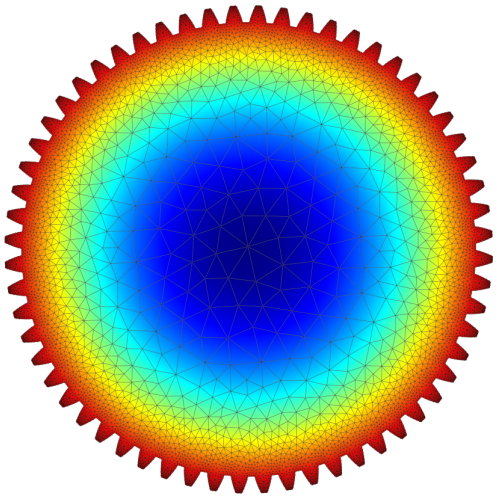
Finite elements for dummies : Inertia

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

La semaine passée, nous avons appris à intégrer numériquement une fonction à l'aide de la méthode de Hammer à 3 points. Cette semaine, nous allons apprendre à créer des maillages à partir d'un fichier `.txt`, afin d'intégrer ce dernier à l'aide d'une méthode d'intégration **modulable**, *i.e.,* qui peut être modifiée par l'évaluateur.

Plus particulièrement, nous allons devoir compléter 4 fonctions :

1. Une fonction retournant ρ, la masse volumique de l'acier

```
double inertiaGearSteelRho();
```

2. Une fonction pour créer, sur base d'un fichier, une structure qui va représenter notre maillage

```
femMesh *inertiaGearMeshRead(const char *filename);
```

3. Une fonction pour libérer correctement la mémoire allouée à cette structure

```
void inertiaGearMeshFree(femMesh *theMesh);
```

4. Une fonction pour intégrer l'inertie sur ce maillage

```
double inertiaGearInertia(femMesh *theMesh, femIntegration *theRule, double
    rho);
```

2 `C` comme *chouette mon code compile !*

2.1 Pro-tips

Avant de commencer à travailler sur ce devoir, il est conseillé de changer le maillage utilisé. De base, c'est le fichier `gear60.txt` qui est lu : celui-ci contient plus de 8000 noeuds et 13000 triangles. Si vous voulez débugger votre programme avec des `printf()` partout [1], vous allez avoir du mal à comprendre ce qui se passe vraiment. Pour cela, il vaut mieux utiliser le maillage `example.txt`.

Attention : les fichiers contenant les maillages sont encodés avec un chemin **relatif**. Comme son nom l'indique, le chemin exact vers votre fichier sera déterminé en fonction de **l'endroit d'où vous exécutez votre programme**. Pour ce devoir, il vous faut lancer votre programme depuis votre dossier `build` !

```
(InertiaGear) ./buid/myFem    # Cela ne fonctionne pas
(InertiaGear/build) ./myFem   # Cela fonctionne
```

2.2 Pointeurs

Avant de commencer ce devoir, nous allons vite rappeler la base des pointeurs et des structures. Précédemment, nous n'avions besoin que de variables simples, c'est à dire quelque chose du style :

```
int patate = 3; // Utile pour faire de bonnes frites
```

Le mot clé `int` indique le type de variable. Ici, on va travailler sur un entier où `patate` est le nom de la variable (qui est très mal choisi au passage). C'est un raccourci vers l'adresse de notre variable en mémoire. `3` est la valeur de notre variable actuellement.

Cependant, en n'utilisant que des types primitifs (`int`, `long`, `double`, `char`, etc.), on se retrouve parfois limité. Par exemple, on aimerait bien pouvoir retourner ou modifier un tableau d'éléments mais nos fonctions classique ne nous permettent que de retourner une seule variable à la fois.

L'idée des pointeurs est de pouvoir stocker l'adresse d'une variable dans une autre. **Mais pourquoi faire ?** Un tableau est un bloc de mémoire contigu, qui peut être vu comme un train où chaque wagon est un élément. Ainsi, si je déclare un tableau :

```
// Moyen pratique pour déclairer un tableau de taille constante
int patate2[3] = {42, 69, 666};
```

`patate2` est un raccourci vers une adresse en mémoire (e.g. l'adresse `X`). Quand je veux accéder au premier élément du tableau `patate2[0]`, j'accède à cette adresse `X`. Quand je veux accéder à son deuxième élément `patate2[1]`, j'accède à l'adresse `X + 1`, et ainsi de suite. De fait, connaitre l'adresse d'un élément nous permet d'avoir accès aux éléments liés.

