Rapport sur le **T**ravail d'**É**tude et de **R**echerche effectué durant le second semestre

Sujet :

Optimisation topologiques de maillages structurés

Hatim ISMGH, Omar MAFTOUL
(hatim.ismgh@gmail.com) (maftoul.omar@gmail.com)

Master 1 Informatique, parcours Conception et Intelligence des Logiciels et des Systèmes

---

Encadré par :
Prof. Franck Ledoux

Table des matières

1 Introduction **1**

2 Problème **1**

3 Objectif **1**

4 Le problème binaire du Selective Padding **2**

5 Problème binaire simple **4**
5.1 Étude théorique 4
5.2 GLPK 4

6 Extension du problème binaire simple **9**
6.1 Étude théorique 9
6.2 Implémentation en C++ 11

7 Conclusion **14**

References **15**

1 Introduction

Le but de ce rapport est de décrire l'ensemble des démarches mises en oeuvre pour la réalisation du TER. Notre travail pour ce TER consiste à effectuer une contribution dans la bibliothèque **GMDS** (***G**eneric **M**esh **D**ata & **S**ervices*), écrite en C++ et qui a pour objectif d'offrir aux développeurs concernés des structures de données et des algorithmes avec lesquels on peut respectivement créer et manipuler des **maillages** [1] (**meshes**), ce travail s'inscrit au sein d'un projet au CEA pour fournir un outil de maillage/remaillage hexaédrique open-source dans les années qui viennent.

2 Problème

Les maillages en question sont de types *Polycube Hexaèdre*, et on les génère afin de permettre l'exécution des simulations numériques, notamment pour étudier des phénomènes physiques. Par contre, il est difficile de générer des maillages en 3D, car ceci nécessite beaucoup de temps et d'effort. Par ailleurs, la génération d'un maillage 3D n'est pas parfaite, car celle ci engendre un objet géométrique complexe composé de **cubes déformés** [2]. En ce moment, aucun algorithme automatique n'apporte une solution efficace pour ce problème protagoniste.

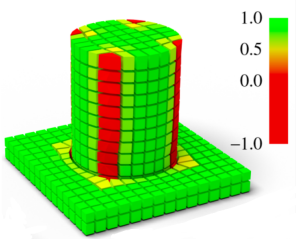


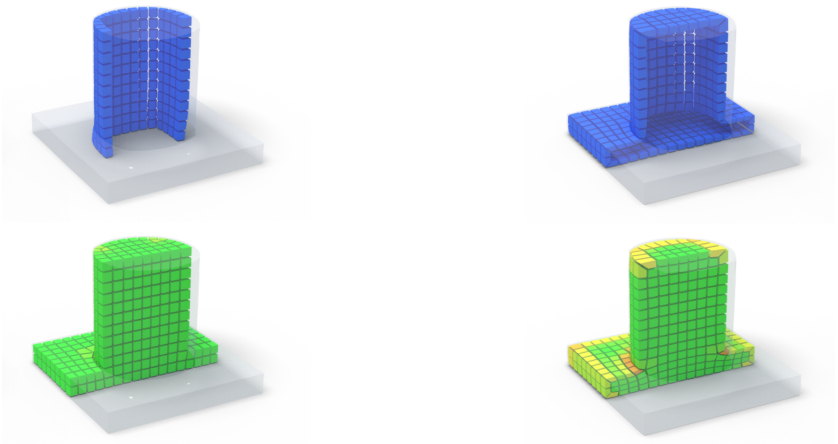
FIGURE 1 – Un maillage polycube 3D de mauvaise qualité (cubes déformés en rouge)

3 Objectif

La résolution du problème de déformation est notre objectif majeur, c'est ainsi que dans un premier temps,il existe une technique qui peut éventuellement résoudre le problème, on appelle cette technique **global padding** ou **pillowing**, cette dernière propose d'ajouter une extra couche d'hexaèdres sur l'ensemble du domaine où les cubes mal-formés appartiennent, en revanche, le **global padding** est une proposition moins optimale, car cette opération augmente la complexité (vu le très grand nombre d'hexaèdres qui seront ajoutés), ainsi que l'existence de certains de ces hexaèdres qui viennent de s'ajouter sont inutiles, et la qualité des éléments se trouvant près des bords du maillages peut diminuer.
Dans un article [*Selective padding for polycube based hexahedral meshing, Computer Graphics Forum, vol. 0, no. 0.*], les auteurs proposent une alternative au **global** padding : le **Selective Padding**, cette méthode est plus optimisée que celle qu'on a mentionné en premier, son principe consiste à sélectionner

1. Des collections de polygones et d'objets géométriques complexes.
2. Des cubes qui ont une mauvaise qualité.

le minimum des facettes qui doivent d'être *paddées* pour minimiser le nombre d'hexaèdres à ajouter. Selon le même article, la méthode du **Selective Padding** permet de faire un équilibre entre l'amélioration de la qualité du maillage, et l'augmentation de la complexité (cet équilibre qui ne pouvait pas se réaliser dans le **Global Padding**). La technique du **Selective Padding** est basée sur la résolution d'un ***problème binaire***. À cet effet, notre objectif est d'implémenter cette résolution avec le langage **C++** sous la bibliothèque **GMDS** (qui nous fournira elle même des algorithmes qui facilitera des éventuels calculs) et en utilisant **GLPK (GNU Linear Programming Kit) [2]**, une bibliothèque écrite en **C** qui permet de résoudre des problèmes de type programme linéaires (Linear Programming).



