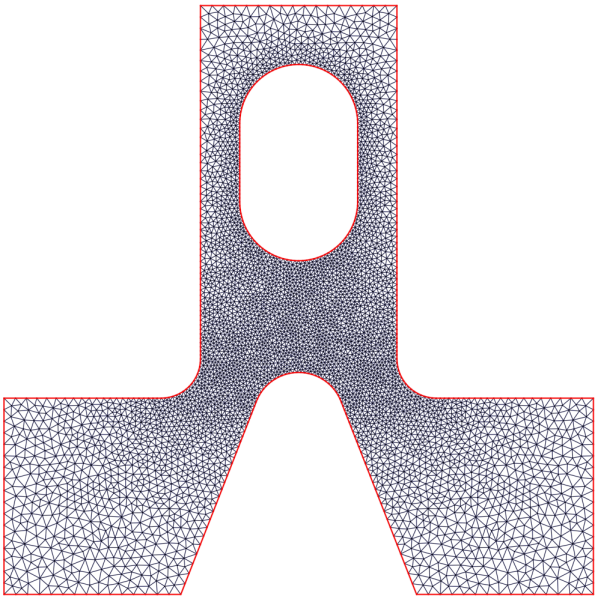
Finite elements for dummies : Edges

Louis Devillez
Jérome Eertmans

Ce document a pour but daider les étudiants du cours d'Éléments Finis, donné par MM. Vincent Legat et Jean-François Remacle à l'Ecole Polytechnique de Louvain, en leur offrant une solution détaillée aux devoirs.



1 Le problème à résoudre

Cette semaine encore, votre contribution sera répartie sur plusieurs petites fonctions qui, une fois assemblées, permettront de calculer le périmètre d'un maillage.

Dans cet objectif, nous allons devoir compléter 4 fonctions :

1. Une fonction qui va assigner, à chaque arête, l'indice du triangle dans lequel elle apparaît

```
void edgesExpand(femEdges *theEdges);
```

2. Une fonction pour comparer deux arêtes, sur base de leurs noeuds respectifs

```
int edgesCompare(const void* e0, const void *e1);
```

3. Une fonction qui va fusionner toutes les arête égales mais appartenant à deux triangles différents

```
void edgesShrink(femEdges *theEdges);
```

4. Une fonction qui calculera le périmètre du maillage

```
double edgesBoundaryLength(femEdges *theEdges);
```

2 C quoi les # partout ?

Vous l'avez peut être lu sur le site du prof, mais il ne faut pas supprimer les instructions de type `# ifndef NOEXPAND`, etc. Mais pourquoi ? Voici une question bien légitime à laquelle on va tenter de répondre :-)

2.1 Définir pendant la compilation

De manière très résumée, toutes les instructions précédées d'un `#` sont evaluées pendant le processus de compilation. Cela permet au compilateur d'optimiser votre code en fonction de certaines conditions, établies en amont. C'est grâce à cela que vous pouvez importer des fonctions définies dans d'autres fichiers, en incluant leur header. Comme tout ceci est réalisé lors de la compilation, ce qui est défini par ces instructions doit être vu comme constant.

Dans le cas de l'évaluation du devoir, ceci est très utile car le professeur peut tester vos fonctions, une par une, sans devoir toucher à votre code : il devra juste modifier son fichier à lui qui définira si le test doit utiliser une fonction de votre code (fonction à tester) ou une fonction de son code (fonction de référence). C'est très pratique dans le cas où vous avez seulement une partie de vos fonctions qui est fonctionnelle, et que vous voudriez quand même être évalué sur la partie qui marche ;-)

2.2 Définir des constantes

Une grande utilité de l'instruction `#` est la définition de constantes. D'ailleurs, c'est comme ça qu'est définie la variable `M_PI` (π) dans `math.h`. Ce qui est défini après un `#` peut être vu comme une chaîne de caractères qui va être intégrée, par le compilateur, partout où votre constante sera utilisée. De ce fait, ces constantes n'ont pas de typage.

