Notes du TP du cours de Méthodes d'adaptation pour la simulation numérique (NM491)

I. Questions de structures de données et de notations

Ce qui suit décrit brièvement les structures de base qui préexistent dans le code C fourni. Bien entendu, elles sont vouées à être enrichies au gré des besoins.

- La structure de point Point comporte initialement deux champs importants : x et y (qui peuvent être des entiers ou des flottants, suivant le contexte), ses coordonnées cartésiennes. Ainsi, si p0 est un objet de type pPoint, p0→x et p0→y sont les deux commandes qui permettent d'accéder à ses coordonnées.

- La structure dédiée aux triangles, Tria se ramène principalement à un tableau de trois entiers v[3]. Si pt est un objet de type pTria, pt→v[0], pt→v[1] et pt→v[2] sont les trois numéros de ses sommets. 0,1 et 2 sont les indices *locaux* des sommets du triangle (la numérotation est propre à chaque triangle), alors que pt→v[0], pt→v[1] et pt→v[2] sont les indices *globaux* de ses sommets (qui permettront d'accéder aux points correspondants, dans le tableau de tous les points du maillage : voir un peu plus bas).

- Un maillage (structure Mesh) est pour l'essentiel un tableau de sommets (c'est le champ Point ∗ point) et un tableau de triangles (c'est le champ Tria ∗ tria). Soit mesh un objet de type pMesh. mesh→np est le nombre total de points du maillage. Le tableau des points mesh→point permet d'accéder à ces points. Bien qu'en langage C, il soit d'usage de démarrer les tableaux à l'indice 0 (et donc de terminer par l'indice 'taille du tableau -1), pour diverses raisons, il est plus commode ici de démarrer le tableau mesh→point à l'indice 1, et de le terminer à l'indice mesh→np. Ainsi les points du maillage seront :

  mesh→point[1], ..., mesh→point[k], ..., mesh→point[mesh→np].

  Bien sûr, le Point mesh→point[0] existe physiquement, mais il n'est jamais utilisé.

  De manière analogue, mesh→nt est le nombre total de triangles du maillage. Le tableau des triangles mesh→tria est également numéroté de 1 à mesh→nt.

II. Relations d'adjacences d'un maillage

Lorsque l'on travaille sur un maillage, et en fonction dudit travail, on peut avoir besoin de beaucoup plus de renseignements que les seules coordonnées des points de ce maillage, ou les sommets qui constituent chaque triangle pris un par un. Par exemple, un calcul par éléments finis *basique* (disons par éléments finis $\mathbb{P}^1$ de Lagrange) demande a minima la connaissance de tous les triangles qui ont pour sommet un point donné (c'est le support de la 'fonction-chapeau' associée au point). Pour le travail que l'on va être amenés à effectuer, on aura besoin de beaucoup plus d'informations que cela, et il ne faut pas compter tout pouvoir stocker dans la mémoire !

L'idéal est de stocker quelques informations 'minimales', à partir desquelles on pourra reconstruire toutes les autres dont on aura besoin. Celles que l'on propose de déterminer et stocker sont les *relations d'adjacence* du maillage. Plus précisément, pour un triangle donné, on souhaite stocker

ses trois voisins (un par chaque arête dans le cas d'un maillage conforme), ainsi que la manière dont chacun de ces triangles adjacents 'voit' le triangle.

Pour stocker ces informations sous une forme compacte, on va associer trois entiers à chaque triangle du maillage (un pour chaque voisin). Considérons le cas de la figure 1.

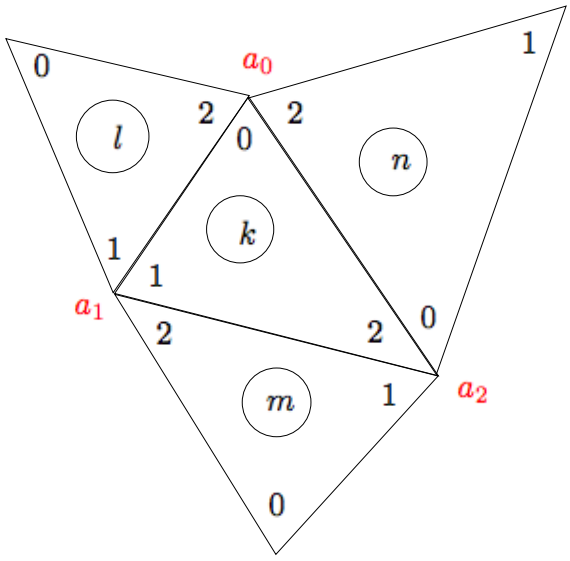


FIGURE 1

Ici, on a quatre triangles k, l, m et n. Les entiers 0,1 et 2 correspondent aux numérotations *locales* des trois sommets de chaque triangle. a_0, a_1 et a_2 sont les numéros des sommets de k (en numérotation *globale*).