(a) Selective Padding (b) Global Padding

FIGURE 2 – Comparaison entre le **Selective Padding** et le **Global Padding**, l'application de la méthode en bleu et le résultat en vert

4 Le problème binaire du Selective Padding

Avant d'aborder l'approche utilisé par le Selective Padding, il est préférable d'apporter une vision sur qu'est ce qu'un maillage. La combinaison de plusieurs éléments forment un mailalge de type polycube hexaèdre, le *padding* nécessite ces éléments afin de reconnaître le maillage. Les parties de ce dernier sont : $M = V, E, F, H$, soit respectivement l'ensemble des noeuds V, l'ensemble des arêtes E, l'ensemble des facettes F et l'ensemble des régions (ou hexaèdres) H.

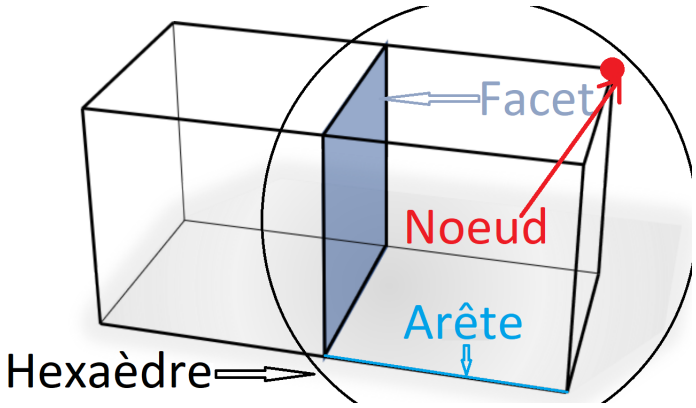


FIGURE 3 – Élements d'un simple maillage qui se constitue de deux hexaèdre

Comme on l'a mentionné précédemment, le but du Selective Padding est de minimiser le nombre d'hexaèdres ajoutés (à l'encontre du Global Padding), ceci s'effectuera en faisant retourner les couches d'hexaèdres nouvellement ajoutées autour d'un noeud ou une arête (***edge turn or vertex turn***). À cet effet, pour mettre en oeuvre cette démarche, le Selective Padding se repose sur la résolution du problème de déformation qui sera transformé en un problème d'optimisation linéaire, ce dernier imposera des contraintes afin de préserver la cohérence des éléments constituant la structure topologique du maillage. La résolution du problème binaire utilisé par la méthode du Selective Padding se conclue par avoir une solution qui permet d'obtenir des facettes permettant à leurs tour de fournir une optimisation topologique (*padding*) consistante et correcte, ces facettes inclues forcément des facettes issue d'hexaèdres mal-formés qu'on appelle **Hard-constrained Facets**(HF), ces HF sont notre objectif, ce sont les facettes qu'on doit optimiser en effectuant ce qu'on appelle ***Facet extrusion***[3], cette démarche sera élaboré sous forme de fonctions avec des contraintes et des variables binaires (c'est ainsi qu'il est considéré comme problème binaire) : une variable pour chaque facette, chaque arête (pour calculer le ***edge turn***) et chaque noeud (pour calculer le ***vertex turn***). Notons que l'application d'un ***edge turn*** ou un ***vertex turn*** résulte des anomalies[4], on essayera de diminuer par la suite le nombre d'anomalies.

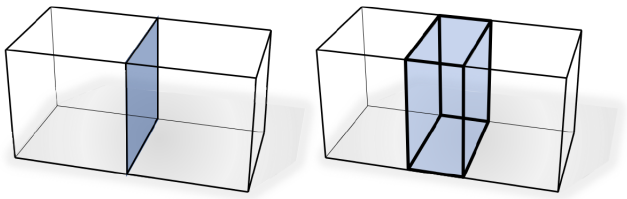


FIGURE 4 – L'extrusion d'une seule facette (à gauche) résultant un seul hexaèdre (à droite)

3. Pour chaque facette on crée un hexaèdre : l'extrusion de n facettes résulte la création de n hexaèdres
4. Un **edge turn** résulte 2 arêtes déformés, et un **vertex turn** résulte 7 arêtes déformés

5 Problème binaire simple

5.1 Étude théorique

Dans cette section, nous allons aborder l'étude du problème binaire simple qui est proposé par l'article [1]. On va associer pour chaque facette $f \in F$ une variable binaire x_f, la valeur de cette variable déterminera si la facette sera *paddée* ou non. Notre premier objectif est de *padder* le minimum possible des facettes. Donc on cherche à minimiser la fonction $E_{padding}$ qui sera définie par :

$$E_{padding} = \|H\|^{\frac{-2}{3}} \sum_{f \in F \setminus HF} x_f \tag{1}$$

$\|H\| = \text{card}(H)$, l'ensemble HF désigne les facettes qui doivent être *paddées* qu'on a extrait des cubes déformés et donc on doit les avoir dans la solution finale.

$$x_f = 1 \qquad \forall f \in HF \tag{2}$$

Ainsi la deuxième contrainte sera ajoutée pour préserver la cohérence [5] du *polycube*.

$$\sum_{f \in F(e)} x_f = 2 \cdot k_e \qquad \forall e \in E \tag{3}$$

$F(e)$ désigne l'ensemble des facettes incidentes a l'arête e_j.
k_e est une variable entier associée a chaque arête.