En d'autres mots, le code

```
#define PI 3

double monAire(double rayon) {
  return rayon * rayon * PI;
}
```

sera **d'abord** remplacé par

```
double monAire(double rayon) {
  return rayon * rayon * 3;
}
```

pour, **ensuite**, être compilé.

2.3 Les macros, vos fonctions super-rapides

Comme expliqué plus-haut, ce qui est défini après une instruction `#` peut être essentiellement vu comme du texte qui va être remplacé partout dans votre code où le mot défini sera utilisé. Il est également possible d'utiliser cette instruction pour déclarer des fonctions, qu'on appelle des macros.

Les macros sont fortement utilisées quand on doit utiliser une fonction assez courte de manière très récurrente. En `C`, l'appel d'une fonction induit un overhead, i.e., un coût supplémentaire, lié au fait que

le code de la fonction est stocké quelque part en mémoire et qu'il faut charger ce code, tout en envoyant les paramètres requis à la fonction. L'avantage des macros est de retirer tout ce coût supplémentaire en ré-écrivant pour vous, partout où il le faut, le code de votre fonction.

De manière classique, on définira souvent des fonctions comme `MIN` et `MAX` en utilisant des macros :

```
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))
```

Remarque : la notation `x ? y : z` est équivalente à écrire

```
if (x) {
  return y;
}
else {
  return z;
}
```

Attention : les parenthèses ici sont très importantes. Comme l'intégralité du contenu d'une macro est recopié bêtement dans votre code, mettre des parenthèses assure que l'ordre des opérations attendu sera conservé. Vous devez faire pareil si vous définissez des constantes du style `(1/3)`, `(5/4)`, etc.

2.4 `#if`, `#else` et `#elif`

Disons que vous souhaitez tester deux implémentations d'une fonction afin de savoir laquelle est la plus rapide. Une solution assez chronophage serait de définir deux fois votre fonction et de commenter celle que vous ne voulez pas tester :

```
// double maNoteEnjuin(double *notesDevoirs, int nDevoirs) {
//  double sum = 0.0;
//  for (int i=0; i<nDevoirs; i++) sum += notesDevoirs[i];
//
//  return sum / nDevoirs;
// }

double maNoteEnjuin(double *notesDevoirs, int nDevoirs) {
  // Version optimale
  return 20.0;
}
```

Une version beaucoup pratique, serait d'utiliser les instructions `#if` et `#else` afin de pouvoir contrôler, via la constante `VERSION_NULLE`, le code que l'on souhaite compiler (et donc exécuter). Cette constante ne doit d'ailleurs pas obligatoirement être définie dans le même fichier que les fonctions.

```
#define VERSION_NULLE = 0 // 1 - true, 0 - false

#if VERSION_NULLE
double maNoteEnjuin(double *notesDevoirs, int nDevoirs) {
  double sum = 0.0;
  for (int i=0; i<nDevoirs; i++) sum += notesDevoirs[i];
  return sum / nDevoirs;
}
#else
double maNoteEnjuin(double *notesDevoirs, int nDevoirs) {
```

```
  // Version optimale
  return 20.0;
}
#endif
```

3 Résolution du problème

Avant de commencer, on va regarder les nouvelles structures introduites :

```
typedef struct {
  int elem[2];
  int node[2];
} femEdge;

typedef struct {
  femMesh *mesh;
  femEdge *edges;
  int nEdge;
  int nBoundary;
} femEdges;
```

`femEdge` va représenter une arête dans notre maillage. Cette arête est définie par 2 points (`node`) et par le ou les triangle(s) dont elle fait partie (`elem`). De fait, une arrête peut être à l'extrémité de notre maillage et donc ne border qu'un seul élément. Dans ce cas, on assignera la valeur `-1` au second élément de `elem`.

`femEdges` va représenter l'ensembles des arêtes de notre maillage avec quelques infos comme le nombre total d'arêtes et celles situées à l'extrémité du maillage, ainsi qu'un pointeur vers le maillage en question.

