relation d'adjacence de l'arête j dans le triangle i et l'autre est la relation recherché qui est donc stockée dans

```
adja[3*(i-1)+1+j]
```

Il devient ainsi très facile de récupérer en une ligne de code le voisin d'un triangle grâce au tableau d'adjacence généré.

5.4 Localiser un point dans un maillage

Dans le cas de coordonnées cartésiennes, il est trivial de récupérer le numéro du triangle. Néanmoins cette opération n'est pas facile à transposer dans le cas d'un maillage triangulaire si on veut une implémentation efficace, c'est à dire qui n'utilise pas de techniques "force brute" de complexité au moins quadratique.