En numérotation *locale*, il est d'usage de noter les arêtes d'un triangle en leur attribuant le numéro du sommet opposé. Ainsi, l'arête 0 du triangle k est l'arête (a_1, a_2) en numérotation globale (ou $(1, 2)$ en numérotation locale). De même, l'arête 2 du triangle k est l'arête (a_0, a_1) en numérotation globale (ou $(0, 1)$ en numérotation locale).

La règle que l'on propose de suivre pour la notation des relations d'adjacence est la suivante (mais elle n'a rien d'obligatoire !) : si un triangle k a pour voisin un triangle l par l'arête $i = 0, 1$ ou 2, et que ce voisin l a pour voisin le triangle k par l'arête $j = 0, 1$ ou 2, on traduira cette relation par

$$(3k + i) \text{ a pour voisin } (3l + j)$$

La connaissance de l'entier $(3l + j)$ permet de recouvrer le triangle voisin de k par l'arête i (il suffit de faire $l = (3l + j)/3$, où / désigne la division euclidienne entière) et le numéro de l'arête i de k vue de l (qui est encore l'indice du point opposé dans l appelé *voyeur*), par l'opération $j = (3l + j)\%3$ (% étant le modulo).

Exemple : Dans le triangle k :

- $(3k + 0)$ a pour voisin $(3m + 0)$.
- $(3k + 1)$ a pour voisin $(3n + 1)$.

- $(3k+2)$ a pour voisin $(3l+0)$.

III. Construction des relations d'adjacence d'un maillage

À partir d'un maillage donné sous la forme minimale d'une liste de points, et d'une liste de triangles (voir la section *I.* pour le format de ces données), plusieurs techniques sont possibles pour construire ses relations d'adjacence. Si l'on s'y prend de manière 'naïve', on est conduits à réaliser une double boucle imbriquée sur les triangles du maillage : on obtient quelque chose comme suit :

1: **for** $i = 1, ..., nt$ **do**
2: **for** $j = 1, ..., nt$ **do**
3: Tester si le triangle T_i est voisin du triangle T_j.
4: **end for**
5: **end for**

---

dont la complexité est quadratique, ce que est inutilisable en pratique.

Il est possible, en ayant recours à une astuce très fréquente en programmation, de réaliser cette construction des relations d'adjacence en un coût de calcul linéaire en le nombre de triangles du maillage. Il faut utiliser une *table de hachage*.

En toute généralité, on a recours à une telle structure à chaque fois que l'on souhaite stocker des informations relatives à des données dont la numérotation naturelle n'est pas adaptée à un stockage optimisé (voir Wikipedia pour des détails généraux, ou le livre [1] pour divers exemples d'utilisation dans le contexte du maillage). Il en va ainsi des numéros de téléphone dans l'annuaire, auxquels on souhaite associer des noms de personne, des adresses,...

Ici, les relations d'adjacence vont être construites par un parcours astucieux des arêtes du maillage. L'idée est que si l'on parcourt tous les triangles, puis pour chaque triangle, chacune de ses trois arêtes, chaque arête du maillage est vue exactement deux fois (sauf pour les arêtes du bord), une pour chaque triangle auxquels elle appartient. Ainsi, en stockant toutes les arêtes du maillage, et pour chaque arête les triangles par lesquels on les a vues, on est capable de mettre en rapport les triangles voisins l'un de l'autre.