3.1 edgesExpand

Dans cette fonction, à partir de la structure `femEdges`, on veut construire toutes les arêtes qui existent, même les doublons, c'est à dire `3*nElem` arêtes (vu qu'on travail avec des triangles). Dans un premier temps, on va considérer que toutes les arêtes sont à l'extrémité du maillage et ça sera à l'aide des autres fonctions qu'on va réduire ce nombre.

```
void edgesExpand(femEdges *theEdges) {
  int a, b;
  theEdges->nBoundary = theEdges->nEdge;
  for(int i = 0; i < theEdges->mesh->nElem; i++){
    for(int j = 0; j < 3; j++){
      a = theEdges->mesh->elem[3*i+j];
      b = theEdges->mesh->elem[3*i+(j+1)%3];
      // Ici on trie en amont, on explique pourquoi apres :)
      theEdges->edges[3*i+j].node[0] = MIN(a, b); // node[0] <=
      theEdges->edges[3*i+j].node[1] = MAX(a, b); // node[1]
      theEdges->edges[3*i+j].elem[0] = 3*i;
      theEdges->edges[3*i+j].elem[1] = -1;
    }
  }
}
```

On commence le code de la fonction en déclarant les variables. La mémoire pour les `3*n` arêtes a déjà été allouée. Il faut noter ici que `edges` est un pointeur `femEdge*`, c'est à dire que c'est l'adresse d'une structure `femEdge`. L'utilisation du pointeur nous permet donc de définir un tableau mais c'est un tableau de `femEdge` et non de `femEdge*`. Donc, petit rappel de la semaine précédente, si on veut accéder à un attribut d'une structure on doit utiliser l'opérateur `.` et non `->`.

Une autre chose à remarquer, c'est qu'on a voulu utiliser 2 boucles `for` pour éviter de devoir écrire 3 fois la même chose mais ça nous impose d'utiliser l'opérateur modulo (`%`) pour récupérer le noeud `b`. Également, on utilise les macros `MIN` et `MAX` alors que ça n'était pas demandé. Cela nous permet d'être certain d'avoir que `node[0]` est plus petit que `node[1]` et ça aura de l'importance pour la prochaine fonction.

3.2 edgesCompare

Dans cette fonction, on veut pouvoir comparer 2 arêtes. Cependant, si on regarde la définition de la fonction, on voit qu'on utilise des pointeurs `void*` :

```
int edgesCompare(const void* e0, const void *e1);
```

Ces types de pointeurs sont demandés par la fonction `qsort` qu'on utilise. Et pourquoi cette fonction veut des pointeurs `void*` et pas `femEdge*` ? C'est tout simplement pour être général : on peut convertir un pointeur `void*` en un autre type de pointeur (cependant il faut bien faire attention à ce que l'on fait). Ça nous permet donc d'utiliser la fonction `qsort` avec n'importe quel type de variable.

Si on sait que, pour n'importe quelle arête, `node[0]` est plus petit que `node[1]` alors, en suivant les specifications données par votre gentil professeur, il est simple d'arriver au code suivant :

```
int edgesCompare(const void* e0, const void *e1) {
  int diagnostic = ((femEdge*) e0)->node[0] - ((femEdge*) e1)->node[0];
  if(diagnostic != 0){
    return diagnostic;
  }
  else {
    return ((femEdge*) e0)->node[1] - ((femEdge*) e1)->node[1];
  }
}
```

Il ne faut pas oublier de transformer notre `void*` en `femEdge*` avec l'opération suivante : `(femEdge*) e0`. Cette opération s'appelle un typecast, c'est à dire changer le type d'une variable. Notez que l'on compare les `node[1]` uniquement si les `node[0]` sont égaux.