5.2 GLPK

L'usage de la bibliothèque `GLPK` sera nécessaire pour la résolution du problème linéaire, c'est ainsi qu'on va l'utiliser tout au long de notre implémentation. Nous allons donc introduire consciencieusement et brièvement le principe du `GLPK` à travers cette section. `GLPK` a une formulation standard qui doit être respectée, comme suit : supposons qu'on veut *minimiser* ou *maximiser* l'équation $z = 10x_1 + 6x_2 + 4x_3$ sous les contraintes suivantes :

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 100 \\ 10x_1 + 4x_2 + 5x_3 &\leq 600 \\ 2x_1 + 2x_2 + 6x_3 &\leq 300 \end{aligned}$$

pour le faire, `GLPK` suggère de poser les contraintes dans un format standard que les fonctions `glpk` peuvent accepter. Le format standard des contraintes de z est alors :

$$\begin{aligned} p &= x_1 + x_2 + x_3 \\ q &= 10x_1 + 4x_2 + 5x_3 \\ r &= 2x_1 + 2x_2 + 6x_3 \end{aligned}$$

5. pour comprendre la cohérence d'un *polycube*, on suppose qu'une arête est incidente a quatre facettes, si 3 facettes sont *paddées*, on va avoir un problème d'inconsistance et la contrainte (3) permet de l'éviter.

Et on se retrouve désormais avec les inégalités suivantes :

$$\begin{aligned} -\infty < p \leq 100 \qquad & 0 \leq x_1 \leq +\infty \\ -\infty < q \leq 600 \qquad & 0 \leq x_2 \leq +\infty \\ -\infty < r \leq 300 \qquad & 0 \leq x_3 \leq +\infty \end{aligned}$$

Cette standardisation permet de représenter le programme linéaire sous la forme d'une matrice. Les variables auxiliaires p, q et r correspondent à des lignes de la matrice. Aussi, x_1, x_2 et x_3 sont des variables structurelles (*structural variables*), qui correspondent aux colonnes de la matrice. La fonction z peut être généralisée de cette manière :

$$z = c_1 x_{m+1} + c_2 x_{m+2} + \cdots + c_n x_{m+n} + c_0 \tag{4}$$

où c_0 est une constante, aussi, la généralisation d'une contrainte peut prendre la forme suivante : $x_m = a_{m1} x_{m+1} + a_{m2} x_{m+2} + \cdots + a_{mn} x_{m+n}$ où m correspond au numéro de la ligne de la matrice, et n représente le numéro de la colonne.
Maintenant dans cette section, on va mettre en pratique ce qui a été introduit dans la section précédente sur le problème binaire simple. Il est primordial de présenter les différents outils utilisés à cet effet : **GMDS** et **GLPK**. On crée une fonction **SelectivePadding(Mesh)** où tout se déroulera, elle prend en paramètre un maillage (extrait via un fichier en entrée de la fonction principale du programme **main**) sous l'extension *.vtk* qu'on pourra visualiser avec un programme dédié **[3]**, ce type de fichier contient plusieurs informations qui décrivent le maillage, ainsi qu'un paramètre qui définira le degré de complexité qu'on souhaite avoir (qu'on verra au cours de l'extension du problème binaire simple dans les prochaines sections). Notons que chaque maillage sous **GMDS** est composé de cellules, instances de la classe `Cell` qui sont liés topologiquement entre eux. Une cellule peut être un sommet (dimension 0), instance de la classe `Node`, une arête (dimension 1),instance de la classe `Edge`, une facette (dimension 2), instance de la classe `Face`, ou un hexaèdre (dimension 3), instance de la classe `Region`. Ces éléments sont représentés via les classes fournies par **GMDS** comme suit :

```
class Region, class Node, class Edge, class Face
```

Après extraction du maillage on va résoudre le problème d'optimisation linaire pour déterminer les facettes qui doivent être *paddées* en utilisant **GLPK** : on commence par l'initialisation du problème linéaire qu'on souhaite résoudre, dans notre cas, on veut *padder* le moins possible de facettes, donc il s'agit d'une minimisation.

```
1 glp_prob *lp;
2 lp = glp_create_prob();
3 glp_set_prob_name(lp,"Simple Binary Problem");
4 glp_set_obj_dir(lp, GLP_MIN);
```

Notons que toute fonction qui commence par `glp` est prédéfinie dans `GLPK` sous `glpk.h` qu'on doit inclure. Après la création du problème, on doit alimenter ce dernier avec plus de détails afin qu'il puisse

nous donner une solution.
On va commencer à standardiser $E_{Padding}$ ((1) 4). Selon les deux contraintes (2) et (3), on aura un nombre de variables structurelles qui sera égale au nombre de facettes, plus le nombre d'arêtes, car selon (2), on attribuera pour chaque facette mal formée la valeur 1, cette dernière appartient à HF, et HF fait partie de tout l'ensemble de F, et selon (3) l'ensemble des variables binaires x_f lié à chaque facette i sera égale $2k_e$, où cette dernière sera liée à une arête e qui appartient à E. Donc le nombre de colonnes de la matrice sera le nombre d'arêtes plus le nombre de facettes. Pour les variables auxiliaires, le nombre de lignes de la matrice selon (5) sera le nombre d'arêtes du maillage. Le code ci dessous exprime ce qu'on vient d'expliquer.

```
 //nb rows
glp_add_rows(lp, m_mesh->getNbEdges());
 //nb columns
glp_add_cols(lp, m_mesh->getNbEdges()+m_mesh->getNbFaces());
```

Notons que `m_mesh` est une variable de type `Mesh`, une classe fournie par `GMDS`, elle permet d'acquérir le maillage introduit via le fichier *.vtk* qu'on a mentionné précédemment. Dans le code ci-dessus, on voit que la variable est un pointeur ayant accès a plusieurs fonctions, notamment `getNbEdges();getNbFaces()` qui respectivement fournissent le nombre total des arêtes et des facettes dans le maillage.