En général, on déclare un pointeur avec une `*` :

```
int* patate3 = NULL;
```

1. On raconte que c'est la technique ultime de débuggage.

Remarque : si on veut déclarer un pointeur vide, il faut de préférence l'initialiser à `NULL`.

Maintenant qu'on a parlé d'adresse, cela peut-être intéressant de pouvoir récupérer l'adresse d'une variable. Pour ça, il suffit d'utiliser l'opérateur `&` :

```
patate3 = &patate;
```

Notre variable `patate2` est elle aussi un pointeur, et on peut donc accéder à ses éléments via leur adresse :

```
// pointeur = &valeur; => valeur = *pointeur;
printf("Adresse = %p, valeur stockée = %d\n", patate2, *patate2);
printf("Adresse = %p, valeur stockée = %d\n", patate2 + 1, *(patate2 + 1));
printf("Adresse = %p, valeur stockée = %d\n", patate2 + 2, *(patate2 + 2));
> Adresse = 0x7ffc801ae9a4, valeur stockée = 42
> Adresse = 0x7ffc801ae9a8, valeur stockée = 69
> Adresse = 0x7ffc801ae9ac, valeur stockée = 666
```

Pour pouvoir déclarer un tableau de taille non connue lors de la compilation, on peut utiliser la fonction `malloc`. Cette dernière a besoin de connaitre l'espace mémoire que l'on souhaite d'allouer. Pour ça, le plus simple est d'utiliser la fonction `sizeof` qui nous donne la taille d'un objet, qu'il ne faut pas oublier de multiplier par le nombre d'éléments de notre tableau.

```
patate3 = malloc(sizeof(int) * 15);
```

Remarque : `malloc` est une requête pour allouer de l'espace mémoire, mais nous ne sommes pas certains que la mémoire a suffisament d'espace disponible pour nous. Donc, idéalement, il faut vérifier que `malloc` a bien réussi à allouer notre mémoire :

```
int* patate3 = malloc(sizeof(int) * 100000000000000);
if(patate3 == NULL){
  printf("Error: too many potatoes for your belly\n");
  return -1;
}
```

Il ne nous reste plus qu'à apprendre comment utiliser ces pointeurs avec des fonctions. Pour ça, il suffit de changer le type de variable que la fonction va vouloir renvoyer :

```
int* generatePotatoes(int n){
  int* potatoes = malloc(sizeof(int) * n);
  return potatoes;
}
```

2.3 Structures

Une structure c'est comme un tableau sauf qu'on va vouloir donner des noms aux sous éléments et que l'on veut pouvoir avoir différents type de variable dans notre structure. C'est un peu comme les attributs d'une classe. On définit une structure comme ceci :

```
typedef struct {
  int nPotato;
  double weight;
  int *potatoes;
} potatoSack; // potatoSack va être l'alias de votre structure
```

Pour déclarer une nouvelle *instance* d'une structure, on va de nouveau utiliser la fonction `malloc` :

```
potatoSack *myPotatoes = malloc(sizeof(potatoSack));
```

Remarque : `myPotatoes` est bien un pointeur vu que malloc retourne une adresse.

Pour accéder à un élément spécifique de notre structure, on va utiliser l'opérateur `->` :

```
myPotatoes->n = 100; // Good potatoes
myPotatoes->weight = 3.27;
```

Remarque : on utilise l'opérateur `->` parce que `myPotatoes` est un pointeur vers une structure. Si notre variable est directement du type de la structrue, alors on utilise l'opérateur `.` (point).

```
potatoSack myPotatoesSValue = *myPotatoes;
myPotatoesValue.n = 70; // Lost some potatoes :-(
myPotatoesValue.weight = 4.56;
```

2.4 Libérer la mémoire

Nous avons donc vu dans les sections précédentes comment jouer avec les pointeurs et les structures, c'est à dire allouer de la mémoire pour faire ce qu'on veut. Cependant, il faut apprendre à partager nos ressources et, quand on a plus besoin de nos jouets, il faut les libérer avec la fonction `free` :

```
free(patate3);
free(myPotatoes); // myPotatoes->potatoes are lost...
```