Cependant, vous deviez écrire une fonction générale donc qui ne se base pas sur le fait que `node[0]` soit d'office plus petite que `node[1]`. Il faut donc généraliser. Ici, on ne va pas supprimer notre code mais on va utiliser ce que vous on appris plus haut :

```
int edgesCompare(const void* e0, const void *e1)
{
#if FAST // Code optimise
  int diagnostic = ((femEdge*) e0)->node[0] - ((femEdge*) e1)->node[0];
  if(diagnostic != 0){
    return diagnostic;
  }
  else {
    return ((femEdge*) e0)->node[1] - ((femEdge*) e1)->node[1];
```

```
  }
#else // Code non optimise
  int nodeMin0 = MIN(((femEdge*) e0)->node[0],((femEdge*) e0)->node[1]);
  int nodeMin1 = MIN(((femEdge*) e1)->node[0],((femEdge*) e1)->node[1]);
  int diagnostic = nodeMin0 - nodeMin1;
  if(diagnostic != 0){
    return diagnostic;
  }
  else {
    int nodeMax1 = MAX(((femEdge*) e1)->node[0],((femEdge*) e1)->node[1]);
    int nodeMax0 = MAX(((femEdge*) e0)->node[0],((femEdge*) e0)->node[1]);
    return nodeMax0 - nodeMax1;
  }
#endif
}
```

Donc, pour rester générique, on réutilise les macros vues avant : `MIN` et `MAX`, mais le principe reste le même. Maintenant on peut se poser la question de pourquoi s'enquiquiner à trier les noeuds de nos arêtes si c'est pour utiliser une fonction générique par après. Ici on doit s'assurer que toutes vos fonctions marchent individuellement. Donc si vous faites petites optimisation inter-dépendantes de vos functions ça risque de coincer car le professeur ne peut pas s'assurer que vous ayez fait attention à bien trier votre maillage. Pour le projet, ça sera sûrement différent :-)

Mais, est-ce que l'optimisation était nécessaire ? Dans le cas où on applique les macros dans la fonction précédentes on aura `2*n` appels aux macros vu qu'on utilise 2 macros par arête. Si on les utilise dans `qsort`, on va appeler dans le meilleur des cas `2*n*log(n)` fois les macros et `4*n*log(n)` dans le pire cas. Ceci est directement lié à la complexité de l'algorithme de tri rapide. On étudiera cet impact plus en détail à la fin du document.

3.3 `edgesShrink`

Maintenant que nos arêtes sont créées et triées, il faut supprimer tous les doublons. Comment est défini un doublon ? Ce sont 2 arêtes dans notre `femEdges` qui ont les même noeuds. Celles-ci seront adjacentes vu que notre tableau est trié.

On prévoit ce qu'on veut faire pour être efficace :

1. Parcourir les arêtes en comparant les noeuds avec l'arête suivante
2. Si les noeuds sont les mêmes :
   - On remplace la valeur de `elem[1]` de notre première arête par la valeur de `elem[0]` de la seconde
   - On écrit cette arête à l'indice d'écriture
   - On incrémente notre indice d'écriture de 1 mais notre indice de lecture de 2 (on ne veut pas lire l'arête suivante vu que c'est un doublon)
   - On incrémente le nombre d'arêtes de 1
3. Sinon
   - On écrit l'arête actuelle à l'indice d'écriture
   - On incrémente le nombre d'arêtes et le nombre d'arêtes sur l'extrémité de 1
   - On incrémente l'indice de lecture et d'écriture de 1
4. Quand on a fini de parcourir, on libère la mémoire dont on a plus besoin via un `realloc`

```
void edgesShrink(femEdges *theEdges) {
  int n = 0;
  int nBoundary = 0;

  int indEcriture = 0;
  femEdge a,b;
  for(int i = 0; i < theEdges->nEdge; i++){
    a = theEdges->edges[i];
    b = theEdges->edges[i+1];
#if FAST // Ici, on profite aussi du tri en amont
    if(i != theEdges->nEdge-1 && (a.node[0] == b.node[0] && a.node[1] == b.node[1])
   ){
#else
    if(i != theEdges->nEdge-1 && (a.node[0] == b.node[0] && a.node[1] == b.node[1])
    || ( a.node[0] == b.node[1] && a.node[1] == b.node[0])){
#endif
     theEdges->edges[indEcriture] = theEdges->edges[i];
     theEdges->edges[indEcriture].elem[1] = theEdges->edges[i+1].elem[0];
     i += 1;
    }else {
     theEdges->edges[indEcriture] = theEdges->edges[i];
     nBoundary += 1;
    }

    indEcriture +=1;
    n += 1;
  }
  // Reallocation du tableau des edges

  theEdges->edges = realloc(theEdges->edges, n * sizeof(femEdge));
  theEdges->nEdge = n;
  theEdges->nBoundary = nBoundary;
}
```

Ici encore, il faut faire attention que, dans le cas général, on doit tester 2 égalités entre les noeuds et non une seule dans le cas où on a déjà trié nos noeuds dans `edgesExpand`.