Maintenant qu'on est arrivé à savoir le nombre de lignes et de colonnes pour la matrice du problème $E_{Padding}$, on doit affecter à chaque arête et à chaque facette un entier, ceci nous permettra par la suite de construire les éléments non nulle de la matrice de contrainte qu'on verra après, et à attribuer pour chaque ligne et pour chaque colonne un nom et des bornes. Pour se faire, on initialise deux variables qui sont du type `map<TCellID, int>`, ce dernier doit faire correspondre un entier à un élément (arête, facette) de la manière suivante : `variable[element_id] = integer`, certes, la variable a une **clé (son identifient unique)** et **une valeur (un entier)**, si par exemple on veut accéder à la variable affecté à une arête dont l'id est 4, `variable[4]` sera l'instruction dont on a besoin.
On peut maintenant procéder à rajouter les bornes pour chaque ligne. Selon (5), pour notre cas sur $E_{Padding}$, les contraintes pour les lignes sont des variables fixes, et donc on va mettre en place une borne minimale et maximale fixées à 0, car $C_{k_e} = 0$. Pour mettre ceci en pratique, `GLPK` met à notre disposition des routines (ou fonctions) qui prennent en argument respectivement le problème en question, la position ou l'indice de la ligne de la matrice (pour chaque arête), le type de la variable correspondante à la ligne, et selon le type on met les bornes.

```
for (auto ei_id:m_mesh->edges()) {
        std::string aux_name = std::string("E")+std::to_string(ei_id);
        //name
        glp_set_row_name(lp, E2Xi[ei_id] , aux_name.c_str());
        //Binary variable, GLP_FX means fixed variable
        glp_set_row_bnds(lp, E2Xi[ei_id] , GLP_FX,0,0);
    }
```

La deuxième contrainte (3) doit être transformée pour respecter le format standard des contraintes définies par `glpk` :

$$C_{k_e} = \sum_{f \in F(e)} X_f - 2k_e \qquad \forall e \in E \qquad C_{k_e} = 0 \tag{5}$$

Pour les colonnes, on suit les mêmes procédures que pour les lignes en utilisant les routines `glpk` dédiées. Selon (1), on rajoute une constante $H^{\frac{-2}{3}}$ qu'on multipliera avec la somme des x_f. La routine `glp_set_obj_coef` permet de rajouter un coefficient pour le multiplier à chaque colonne, elle prend en argument respectivement le problème linéaire en question, la variable qu'on veut multiplier avec le coefficient (il s'agit d'une colonne), et le coefficient. En revanche, aucune arête ne sera impactée par $H^{\frac{-2}{3}}$, donc on fixera 0 comme coefficient pour les arêtes (ils ne font pas partie de $E_{padding}$), par contre ce sont les facettes qui seront concernés par ce coefficient. Notons que `glp_set_col_kind` est une fonction qui spécifie le type de la colonne (variable continue, entier, ou binaire), en ce qui nous concerne, toutes le colonnes sont des variables binaires, on utilisera à cet effet le type `GLP_BV` prédéfinit par `glpk`. Étant donnée que les facettes de mauvaise qualité auront la valeur 1, on utilisera des bornes fixés à 1 en utilisant `GLP_FX` :

```
    for (auto ei_id:m_mesh->edges()) {
          //Structural Variables associated to edges
         std::string name = "e"+std::to_string(ei_id);
         glp_set_col_name(lp,E2Xi[ei_id] , name.c_str());
         glp_set_col_kind(lp, E2Xi[ei_id] , GLP_BV);          //Binary variable
         glp_set_obj_coef(lp, E2Xi[ei_id] , 0); //no impact on Energy value
     }
     for (auto fi_id:m_mesh->faces()) {
     //Structural Variables associated to facets
     if (m_hard_constraint->value(fi_id) == 1) {
         //codage de la première contrainte (2)
         std::string name = "hf"+std::to_string(fi_id);
         glp_set_col_name(lp, F2Xi[fi_id] , name.c_str());
         glp_set_col_bnds(lp, F2Xi[fi_id] , GLP_FX, 1.0, 1.0);
         glp_set_obj_coef(lp, F2Xi[fi_id] , coeff);
     }
     else {
         std::string name = "f"+std::to_string(fi_id);
         glp_set_col_name(lp, F2Xi[fi_id] , name.c_str());
         glp_set_col_kind(lp, F2Xi[fi_id] , GLP_BV);
         glp_set_obj_coef(lp, F2Xi[fi_id] , coeff);
     }
    }
```

A posteriori, on initialise la matrice de contraintes en créant trois tableaux d'entiers **`ia, ja, ar`**, `ia` sert à stocker les indices de chaque élément des lignes de la matrice, `ja` a la même fonction pour les colonnes de la matrice, et `ar` stocke la valeur numérique de chaque élément ij de la matrice . Selon (5), chaque élément constituant une arête aura la valeur -2 et chaque élément constituant une facette aura la valeur 1 (stockée dans ar) :

