Simulation d'un moteur à reluctance variable par la méthode des éléments finis

Laureline Willem, NOMA : 21381700

16 août 2021

ÉCOLE POLYTECHNIQUE DE LOUVAIN

Introduction

Dans le cadre du cours LEPL1110, j'ai implémenté un programme permettant de simuler un moteur à reluctance variable. Ce rapport vise à expliquer différents choix d'implémentation, fournir un rapport des tests réalisés afin de valider le code et déterminer l'ordre de précision de mes calculs.

1 Optimisations des fonctions

1.1 motorComputeMagneticPotential

Le premier point ici est de remarquer que l'équation permettant de trouver le potentiel magnétique, a, est donné par une équation de Poisson. Ceci est largement expliqué dans l'énoncé, je ne m'y attarde donc pas plus. Dans le cadre des devoirs, plusieurs solveurs capables de résoudre une équation de Poisson ont été implémenté. Ceux-ci m'ont donc servis de base pour cette fonction.
Ma première implémentation se base sur un solveur basique du système linéaire. Cette méthode n'est pas la plus efficace, et elle devient rapidement trop lente pour de trop gros maillage. Cependant, adapter le devoir 4 était assez rapide, c'est pourquoi je l'ai implémentée en premier pour assurer un code fonctionnel, quitte à l'optimiser par la suite. Cette fonction n'est plus utilisé mais peut être retrouvée dans la fonction *motorComputeMagneticPotentialBasic*.
J'ai ensuite implémenté une adaptation du code de l'équipe didactique présent dans *fem.c*. Il s'agit quasiment des même fonctions à la différences qu'elle utilisent un *motorMesh* et résolvent en prenant en compte les coefficients μ et j_s de chaque domaine.
Grâce à cette adaptation, je peux maintenant résoudre cette équation avec un solveur Bande. Ces fonctions proposent aussi une renumérotation selon X et Y. Je peux donc comparer leur efficacité en termes de temps de calcul et d'utilisation de mémoire.

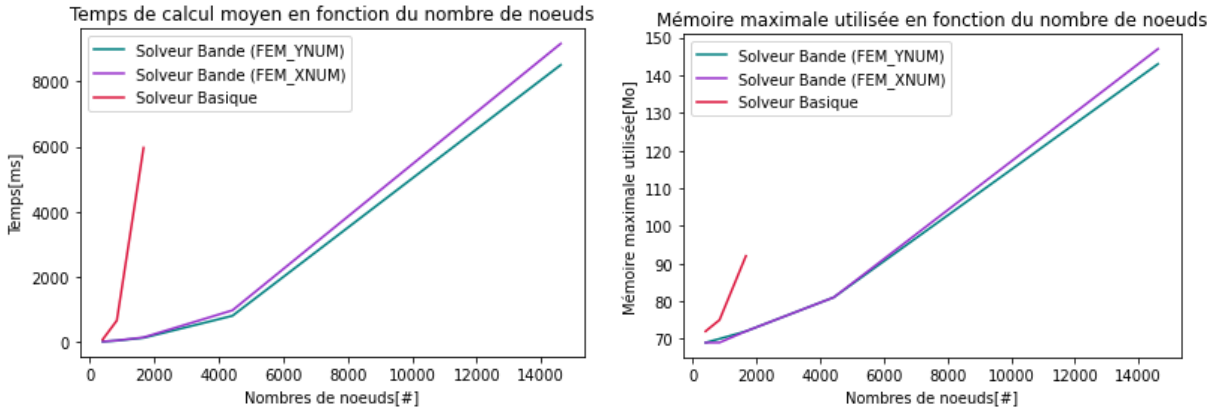

FIGURE 1 – Graphique d'analyse du temps de calcul et de l'utilisation de la mémoire en fonctions de la taille du maillage pour les différents maillages

Ces graphes nous indique que l'usage du solveur Bande avec une renumérotation en fonction de Y est est optimale pour le temps de calcul et pour la mémoire. C'est donc cette option que sera exécutée par le programme final.

1.2 motorComputeCouple

Cette fonction utilise à nouveau une édition du code fonctionnel du devoir 4. J'ai choisi d'utiliser la fonction *motorFree* ici, puisque l'espace d'intégration est toujours le même, j'ai mis en place la variable globale *theGlobalSpace* de type *femDiscrete**. Celle-ci est initialisée à la première itération, puis réutilisée pour toute les itérations suivantes. Elle est donc libérée en fin d'exécution du programme par l'appel de *MotorFree*. Je me suis rendue compte en retravaillant mon code que je calculais 2 fois le jacobien, ce calcul étant inutile, le second à été supprimé.