`realloc` nous permet dire à la mémoire qu'on a besoin de moins d'espace et, donc, d'en libérer une partie.

3.4 edgesBoundaryLength

Maintenant qu'on peut faire la différence entre une arête d'extrémité et d'intérieur, on peut donc calculer le périmètre de notre maillage. Ici, rien de compliqué :

```
double edgesBoundaryLength(femEdges *theEdges)
{
  double L = 0;
  for(int i = 0; i < theEdges->nEdge; i++){
    if(theEdges->edges[i].elem[1] == -1){ // -1 = extremite
      int node0 = theEdges->edges[i].node[0];
      int node1 = theEdges->edges[i].node[1];
      double X = theEdges->mesh->X[node0] - theEdges->mesh->X[node1];
      double Y = theEdges->mesh->Y[node0] - theEdges->mesh->Y[node1];
      L += sqrt(X*X+Y*Y);
    }
```

```
12    }
13    return L;
14 }
```

3.5 Légère optimisation

On va regarder la différence de temps d'exécution avec et sans nos petites optimisations sur le fichier `holes.txt` vu qu'il contient beaucoup d'élements.

Étape	Pas d'optimisation [ms]	Avec optimisations [ms]
edgesExpand	4.48	4.49
edgesSort	75.6	70.3
edgesShrink	1.81	1.78
Total	81.9	76.6

Au final, on a bien les résultats attendus : notre fonction `edgeExpand` est légèrement plus lente mais le gain se fait ressentir sur `edgeSort` et `edgesShrink`.

Pour aller plus loin, on peut utiliser des outils tels que `perf` pour analyser plus en détail le temps d'éxécution (plutôt le nombre de cycle) pour avoir un résultat comme à la FIGURE 1. On y vois d'ailleurs que le temps d'exécution est en partie dominé par le tri des arêtes.

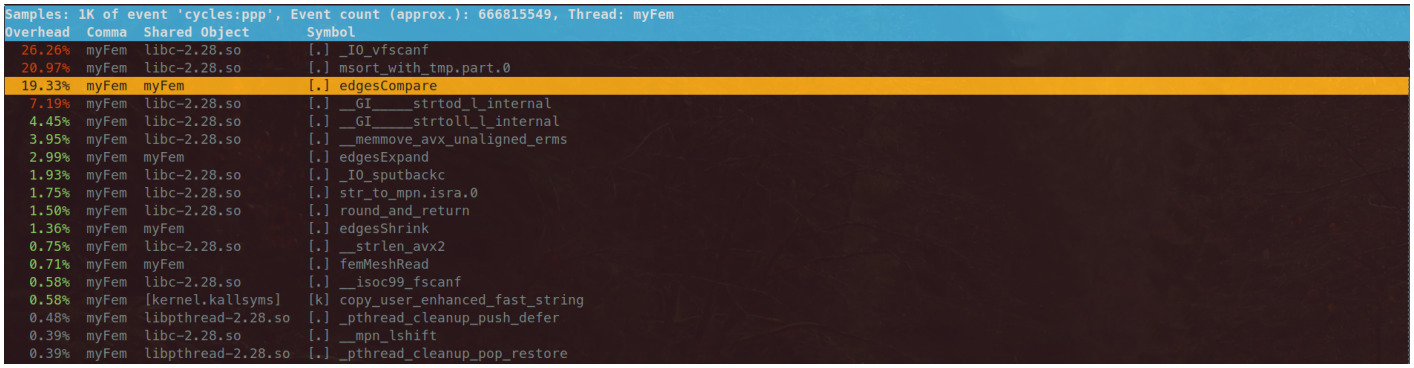


FIGURE 1 – Évaluation par l'outil `perf` des performaces de notre devoir.