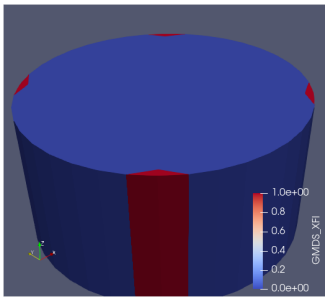
```
    int    ia[1 + 4*m_mesh->getNbEdges() + 4*m_mesh->getNbFaces()];
    int    ja[1 + 4*m_mesh->getNbEdges() + 4*m_mesh->getNbFaces()];
    double ar[1 + 4*m_mesh->getNbEdges() + 4*m_mesh->getNbFaces()];
    int i=1; //compteur qui enregistre le nombre des coefficients  de la matrice
    // Constraint matrix for simple binary problem
```

```
    for (auto ei_id:m_mesh->edges()) {
        Edge ei = m_mesh->get<Edge>(ei_id);
        ia[i] = E2Xi[ei_id];
        ja[i] = E2Xi[ei_id];
        ar[i]= -2;
        i++;
        // For each facet incident to the current edge
        std::vector<TCellID > adj_face_ids = ei.getIDs<Face>();
        for(auto fi_id:adj_face_ids){
            ia[i] =E2Xi[ei_id];
            ja[i] =F2Xi[fi_id];
            ar[i]= 1;
            i++;
        }
    }
```

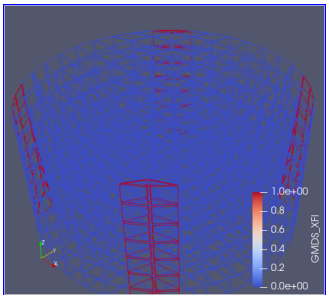
Après, on récupère les informations depuis les trois tableaux **ia, ja, ar** afin de charger et résoudre le problème en utilisant la routine `glp_load_matrix` et on extrait la valeur de la variable binaire associée à chaque facette.

```
int e=0;
    for(auto ei_id:m_mesh->edges()){
        double val = glp_mip_col_val(lp, E2Xi[ei_id] );;
        if(val!=0)
            e++;
    }
```

La phase d'implémentation du problème binaire simple est achevée, on teste cette dernière sur un cylindre (un fichier $*.vtk$ qu'on fournit en argument). Dès lors, on obtient les résultats qu'on a souhaiter avoir (le minimum de facettes + la consistance du cylindre), ce qui signifie qu'on a bien pu acquérir une solution optimale suite à l'implémentation du code en C++ : La figure (5) illustre cet exemple. En (a) sont visibles en rouge, les faces à padder impérativement (celles qui composent l'ensemble HF). Ce ne sont que des faces au bord qui ne forment pas une surface fermée. En (b), on visualise en transparence toutes les faces du maillage. En rouge, les faces telles que $Xf{=}1$ forment bien des surfaces. En l'occurence, la résolution du problème de minimisation a induit de sélectionner le minimum de faces telles que l'on forme des surfaces fermées pour chaque composante connexe de faces rouges passées en paramètre.



(a) Le volume qui doit être *paddé*



(b) Un filtrage qu'on a appliqué sous **`ParaView`** pour afficher les facettes intérieures et extérieures du cylindre sélectionnées par l'algorithme

FIGURE 5 – La couleur rouge représente les facettes qui ont la valeur 1 après la résolution du problème linéaire et qui doivent être *paddés*

6 Extension du problème binaire simple

6.1 Étude théorique

La méthode proposée à la section 5 n'est pas toujours optimale pour tous les maillages, car il existe des cas où on peut l'améliorer en insérant moins d'hexaèdres. Considérons l'exemple de la figure (6) ci dessous :

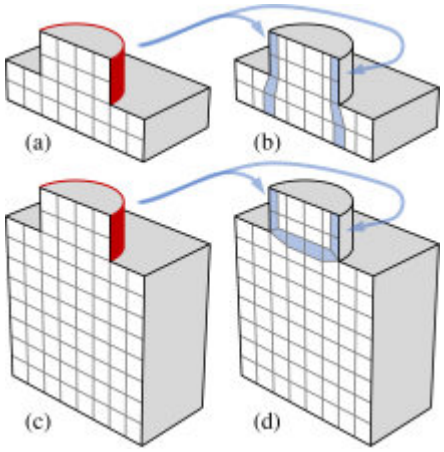


FIGURE 6 – (b) est la solution qu'on a trouvé en utilisant la méthode du problème binaire simple (par le biais d'extrusion des facettes) (d) est une autre solution plus efficace qu'on souhaite avoir.

Pour avoir la solution (d) il faudra modifier le problème binaire simple afin qu'on puisse minimiser aussi le nombre d'hexaèdres à rajouter.
On peut constater que la minimisation du nombre d'hexaèdres est la même que la minimisation des ***edge-turn*** et des ***vertex-turn***, donc le changement qu'on apportera au problème en question s'effectuera via l'ajout de nouvelles variables et contraintes qui vont nous permettre d'atteindre cet objectif.

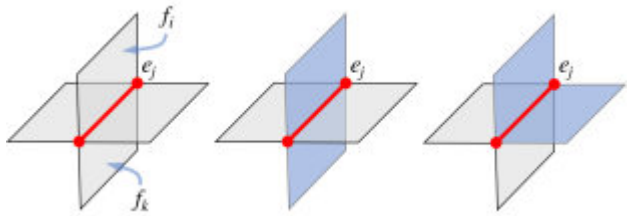


FIGURE 7 – La couleur bleue foncée représente les facettes qui doivent être *paddés*. Au milieu, e_j n'est pas un ***edge-turn***, par contre à droite, e_j est un ***edge-turn***

On ajoute pour chaque arête $e \in E$ une variable binaire t_e qui déterminera si elle est un ***edge-turn*** ou non, ainsi que pour les noeuds on ajoutera la variable t_v qui déterminera si elle est un ***vertex-turn*** ou pas, donc notre but est de minimiser aussi la somme de ses variables.