1.3 motorAdaptMesh

J'ai d'abord pensé à parcourir les éléments du maillage pour trouver les triangles optimaux pour chacun des points, mais j'ai cependant abandonné l'idée assez rapidement, car parcourir l'ensemble des éléments possibles pour chaque noeud aurait pris énormément de temps, ce qui serait contre productif dans le cadre d'un projet qui vise à optimiser le programme et sa vitesse d'exécution. Puisqu'on a un maillage déjà existant, on peut se baser sur celui-ci pour créer le nouveau. Je me suis alors rappelée de la structure *femEdges*, que nous avons découvert dans le cadre du devoir 3, et qui est fournie, avec ses fonctions de bases, dans *fem.c*. Grâce à celle-ci, il m'est possible de savoir à quels éléments appartient une arrête, et via la structure *motorMesh* (plus particulièrement, son champs *nElemDomain*) on peut alors de reconnaître les segments frontières entre deux domaines. Puisque le domaine air_gap est une bande fine de ou tout les noeuds sont sur une frontière, on peut alors simplement chercher le noeud optimal sur l'autre frontière de domaine air_gap. On a donc un programme en $O(n^2)$ puisqu'on parcourt l'ensemble des élément une fois pour trouver chaque segment à remailler, puis une seconde fois pour chaque segment frontière pour trouver le sommet optimal pour ce segment. Il ne reste alors qu'à modifier *motorMesh->elem* pour indiquer le nouvel élément. Comme cette méthode se base sur le maillage existant, elle donnera le même nombre d'élément.

1.3.1 Choix du critère pour un triangle optimal

J'ai choisi d'utiliser une autre méthode que celle présentée dans l'énoncé du projet. En effet, les triangles testés n'auront pas forcément le même périmètre via ma méthode. J'ai choisi de minimiser la médiane.

1.3.2 Limites de cette méthode

Si le domaine air_gap devait avoir des noeuds supplémentaires qui n'appartiennent pas à la frontière du domaine, le méthode ne générerait des éléments non-optimaux en trop petites quantités.

1.4 motorComputeCurrent

Il s'agit ici de savoir quelle bobine génère le couple le plus élevé en fonction de l'angle theta de rotation du moteur. Puisque la pièce centrale a quatre dent, on sait qu'on a un couple de période $\frac{\pi}{2}$. J'ai donc exécuté mon programme sur chaque bobine, récupéré la valeur calculée par le couple avant de retourner un couple de 0 au programme. Ce choix vise a maintenir une vitesse de rotation constante pour avoir une évolution de theta fixe. J'ai ensuite créer grâce un un script python un graphe du couple en fonction de l'angle theta pour chaque bobine, et déterminé les angles critique ou il fallait changer de bobine pour maximiser le couple. Ceci revient à relever les angles ou la courbe supérieure change sur le graphe. Le code a été modifié dans cette nouvelle version. En exécutant le code à nouveau, je me suis aperçu que la rotation perdait en vitesse sur les plus grand maillage. J'ai donc effectué de nouvelle mesure sur le maillage le plus grand (14608 noeuds) pour obtenir des angles critiques plus précis. Je me suis également aperçue que j'avais décalé la valeur de theta dans la mesure.

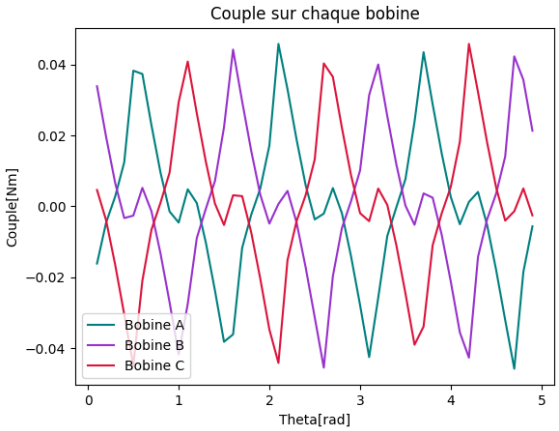

FIGURE 2 – Gaphique d'analyse du couple en fonction de theta sur le maillage 14608.

2 Validation du code

2.1 Fuite de mémoire

L'état de la mémoire pendant l'exécution peut-être vérifiée directement depuis l'outil Visual Studio 2019, à chaque instant pendant l'exécution. Cet outils m'a donc permis de vérifier qu'il n'y avait pas de fuite dans au sein des itérations. Pour m'assurer ne pas avoir de fuite à la fermeture du programme, j'ai utilisé l'outil Valgrind.