Le problème est que la manière 'naturelle' de numéroter les arêtes d'un maillage est de les considérer comme un couple (na, nb), na et nb étant les deux extrémités de l'arête (dans la numérotation globale des points du maillage). Cette manière de les numéroter est particulièrement malcommode : si l'on veut pouvoir stocker toutes les arêtes sous cette forme, il faut prévoir un tableau de taille $np \times np$, np étant le nombre de points du maillage (ce qui en pratique est beaucoup trop lourd !), alors que, bien entendu, tous les couples de points (na, nb) ne sont pas des arêtes du maillage.

C'est là qu'intervient le hachage, qui va permettre de stocker un peu plus astucieusement ces arêtes : à une arête (na, nb), on associe un objet d'une structure plus grosse, que l'on peut appeler Hedge :

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

Ici, $ia = min(na, nb)$, $ib = max(na, nb)$, adj1 et adj2 sont destinés à recueillir les deux relations d'adjacence par lesquelles l'arête a été vue, et nxt va servir à chaîner les objets Hedge, comme on va l'expliquer un peu plus loin. Sur l'exemple de la figure 1, à l'arête (a_1, a_2) sera associée, après parcours, l'objet Hedge

$$\{min(a_1, a_2), max(a_1, a_2), 3m + 0, 3k + 0, nxt\}.$$

C'est lorsque l'on aura construit tous les objets Hedge associés à toutes les arêtes du maillage que l'on sera en mesure de reconstruire les relations d'adjacence du maillage.

Ces objets Hedge sont rangés dans un tableau global

```
Hedge *tab;
```

À chaque arête (na, nb) est associée une *clé de hachage* key, qui se calcule simplement à partir des extrémités na et nb, et qui va permettre de retrouver l'objet Hedge correspondant. Un choix courant de fonction de hachage est :

$$\text{key} = (KA * min(na, nb) + KB * max(na, nb)) \,\%\, hsize,$$

$$KA = 7,\ KB = 11\ ;\ hsize = \text{mesh} \rightarrow \text{np}$$

il s'agit d'une heuristique qui 'statistiquement', permet d'éviter au maximum les *collisions* : en effet, en pratique, cette manière de numéroter les arêtes du maillage par l'entier key n'a aucune chance d'être injective. En d'autres termes, on ne peut pas rechercher l'objet Hedge associé à (na, nb) par la commande

```
tab[key].
```

En revanche, on peut, grâce au champ nxt de la structure Hedge s'imposer que tous les objets Hedge des arêtes qui partage une même valeur de clé soient chaînés. Ainsi, si l'arête associée à tab[key] n'est pas (na, nb), on ira voir la suite de la chaîne :

- on passe à l'objet suivant en allant voir l'élément suivant de la chaîne : key = tab[key].nxt
- on teste si l'arête associée est celle que l'on cherche par

$$(\text{tab[key]}.ia == min(na, nb))\&\&(\text{tab[key]}.ib == max(na, nb)).$$

Si la fonction de hachage est convenablement choisie (c'est l'avantage de l'heuristique proposée plus haut), on peut espérer que le nombre d'arêtes qui partagent une valeur de clé donnée est assez faible. Alors, la recherche d'une arête du maillage dans cette structure aura un coût constant.

Concrètement, pour utiliser cette technique dans la construction des relations d'adjacence, on propose de procéder comme suit, en codant deux (petites) fonctions (mais toute autre méthode est la bienvenue !) :

- Écrire une fonction

  ```
  int hashHedge(pMesh mesh)
  ```

  qui progressera comme l'algorithme 1 :

- Écrire une fonction

  ```
  int setAdj(pMesh mesh)
  ```

  qui progressera comme l'algorithme 2 :

IV. Implémentation d'autres opérations de base

La connaissance des relations d'adjacence du maillage va maintenant permettre d'implémenter rapidement et efficacement toutes les autres opérations de base que l'on peut être amené à exécuter. Les

Algorithm 1 Construction de la table de hachage

1: Allouer le tableau de tous les objets Hedge par la commande :

tab = (Hedge*)calloc(3*mesh→nt+1,sizeof(Hedge));

