Simulation numérique d'un dispositif de refroidissement

Javier Cladellas

December 15, 2021

Introduction

L'objectif de ce projet est d'analyser le comportement thermique d'un dispositif de refroidissement d'un micro-processeur, plus précisement, d'un dissipateur. Pour des questions de simplification du problème physique, seule la simulation thermique d'une ailette du dissipateur de chaleur sera traitée. Pour cela, des simulations dans plusieurs scénarios, tels que l'ajout d'un ventilateur et la variation des dimensions du dispositif, seront effectuées.

Tout d'abord, le fonctionnement du programme C++ pour simuler le problème sera traité. En plus sa structure et son utilisation seront expliquées. Ensuite, les résultats des simulations seront étudiés dans les differents scénarios.

La documentation du code est présente en annexe.

1 Programme

Le code associé à la simulation du problème en question a été construit en considérant non seulement sa lisibilité, mais aussi la simple utilisation, performance et adaptabilité à la simulation d'autres problèmes physiques.

1.1 Structure

Le programme est composé de:

- Un fichier principal (simu.cpp).

- Quatre fichiers header (.hpp), contenant des classes et fonctions pour la simulation du problème et traitement des résultats.
- Un fichier de configuration (simu.cfg), qui permet la modification des paramètres géométriques et physiques du problème.
- Un fichier Python (plots.py), en charge de générer les graphiques résultants des simulations.
- Deux dossiers contenant les solutions du problème, dans le cas bidimensionel et tridimensionel. Puis un dossier avec les graphiques associés.

Simu.cpp

Il s'agit du programme principal. Son role est de calculer les solutions des équations relationnées au problème en question, dans chaque scénario.

D'abord, le fichier de configuration est lu. Ensuite, les solutions numériques du problème correspondant aux paramètres sélectionnés sont calculées. Finalement, les solutions sont exportées en format csv pour la visualisation bidimensionelle, ou en format vtk pour visualisation tridimensionelle.

Pour des questions de factorisation de code, les solutions initiales du système instationnaire sont calculées au même endroit que les solutions du système stationnaire.
Selon le cas, la solution exacte du système stationnaire est calculée, et ensuite la solution exacte et numérique sont enregistrées, ou la solution du modèle instationnaire est calculée dans deux scénarios: un flux de chaleur constant, et une activation/désactivation du flux de chaleur.

Fichiers header

- **ConfigFileHandler.hpp**
  Ce fichier contient toutes les fonctions responsables de la lecture correcte du fichier de configuration.
  Celui-ci permet que l'ordre des paramètres du fichier ne soit pas important, tant que les valeurs soient placées respectivement à droite du nom du paramètre corréspondant, séparés d'un espace.
- **matrices.hpp**
  Le fichier regroupe des classes qui représentent différents objets dérivés de matrices.

Une classe pour les vecteurs et une classe pour les matrices tridiagonales a été programmée.
Les deux classes contiennent des méthodes, opérateurs et attributs nécéssaires pour que l'utilisation des instances soit pratique et intuitive.

- **ailette.hpp**
  Contient une classe "Ailette" qui représente une ailette du dissipateur. Cette classe est utilisée pour représenter la discretisation géométrique du dispositif de refroidissement.

- **textformats.hpp**
  Regroupe des classes responsables de la construction, traitement et exportation de fichiers contenant du texte.
  En particulier, les fichiers d'extension .csv et .vtk sont pris en compte. Le fichier comporte una classe mère "TextFile" qui traite des fichiers contenant du texte de manière générale. Les classes "CSV" et "VTK" héritent de celle-ci, et traitent les cas spécifiques aux types de fichier respectifs.

config.cfg

Il s'agit du fichier de configuration contentant les paramètres physiques et géométriques liés au problème.
Ce fichier doit contenir les noms de paramètres et les valeurs respéctives, que l'on veut traiter dans le code. Les éléments doivent être séparés par n'importe quel separateur par défaut de la bibliothèque standard de C++. Un exemple de la structure de ce fichier est la suivante.