$$E_{complexity} = \|H\|^{\frac{-1}{3}} \sum_{e \in E} t_e + \sum_{v \in V} t_v \tag{6}$$

(6) est une équation modifiée de celle qu'on retrouve dans [1], cette légère modification concerne les noeuds et les arêtes, ces derniers appartiennent respectivement à E^* et V^* :

$$E_{complexity} = \|H\|^{\frac{-1}{3}} \sum_{e \in E^*/E1H} t_e + \sum_{v \in V^*/V1H} t_v \tag{7}$$

V^* et E^* sont respectivement les ensembles de noeuds et arêtes qui sont incidents à deux facettes orthogonales appartenant à HF. Cette proposition a pour but de diminuer la complexité du calcul, mais elle n'as pas d'impact sur la solution. Nous allons mettre en pratique une implémentation adéquate avec les deux versions d'équations.
$E1H$ et $V1H$ sont l'ensemble des noeuds et des arêtes incidents à un seul hexaèdre.
Pour savoir si e est un ***edge-turn***, il suffit de soustraire les variables binaires associées à deux facettes incidents à e et qui ont la même orientation, par exemple dans la figure 7 on a :
$|x_{f_i} - x_{f_k}| = 0$ et $\overrightarrow{\mathrm{f}_i} = \overrightarrow{\mathrm{f}_k} \iff$ e_j n'est pas un ***edge-turn*** (au milieu)
$|x_{f_i} - x_{f_k}| = 1$ et $\overrightarrow{\mathrm{f}_i} = \overrightarrow{\mathrm{f}_k} \iff$ e_j est un ***edge-turn*** (à droite)
On va raisonner avec la même manière afin de détecter les ***vertex-turn***, donc les contraintes de l'équation (7) sont définies par :

$$t_e = |x_{f_i} - x_{f_k}| \quad \forall e \in E \quad \overrightarrow{f_i} = \overrightarrow{f_k} \quad et \quad f_i, f_k \in F(e) \tag{8}$$

$$t_v = |t_{e_i} - t_{e_k}| \quad \forall v \in V \quad \overrightarrow{e_i} = \overrightarrow{e_k} \quad et \quad e_i, e_k \in E(v) \tag{9}$$

F(e) et E(v) sont l'ensemble des facettes incidents à e et l'ensemble des arêtes incidentes à v (respectivement). Finalement on optimisera une combinaison linéaire de $E_{padding}$ et $E_{complexity}$ suite aux contraintes (2) (3) (8) (9).

$$minimiser(E = E_{padding} + \lambda \cdot E_{complexity}) \tag{10}$$

λ est le degré de complexité qu'on souhaite avoir, plus λ est grande plus on favorisera la minimisation des termes de (6) par rapport à $E_Padding$ (1).

6.2 Implémentation en C++

En suivant toujours les normes utilisées par **GLPK**, on procède tout d'abord par indiquer le nombre de lignes et de colonnes pour notre matrice de contraintes, ceci étant fait après que l'on initialise des dictionnaires de type `map<TCellID, int>` pour chaque variable correspondante à un noeud, arête et facette. D'après (9) et (8), le nombre de colonnes dans la matrice de contrainte sera égale à la somme du double du nombre d'arêtes et du double du nombre de noeuds, puisque les variables t_v et t_e sont des valeurs absolues et peuvent par conséquent avoir deux valeurs possibles. De plus, le nombre de lignes sera égale à la somme du nombre des noeuds et le nombre des arêtes. Puisque l'implémentation en cours et une extension du problème binaire simple du début, on va se contenter de continuer à travailler avec la même instance du problème, et on modifiera donc les routines `glp_add_rows` et `glp_add_cols` :

```
1  glp_add_rows(lp, m_mesh->getNbEdges()*2+m_mesh->getNbNodes());
2  glp_add_cols(lp, m_mesh->getNbEdges()*3+m_mesh->getNbFaces()+m_mesh->getNbNodes()*2
```

La prochaine étape consiste à créer des nouvelles variables structurelles, nous allons de plus rajouter le coefficient λ. La fonction `SelectivePadding(Mesh,Lambda)` qu'on a mentionné précédemment (section 5.3) verra un second paramètre `Lambda`, il sera prédéfinit manuellement dans le programme. Pendant le test, `Lambda` est initiée à 1. Bien évidemment, d'après (7), le coefficient des noeuds sera λ, tandis que les arêtes auront deux coefficients (λ et $H^{-\frac{1}{3}}$). Les variables t_e et t_v seront de type `GLP_BV` car ce sont des variables binaires.

```
1  for (auto ei_id:m_mesh->edges()) {//pour chaque arête on ajoute une variable Tej
2        std::string name = "e2"+std::to_string(ei_id);
3        glp_set_col_name(lp,T2Xi[ei_id] , name.c_str());
4        glp_set_col_kind(lp,T2Xi[ei_id] , GLP_BV);
5        glp_set_obj_coef(lp,T2Xi[ei_id] , coeff2*lamda);
6
7     }
8     for (auto ni_id:m_mesh->nodes()) {//pour chaque noeud on ajoute une variable Tvl
9        std::string name = "v"+std::to_string(ni_id);
10       glp_set_col_name(lp,V2Xi[ni_id] , name.c_str());
11       glp_set_col_kind(lp,V2Xi[ni_id] , GLP_BV);
12       glp_set_obj_coef(lp,V2Xi[ni_id] , lamda);
13    }
```