2.2 Résultats numérique

Puisque nous n'avons pas d'exemple de résultats, il est compliqué de vérifier l'exactitude des chiffres. Cependant, dans les consignes nous avons un exemple du visuel attendu pour le champ magnétique. De plus, j'ai comparé la valeur du couple sur les différents maillages. Des valeurs proches entre elles sont un bon indice de chiffres exacts.

2.3 Remaillage

Ici, pour valider le code deux éléments pouvaient être vérifié : le nombre d'élément crée, leur forme, et l'ordre du numérotage de leur noeuds. Pour en vérifier le nombre, la variable *starting* déjà présente dans le code et permettant d'itérer sur le maillage pour la réécriture nous permet aisément de récupérer la numéro du premier élément et de l'élément après le dernier élément remaillé.
La forme et le l'ordre des noeuds peut-être vérifié visuellement, puisque seul les noeuds numéroté dans l'ordre sont affiché en vert, et la forme est vérifiée visuellement.

3 Ordre de précision

Pour analyser la précision, j'ai exécuté mon code sur différents maillages et j'ai récupéré la valeur du couple à la première itération. En effet, cette valeur devrait être la même, et donc la variation entre ces valeurs me donnera une idée de la précision.

Nombre de noeuds	Couple
400	-0.011851
838	-0.027799
1667	-0.038487
4424	-0.044461
14608	-0.045268

TABLE 1 – Variation du couple sur la première itération entre les différents maillages

On choisit une référence. Ici, ce sera la valeur du couple sur le maillage à 14608 noeuds, puisque ce maillage est plus précis, la valeur couple sera plus proche de la réalité sur celui-ci. On calcule l'erreur sur chacun des quatre autre maillages par rapport à la référence via : $err = 1 - \frac{C_{test}}{C_{ref}}$. L'ordre de précision sera alors la valeur de la pente de la droite qui approxime la courbe du logarithme de l'erreur en fonction du logarithme du nombre de noeuds par régression linéaire.

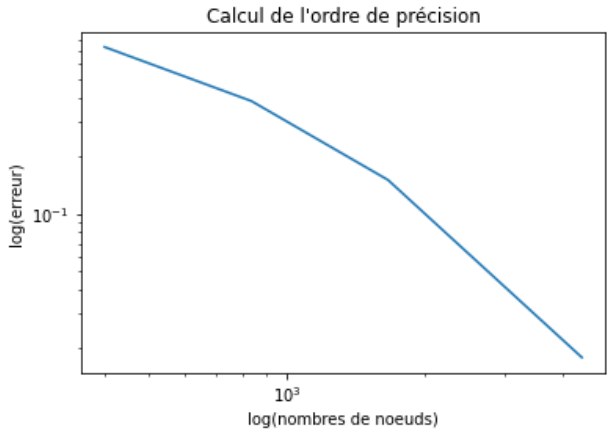

FIGURE 3 – Graphe du logarithme de l'erreur en fonction du logarithme du nombre de noeud

On trouve donc un ordre de précision de -1.56 grâce à une régression linéaire.

Conclusion

Je suis plutôt satisfaite du code. Celui-ci fonctionne bien pour tout les maillages que j'ai pu tester, et ce dans un temps raisonnable. J'ai pu créer plusieurs solutions afin de comparer les temps de calculs, ce qui m'a permis d'analyser différentes méthodes pour sélectionner la plus efficace.
Cependant, j'aimerais émettre quelques pistes d'amélioration. En effet, par manque de temps, je n'ai pas pu implémenter et expérimenter toutes mes idées d'optimisation. J'aurais aimé pouvoir tester plus de méthodes pour calculer le potentiel magnétique (je pense par exemple aux méthodes itératives). Ces méthodes auraient pu rivaliser avec mon code actuel. On aurait pu sacrifier un peu de précision pour avoir une estimation correcte plus rapidement. J'aurais également voulu alléger mon renumérotage. En effet, mon code renumérote tout les noeuds à chaque itération. On aurait pu imaginer ne renuméroter que les noeuds qui se déplacent pour encore diminuer le nombre de calculs à effectuer par la machine.

Références

— **Support du cours :** *https ://perso.uclouvain.be/vincent.legat/zouLab/epl1110.php*
— **Valgrind :** *https ://valgrind.org/*
— **Visual Studio :** *https ://visualstudio.microsoft.com/fr/*