2: Initialiser une variable $hnxt$, qui donnera le premier indice disponible dans le tableau tab : hnxt = $mesh \to np + 1$.
3: **for** $i = 1, ..., nt$ **do**
4: **for** $j = 0, 1, 2$ **do**
5: Pour chaque telle arête (na, nb) du maillage, calculer la clé key, ainsi que la relation d'adjacence $(3 * i + j)$ adj par laquelle cette arête est vue dans le triangle en question.
6: Rechercher dans le tableau tab si tab[key] existe.
7: **if** tab[key] n'existe pas **then**
8: faire tab[key].$ia = min(na, nb)$, tab[key].ib = ..., adj1 = adj.
9: **else**
10: Rechercher si l'un des éléments de la chaîne correspond à (na, nb). S'il y en a un, son champ $adj1$ a déjà été rempli. Si non, aller jusqu'au bout de la chaîne (jusqu'à ce qu'il n'y ait plus de nxt), prendre le premier élément disponible dans tab (qui est donc celui d'indice hnxt), et remplir le champ Hedge associé. Faire pointer le dernier tab[key] visité vers ce nouvel élément, par tab[key].nxt = hnxt. Incrémenter hnxt.
11: **end if**
12: **end for**
13: **end for**

Algorithm 2 Construction des relations d'adjacence

1: Allouer un champ dans la structure Mesh qui va stocker les relations d'adjacence :

mesh→adja = (int *)calloc(3*mesh→nt+1,sizeof(int));

La règle proposée est que, pour chaque triangle $k = 1, ..., nt$, mesh→adja$[3 * (k - 1) + 1 + j]$ $(j = 0, 1, 2)$ sera la relation d'adjacence associée à l'arête j de k, vue dans le triangle voisin.
2: **for** $i = 1, ..., nt$ **do**
3: **for** $j = 0, 1, 2$ **do**
4: Calculer la clé associée à l'arête (na, nb).
5: Parcourir la chaîne du tableau tab associée à cette valeur de clé, jusqu'à ce que l'objet Hedge associé à (na, nb) soit trouvé.
6: L'un des deux champs adj1, adj2 de cet objet est la relation d'adjacence de l'arête dans le triangle i. L'autre est la relation recherchée, à placer dans mesh→adja$[3 * (i - 1) + 1 + j]$
7: **end for**
8: **end for**

sections suivantes précisent quelques-unes des plus courantes, et suggèrent des possibilités quant à leur implémentation (même si, encore une fois, cela ne doit être compris que comme des suggestions !).

1. *Localiser un point dans un maillage*

Localiser un point dans un maillage -i.e. récupérer le numéro du triangle du maillage auquel il appartient à partir de ses coordonnées cartésiennes - est l'exemple typique d' opération triviale dans le cas d'un grille cartésienne, mais qui n'est pas si évidente à transposer dans le cas d'un maillage triangulaire quelconque, si l'on veut que l'implémentation soit assez efficace (il ne s'agit

pas de parcourir tous les triangles du maillage pour s'arrêter au premier qui contient le point !).

Une idée intéressante est d'utiliser les *coordonnées barycentriques* dans les triangles du maillage, pour parcourir celui-ci de manière efficace à partir d'un triangle initial que l'on suppose 'pas trop loin' du triangle qui contient le point.

Si $T \in \mathcal{T}$ est un triangle du maillage de sommets a_0, a_1, a_2, et x est un point, ses *coordonnées barycentriques* constituent l'unique couple de réels $(\lambda_0(x), \lambda_1(x), \lambda_2(x))$ tels que :

$$\begin{cases} \lambda_0(x)a_0 + \lambda_1(x)a_1 + \lambda_2(x)a_2 &= x \\ \lambda_0(x) + \lambda_1(x) + \lambda_2(x) &= 1 \end{cases}.$$

On sait alors que $x \in T$ si et seulement si ses trois coordonnées barycentriques dans T sont positives. En outre, si $\lambda_i(x) \leq 0$ pour un certain indice i, alors x appartient au demi-plan délimité par l'arête opposée à a_i, et situé du côté opposé à a_i. Partant de cette idée :

(1) Écrire une fonction

```
int baryCoord(pMesh mesh, int it, double c[2], double cb[3]);
```

qui à un point x repéré par ses coordonnées cartésiennes c[2], associe ses coordonnées barycentriques dans le triangle de numéro global it, et les place dans le tableau cb (on pourra prendre comme convention que a_0 = pt → v[0], ... pour la numérotation des coordonnées barycentriques).