Les deux contraintes (8) (9) sont des contraintes non linéaires à cause de la valeur absolue, pour les rendre linéaire on a utilisé une méthode qui se porte en fonction de notre situation :

$$C_{t_e} = x_{f_i} - x_{f_k} - t_e + 2 \cdot \varepsilon_e \quad \forall e \in E \quad C_{t_e} = 0 \qquad \varepsilon_e \in \{0,1\} \quad \overrightarrow{f_i} = \overrightarrow{f_k} \quad et \quad f_i, f_k \in F(e) \qquad (11)$$

La contrainte (11) est équivalente à la contrainte (5) car si $x_{f_i} - x_{f_k} = -1$ ou 1, la seule solution pour satisfaire la contrainte c'est quand $t_e = 1$ et $\varepsilon_e = 1$, si $x_{f_i} - x_{f_k} = 0$, alors la seule solution pour satisfaire la contrainte c'est quand $t_e = 0$ et $\varepsilon_e = 0$, donc on peut détecter les ***edge-turn*** en utilisant la contrainte linéaire (11) au lieu de (8).
Le même raisonnement sera mis en oeuvre pour les ***vertex-turn***, la contrainte linaire équivalente à (9) est donc :

$$C_{t_v} = t_{e_i} - t_{e_k} - t_v + 2 \cdot \varepsilon_v \quad \forall v \in V \qquad C_{t_v} = 0 \quad \varepsilon_v \in \{0,1\} \quad \overrightarrow{e_i} = \overrightarrow{e_k} \quad et \quad e_i, e_k \in E(v) \qquad (12)$$

On va rajouter les variables structurelles ε_e qu'on vient d'introduire pour chaque arête $e \in E$ et pour chaque noeud $v \in V$.

```
  for (int j = 0; j < m_mesh->getNbEdges(); ++j) {
        //variable binaire qu'on ajoute pour linéariser la contrainte de la
        //valeur absolue pour les edge-turn
        std::string name = "epsilon"+std::to_string(j);
        glp_set_col_name(lp,i_x, name.c_str());
        glp_set_col_kind(lp,i_x, GLP_BV);          //Binary variable
        glp_set_obj_coef(lp,i_x, 0);
        i_x++;
        //i_x est un compteur qui contient le nombre de toutes les
        //variables auxiliaires qu'on a crée
    }
     for (int j = 0; j <m_mesh->getNbNodes(); ++j) {
        //variable binaire qu'on ajoute pour linéariser la contrainte de la
        //valeur absolue pour les vertex-turn
        std::string name = "epsilon"+std::to_string(j);
        glp_set_col_name(lp,i_x, name.c_str());
        glp_set_col_kind(lp,i_x, GLP_BV);          //Binary variable
        glp_set_obj_coef(lp,i_x, 0);
        i_x++;
    }
```

Rappelons qu'on applique la résolution selon (6) sur l'ensemble des noeuds et des arêtes. À cet effet, nous allons aussi introduire l'extraction des arêtes $\in E^*$ et des noeuds $\in V^*$. Par conséquent, pour avoir E^*, il faut qu'on itérer sur l'ensemble des arêtes et récupérer toute arête adjacente à plus d'une facette, ensuite, il suffit de vérifier si ces facettes adjacentes sont orthogonales (le produit scalaire des vecteurs normaux de chacune des deux facettes dois être nulle). La même procédure peut être suivie pour extraire V^*.

```
    for(auto edge_id : m_mesh->edges()){
        Edge edge = m_mesh->get<Edge>(edge_id);
        std::vector<TCellID> f_e = edge.getIDs<Face>();
        // edge may be incident to 2 or more facets
        if(f_e.size()>1)
            for(int r=0;r< f_e.size();r++)
                for(int j=r+1; j<f_e.size();j++)
                {
                    Face facet_one = m_mesh->get<Face>(f_e[r]);
                    Face facet_two = m_mesh->get<Face>(f_e[j]);
                    // calculate the normal vector of each facet, then
                    //calculate the dot product to see if they are orthogonal
                    if(facet_one.normal().dot(facet_two.normal()) == 0)
                        // if the two facets are orthogonal,
                        //then check if they are hard facets or not
                        if( (std::find(hard_facets.begin(), hard_facets.end(),
                        f_e[r]) != hard_facets.end()) &&
                            (std::find(hard_facets.begin(), hard_facets.end(),
                            f_e[j]) != hard_facets.end()))
                            e_star.push_back(edge_id);
                }
    }
```

Ensuite on va construire la matrice des contraintes qui décrit (11) en utilisant les tableaux `ia, ja et ar`. Afin de vérifier si deux facettes ont la même direction, il faut vérifier que le produit vectoriel de leurs vecteurs normaux est nulle, ceci peut être réalisé à l'aide des fonctions prédéfinies dans GMDS qui permettent d'effectuer des opérations vectorielle tels que `cross()`, `normal()` ou `dot()`.

```
//deuxiéme contrainte
  for (auto ei_id:m_mesh->edges()) {
      Edge ei = m_mesh->get<Edge>(ei_id);
      ia[i] = T2Xi[ei_id];
      ja[i] = T2Xi[ei_id];
      ar[i]= -1;
      i++;
      std::vector<TCellID > adj_face_ids = ei.getIDs<Face>();
      //récupérer la liste des facettes incidents à l'arrêt ei_id
      for(auto f1_id:adj_face_ids){
          //
          bool  a= false;
          for(auto f2_id:adj_face_ids){
              Face facet_one = m_mesh->get<Face>(f1_id);
              Face facet_two = m_mesh->get<Face>(f2_id);
              if(facet_one.normal().cross(facet_two.normal()).isZero() && f1_id != f2_id)
              {//tester si deux facettes ont la même orientation
                  ia[i] = T2Xi[ei_id];
                  ja[i] = F2Xi[f1_id];
                  ar[i]= -1;
                  i++;
                  ia[i] = T2Xi[ei_id];
                  ja[i] = F2Xi[f2_id];
                  ar[i]= 1;
                  i++;
                  ia[i] = T2Xi[ei_id];
                  ja[i] = i_x;
                  ar[i]= 2;
                  i++;
                  i_x--;
                  a= true;
              }
          }
          if (a== true) break;
      }
  }
```