Mais, nous devons faire attention car `C` fait exactement ce qu'on lui dit de faire. Donc, si on lui dit de supprimer notre `myPotatoes`, il ne va libérer que l'espace mémoire alloué aux variables contenues dans cette structure. Or, si on regarde la définition de cette structure, il y a un autre pointeur inclus dedans. `C` ne va pas supprimer la mémoire liée à ce pointeur si on ne lui dit pas (vu qu'il fait exactement ce qu'on lui dit mais rien de plus). Pour corriger ça, il faut donc :

```
free(patate3);
free(myPotatoes->potatoes); // myPotatoes->potatoes are now free :-)
free(myPotatoes); // L'ordre est important !
```

3 Résolution du problème

3.1 `inertiaGearSteelRho`

Cette fonction est simple et elle nécessite de faire une recherche sur votre navigateur préféré vu qu'il est votre ami. On trouve que la masse volumique de l'acier est comprise entre 7500 et 8100 $\mathrm{kg\,m^{-3}}$.

```
double inertiaGearSteelRho()
{
  return 7800;
}
```

3.2 `inertiaGearMeshRead`

Petit rappel de la signature de la fonction :

```
femMesh *inertiaGearMeshRead(const char *filename);
```

Ici on veut donc aller lire le fichier qui est décrit par `char* filename` et renvoyer une structure `femMesh`. Rappel de la structure de `femMesh` :

```
typedef struct {
  int *elem;
  double *X;
  double *Y;
  int nElem;
  int nNode;
  int nLocalNode;
} femMesh;
```

Par chance, cette fonction a été commencée par votre gentil professeur. On va donc commencer par la regarder pour comprendre ce qu'il a fait :

```
femMesh *theMesh = malloc(sizeof(femMesh));

int i,trash;

FILE* file = fopen(filename,"r");
if (file == NULL) Error("No mesh file !");
```

On commence par initialiser notre structure que l'on va devoir retourner, sous forme de pointeur, comme on l'a vu avant. On déclare les variables que l'on va utiliser et, ensuite, on lit le fichier. L'option `"r"` nous indique qu'on ouvre ce dernier en mode *read-only*. La ligne 6 est très importante vu qu'elle nous permet de nous assurer que notre programme a bien réussi à ouvrir notre fichier.

Remarque : c'est une très bonne pratique de s'assurer que ce genre d'action réussi pour pouvoir contrôler nos erreurs et ne pas avoir un crash pour des raisons inconnues.

Remarque : idéalement, il faudrait déclarer notre maillage après s'être assuré que l'on a pu ouvrir le fichier mais ça reste du chipotage.

Maintenant, passons à la suite :

```
ErrorScan(fscanf(file, "Number of nodes %d \n", &theMesh->nNode));
theMesh->X = malloc(sizeof(double)*theMesh->nNode);
theMesh->Y = malloc(sizeof(double)*theMesh->nNode);

for (i = 0; i < theMesh->nNode; ++i)
  ErrorScan(fscanf(file,"%d : %le %le \n",&trash,&theMesh->X[i],&theMesh->Y[i]));
```

La fonction `fscanf` est utilisée pour extraire un pattern dans notre ligne qu'on est entrain de lire. Ici, on veut vérifier que `"Number of node %d \n"` existe et directement écrire dans la mémoire, à l'adresse de `theMesh->nNode`, le nombre de noeuds. On utilise la fonction `ErrorScan` pour s'assurer que l'on a pas de souci lors de la lecture.

Remarque : la lecture se fait bien ligne par ligne avec `fscanf`.

Maintenant que l'on connait le nombre de noeuds, on peut créer nos tableaux en mémoire et, ensuite, les remplir élément par élément. On peut noter l'utilisation de la variable `trash` pour se débarasser du numéro du noeud. C'est à nous de jouer, mais on va pouvoir recopier et adapter ce que le prof a déjà fait :

```
ErrorScan(fscanf(file, "Number of triangles %d \n", &theMesh->nElem));
theMesh->elem = malloc(sizeof(int)*3*theMesh->nElem);
theMesh->nLocalNode = 3;
```

```
4
5 for (i = 0; i < theMesh->nElem; ++i)
6   ErrorScan(fscanf(file,"%d : %d %d %d \n",&trash,&theMesh->elem[3*i],&theMesh->
     elem[3*i+1],&theMesh->elem[3*i+2]));
7
8 fclose(file);
9 return theMesh;
```

On lit le nombre d'éléments et on alloue correctement notre tableau. On va aussi assigner à la variable `nLocalNode` le nombre noeuds par élément. Ici, on travaille avec des triangles, donc 3 noeuds par éléments. La lecture de chaque triangle se fait aussi avec une boucle for. Petite particularité, on lit 3 choses à insérer dans `elem` par ligne donc il faut bien multiplier `i` par 3 pour remplir correctement `elem`. Pour finir, on oublie pas de fermer notre fichier et de retourner le pointeur vers notre maillage.