(2) En déduire une fonction

```
int locElt(pMesh mesh, int start, double c[2], double cb[3]);
```

qui retourne le numéro d'un triangle du maillage mesh auquel appartient le point x repéré par ses coordonnées cartésiennes c[2]. Cet algorithme progressera par adjacence dans le maillage, à partir d'un triangle initial start passé en argument, en utilisant les informations véhiculées par le signe des coordonnées barycentriques pour arriver au triangle final. On retournera également dans cb les coordonnées barycentriques de x dans le triangle d'arrivée.

2. *Calculer la boule d'un point du maillage*

Étant donné un nœud x d'un maillage $\mathcal{T}$, il est très fréquent que l'on ait besoin de connaître les triangles de $\mathcal{T}$ qui partagent ce sommet, ce que l'on appelle parfois la *boule* du nœud x (penser aux fonctions de base de l'espace des éléments finis $\mathbb{P}^1$ de Lagrange, dont ces boules sont les supports).

À vrai dire, le cas le plus fréquent est celui où l'on connaît x comme sommet d'un triangle T du maillage, et où l'on souhaite connaître les autres triangles qui partagent ce sommet. La fonction dont on a alors besoin, et que l'on pourra implémenter est alors :

```
int boulep(pMesh mesh, int start, char i0, int *list);
```

Ici, on cherche à énumérer la boule du sommet i_0 (en numérotation locale) du triangle start. La valeur de retour $ilist$ de cette fonction est le nombre de ces triangles, et ces triangles sont stockés dans le tableau d'entiers list sous la forme 'pratique' suivante : pour tout $i = 0, ..., ilist - 1$, $list[i] = 3\,k + j$, où k est le $i-$ème triangle de la boule du point considéré, et $j = 0, 1$, ou 2 est le numéro local de ce point dans le triangle k (voir la figure 2).

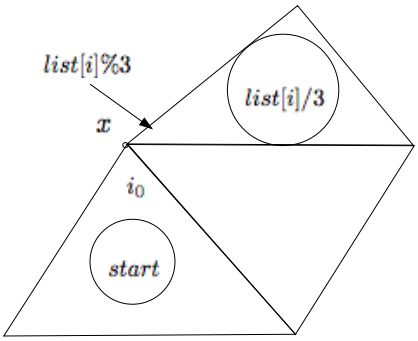


FIGURE 2

V. Implémentation des opérateurs de modification locale du maillage

1. *Insertion d'un point dans un maillage par l'algorithme de la cavité de Delaunay*

2. *Suppression, ou 'collapse' d'un point d'un maillage*

La suppression d'un point du maillage intervient lorsque l'on souhaite décimer celui-ci, pour une raison ou pour une autre. Il s'agit moralement de détruire une arête du maillage en fondant l'une de ses extrémités -disons p - sur l'autre, q : le point p disparaît dans la procédure, ainsi que les deux triangles qui partagent l'arête $[pq]$. Les autres triangles de la boule de p voient simplement ce sommet p transformé en q : voir la figure 3.

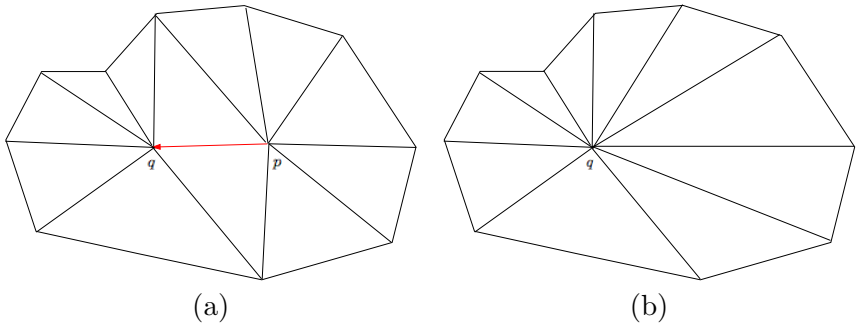


FIGURE 3. (a) Avant la suppression du point p, (b) après cette suppression.