Enfin on achève l'implémentation de notre solution par coder la contrainte (12).

```
//troixiéme contrainte
 for (auto ni_id:m_mesh->nodes()) {
      Node ni = m_mesh->get<Node >(ni_id);
      ia[i] = V2Xi[ni_id];
      ja[i] = V2Xi[ni_id];
      ar[i]= -1;
      i++;
      std::vector<TCellID > adj_edge_ids = ni.getIDs<Edge>();
      //extraire les arrêts incidents au noeud ni_id
      for(auto e1_id:adj_edge_ids ){
          bool  a= false;
          for(auto e2_id:adj_edge_ids){
```

```
                Edge edge_one = m_mesh->get<Edge>(e1_id);
                Edge edge_two = m_mesh->get<Edge>(e2_id);
                if(edge_one.segment().getUnitVector().cross(edge_two.segment().getUnitVector())
                {//tester si deux arrêts on la même orientation
                    ia[i] = V2Xi[ni_id];
                    ja[i] = T2Xi[e1_id];
                    ar[i]= -1;
                    i++;
                    ia[i] = V2Xi[ni_id];
                    ja[i] = T2Xi[e2_id];
                    ar[i]= 1;
                    i++;
                    ia[i] = V2Xi[ni_id];
                    ja[i] = i_x;
                    ar[i]= 2;
                    i++;
                    i_x--;
                    a= true;
                }
            }
            if (a== true) break;
        }
    }
```

L'exécution de la deuxième implémentation sur le même maillage (a) a pris un temps considérablement remarquable ($\simeq 3min$, peut varier selon les spécifités technique de chaque machine) afin de trouver une solution optimale du problème binaire qu'on a modifié, ce qui représente une différence par rapport au temps d'exécution du problème binaire simple, ceci repose sur le nombre des variables structurelles et auxiliaires utilisées sur chacun des problèmes (certes, ce nombre a augmenté par $2 \cdot \|E\| + 2 \cdot \|V\|$ sur les lignes et par $\|E\| + \|V\|$ sur les colonnes suite à la transition depuis le problème binaire simple vers son extension), en sachant aussi que pour $E_{Complexity}$, nous avons utilisé l'ensemble des noeuds et des arêtes au lieu de travailler sur $E^*/V2H$ et $V^*/V1H$ comme on l'a déjà mentionné, par ailleurs, ce choix qui nous a pousser de passer depuis (7) proposé par [1] vers (6) est suite au cylindre qu'on a mis sous test, ce dernier avait les ensembles $E^*/V2H$ et $V^*/V1H$ vides. Cela avait un impact sur le temps d'exécution mais pas sur le résultat.

7 Conclusion

En récapitulatif, notre travaille s'est reposé sur l'analyse de la méthode d'optimisation topologique proposée par [1] et d'implémenter une partie de cette proposition sur **GMDS** en C++, en outre, cette contribution mineure qui fait partie d'un projet d'envergure au CEA était un facteur motivant et nous a donné envie de s'approfondir sur le thème des simulations numérique et les maillages en futur. Le défi majeur de ce projet était pour nous la manière dont on devait traiter l'article [1] afin de convertir ses propositions en une implémentation en code, heureusement on a pu l'effectuer avec l'immense aide de notre encadrant. Les équations ainsi que les contraintes proposée dans les équations de l'article étaient parmi les difficultées rencontrées tout au long du projet, on était donc parfois obligés de reformuler certaines contraintes (tels que travailler dans des ensembles différents de celles proposées par [1] dans $E_{Padding}$ et $E_{Complexity}$) pour qu'on puisse les implémenter d'une manière adéquate avec notre maillage.

On pense que notre effort vis à vis de ces reformulations peut être une contribution pour l'article même. L'incompatibilité avec un maillage donnée peut être un inconvenant de cet algorithme selon l'expérience vécue dans ce projet, ainsi, le temps d'exécution qui augmente en fonction de la complexité du maillage peut être considéré comme inconvénient, donc on suggère de reformuler les contraintes en modifiant les ensembles, puis de paralléliser le code, cette proposition pourrait être une contribution avantageuse et un projet intéressant à faire. Malheureusement on a pas réaliser beaucoup de testes sur plusieurs maillages pour vérifier si notre implémentation était pérformante, bien qu'il n y a pas eu de bugs, en revanche, on peut affirmer qu'un grand pourcentage du travail est déjà fait et il ne reste pas grand chose.

Références

[1] Riccardo Scateni Max Lyon David Bommes Gianmarco Cherchi, Pierre Alliez. Selective padding for polycube-based hexahedral meshing. *Wiley, 10.1111/cgf.13593*, (hal-01970790), 2019.

[2] Free Software Foundation Inc. Gnu linear programming kit reference manual. August 2014.

[3] Kitware Inc. Paraview v5.8. https://www.paraview.org/.