Remarque : maintenant que l'on a un objet qui définit notre maillage, celui-ci devrait apparaitre avec `BOV` (ou autre).

3.3 `inertiaGearMeshFree`

Nous devons écrire la fonction pour libérer entièrement et correctement notre maillage. Nous avons vu comment on peut faire ça avant donc ce n'est pas compliqué :

```
1 void inertiaGearMeshFree(femMesh *theMesh)
2 {
3   free(theMesh->X);
4   free(theMesh->Y);
5   free(theMesh->elem);
6   free(theMesh);
7 }
```

3.4 `inertiaGearInertia`

Avec le devoir précédent et les fonctions que nous venons d'écrire, nous sommes armés pour **le point culminant** de ce devoir : l'intégration de l'inertie sur un maillage, passé en argument, via une méthode d'intégration, elle aussi passée en argument. La formule de l'inertie est la suivante :

$$I = \int\int\int \rho(x^2 + y^2)\,\mathrm{d}x\mathrm{d}y\mathrm{d}z \tag{1}$$

Notes :

- Les dimensions sont en mm.
- Il faut utiliser le `rho` qui est l'argument de notre fonction, et pas appeler la fonction écrite précédement.
- La roue dentée (et oui, une engrenage c'est la combinaison de 2 roudes dentées comme vu lors du cours de *Description et analyse des méchanismes* en bac 2) fait 10 cm d'épaisseur (z).

Le fait d'utiliser une règle d'intégration différente du devoir 1 ne devrait rien changer. Au lieu d'encoder nous même les points d'intégration et les poids, on va utiliser ceux qui sont fournis, rien de plus. Donc, avant de se lancer dans l'écriture, il faut décomposer ce que notre fonction doit faire :

1. Lire chaque triangle 1 par 1

2. Pour chaque triangle, récupérer l'information sur ses sommets
3. Utiliser ces sommets pour intégrer notre fonction
4. Finir notre intégration en sommant le tout

Pour l'étape 1, rien de particulier, une bête boucle `for` nous permet de les parcourir. Pour l'étape 2, il faut se rappeler que les numéros des sommets utilisés par le $i^{\text{ème}}$ triangle sont situés aux index `3*i`, `3*i+1` et `3*i+2` de notre variable `elem`.

Quand nous avons récupéré cette information, il suffit de parcourir les points d'intégrations comme fait dans le devoir 1. Petit rappel : il ne faut pas oublier de multiplier le tout par le jacobien.

Pour la dernière étape, il faut se rappeler que les unités du maillage sont en mm et que nous devons exprimer notre résultat en $\mathrm{kg\,m^2}$, ainsi nous devons diviser notre résultat par 10^6. Il ne faut pas non plus oublier que l'épaisseur de notre roue est de seulement 0.1 m ce qui fait que l'on doit multiplier notre résultat total par un facteur 10^{-7}.

```c
double interpolate(double u[3], double xsi, double eta) {
     return u[0]*xsi + u[1]*eta + u[2]*(1.0-xsi-eta);
}

double inertiaGearInertia(femMesh *theMesh, femIntegration *theRule, double rho) {
  double I = 0, Ipart = 0;
  double jac = 0;
  double xLoc[3];
  double yLoc[3];
  double xInteg = 0, yInteg = 0;

  for(int i = 0; i <  theMesh->nElem; i++) {
    Ipart = 0;
    // Fetch the X and Y of ith triangle
    for(int j = 0; j < 3; j++){
      xLoc[j] = theMesh->X[theMesh->elem[3*i+j]];
      yLoc[j] = theMesh->Y[theMesh->elem[3*i+j]];
    }
    jac = fabs((xLoc[0] - xLoc[1]) * (yLoc[0] - yLoc[2]) - (xLoc[0] - xLoc[2])*(
   yLoc[0] - yLoc[1]));

    for(int j = 0; j < theRule->n; j++){
      xInteg = interpolate(xLoc,theRule->xsi[j], theRule->eta[j]);
      yInteg = interpolate(yLoc,theRule->xsi[j], theRule->eta[j]);
      Ipart += (xInteg*xInteg + yInteg*yInteg) * theRule->weight[j];
    }
    I += Ipart * jac;
  }
  return I * rho * 1e-7;
}
```