Malheureusement, il arrive que l'on rencontre des configurations où mener une telle opération n'est pas possible : on peut penser au cas où la boule du point p n'est pas étoilée par rapport au point q : les nouvelles arêtes crées dans le processus peuvent tout à fait traverser d'autres triangles du maillage que la seule boule de p, ce qui va conduire à un maillage non conforme. On propose pour cette raison d'implémenter cette opération en deux temps :

(1) Écrire une fonction

```
int chkCol(pMesh mesh, int ilist, int *list) ;
```

qui analyse la configuration où l'on souhaite faire disparaître le sommet dont on donne la boule dans le tableau list (qui est de taille ilist), sur le sommet suivant dans la numérotation

locale du premier triangle de cette boule. La fonction retourne 1 si le fondu du point est possible - i.e. ne conduit pas à la production d'éléments invalides - et 0 sinon, mais ne procède à aucune mise à jour.

(2) Écrire une fonction

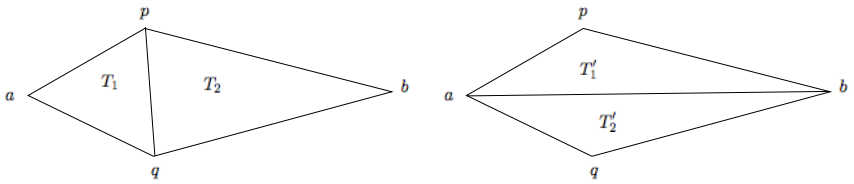
```
int collapse(pMesh mesh, int ilist, int *list) ;
```

qui procède au 'collapse' du point à proprement parler : il s'agit d'une part de mettre à jour les sommets des triangles concernés, d'autre part de mettre à jour les relations d'adjacence qui sont devenues obsolètes.

3. *Bascule d'une arête d'un maillage*

La bascule d'une arête d'un maillage est particulièrement intéressante pour changer sa connectivité, ou plus généralement améliorer sa qualité. C'est le seul parmi ces trois opérateurs dont la mise en œuvre diffère vraiment au passage de la dimension 2 à la dimension 3.

En dimension 2, les choses sont très simples : deux triangles $T_1 = (apq)$ et $T_2 = (bpq)$ partagent l'arête $[pq]$ (voir la figure 4). L'opérateur de bascule d'arête consiste à transformer $[pq]$ en l'arête $[ab]$, et donc à transformer T_1 en $T_1' = (pab)$ et T_2 en $T_2' = (qab)$.



(a) (b)

FIGURE 4. (a) Avant la bascule de l'arête $[pq]$, (b) après la bascule.

Malheureusement, toutes les situations ne se prêtent pas à la bascule d'arête. Si l'on ne prend pas attention, certains éléments peuvent devenir invalides, comme cela est présenté sur la figure 5.

Pour mettre en place cet opérateur, on pourra donc procéder en deux temps :

(1) Écrire une fonction

```
int chkSwp(pMesh mesh, int it, char i0) ;
```

qui analyse la configuration où l'on souhaite basculer l'arête i_0 du triangle it (une seule configuration est possible dans ce contexte). La fonction retourne 1 si la bascule est possible - i.e. ne conduit pas à la production d'éléments invalides - et 0 sinon, mais ne procède à aucune mise à jour.

(2) Écrire une fonction

```
int swap(pMesh mesh, int it, char i0) ;
```

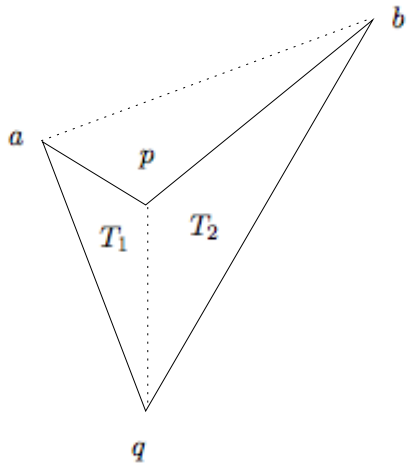


FIGURE 5

qui réalise la bascule 'physique' de cette arête, dans le cas où la fonction précédente aura retourné 1. Des mises à jour de deux ordres sont à traiter : il s'agit de changer les numéros des sommets des deux triangles de la configuration, *mais aussi* de mettre à jour les relations d'adjacence du maillage, qui sont devenues obsolètes.

REFERENCES

[1] P.J. FREY AND P.L. GEORGE, *Mesh Generation : Application to Finite Elements*, Wiley, 2nd Edition, (2008).