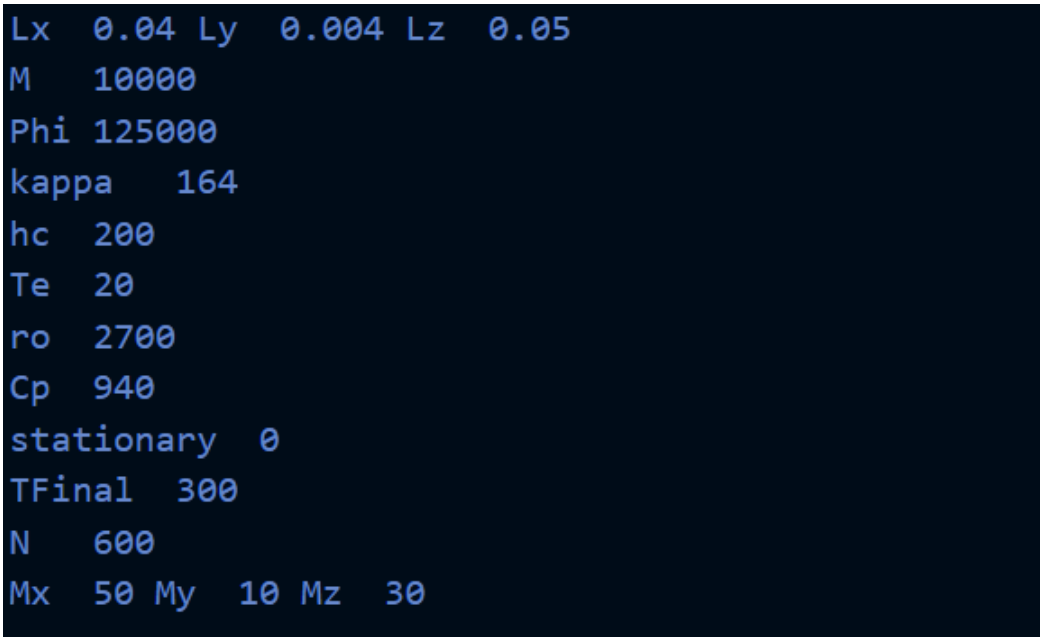

Figure 1: Exemple de fichier de configuration

plots.py

Programme Python qui s'en charge de générer des graphiques à partir des fichiers csv présents dans le dossier "2Dsolutions". Les images sont ensuite enregistrées à l'intérieur du dossier "Images", sous format png.

Dossiers target

Le dossier "2Dsolutions" garde les fichiers .csv qui contiennent les simulations thermique, que pour une composante (x) de l'ailette du dissipateur.
Le dossier "3Dsolutions" contient deux sous dossiers, qui correspondent à les simulations en espace et en temps du problème pour deux scénarios, en format vtk, pour la visualisation en 3D. En plus, la solution stationnaire en 3D est enregistrée dans la racine du dossier, puisqu'il s'agit d'un seul fichier vtk.

1.2 Utilisation

Pour compiler le programme simu.cpp, il suffit de rentrer en ligne de commande, situé à la racine du projet:

g++ -O3 simu.cpp -o simu

Note: changer "g++" par un autre compilateur C++ si nécéssaire.
Ensuite, pour executer la simulation il suffit de rentrer, aprés avoir compilé:

./simu simu.cfg

simu.cfg étant le fichier de configuration.
Il faut remarquer que les parties du code principal éxecutées vont dépendre du choix du paramètre "stationnary" (1 ou 0) du fichier de configuration, qui corréspondent au calcul d'une solution stationnaire ou instationnaire.
Finalement, pour générer les représentations graphiques des solutions, à partir des fichiers csv crées lors de l'exécution, il faut exécuter le fichier plots.py à l'aide d'un interpreteur Python.
Attention: le fichier python créera des graphiques que si le programme C++ a été executé pour les modèles stationnaire et instationnaire.
Les dossiers "S1" et "S2" du dossier "3Dsolutions" peuvent être lus par le logiciel Paraview.

1.3 Explication du programme

Le programme a comme but de simuler lo comportement thermique du dissipateur de chaleur d'un micro-processeur.
Tout d'abord, on considère que une ailette du dispositif a des dimensions L_x, L_y et L_z. La température T en un point dépend uniquement de la position selon l'axe x et de l'instant t.
Sur ce, il suffit de resoudre numériquement l'équation de la chaleur sur l'intervalle $[0, L_x]$, donnée par:

$$\begin{cases} \rho C_p \frac{\partial T}{\partial t} - \kappa \frac{\partial^2 T}{\partial x^2} + \frac{h_c p}{S}(T - T_e) = 0 \\ -\kappa \frac{\partial T}{\partial x}(0) = \Phi_p \\ -\kappa \frac{\partial T}{\partial x}(L_x) = 0 \end{cases} \tag{1}$$

Avec ρ la densité, C_p la chaleur spécifique à pression constante, κ la conductivité thermique, h_c le coefficient de transfert de chaleur surfacique, $S = L_y L_z$ l'aire d'une section transversale, et $p = 2(L_x + Lz)$ le périmètre d'une section transversale, Φ_p le flux de chaleur et T_e la température ambiante.

Dans le cas stationnaire, l'EDP peut être discrétisée avec $M+1$ points de discrétisation, obtenant l'équation

$$\begin{cases} -\kappa\frac{T_{i-1}-2T_i+T_{i+1}}{h^2} + \frac{h_c p}{S}(T_i - T_e) = 0, \forall i \in [1, M-1] \\ -\kappa\frac{T_1-T_0}{h} = \Phi_p \\ -\kappa\frac{T_{M-1}-T_M}{h} = 0 \end{cases} \tag{2}$$

Où h est le pas spatial L_x/M.

De la même façon, pour le cas instationnaire on a pour chaque temps t_n, $n \in [1, N]$:

$$\begin{cases} \rho C_p\frac{T_i^{n+1}-T_i^n}{\Delta t} - \kappa\frac{T_{i-1}^{n+1}-2T_i^{n+1}+T_{i+1}^{n+1}}{h^2} + \frac{h_c p}{S}(T_i^{n+1} - T_e) = 0, \forall i \in [1, M-1] \\ -\kappa\frac{T_1^{n+1}-T_0^{n+1}}{h} = \Phi_p \\ -\kappa\frac{T_{M-1}^{n+1}-T_M^{n+1}}{h} = 0 \end{cases} \tag{3}$$

Où Δt est le pas temporel t_{final}/N

Lecture des paramètres

Les paramètres mentionnés au-dessus, à modifier sur le fichier simu.cfg, seront lus par le programme en utilisant le fichier header ConfigFileHandler.hpp.
Premièrement, le fichier est lu en tant que chaîne de charactères. Ensuite, les éléments sont stockés dans deux autres chaînes de charactères, en alternant selon la parité de l'emplacement de l'élément.
De cette façon, les noms de paramètres se retrouveront dans un string, et les valeurs respectives seront dans un deuxième string. À l'aide d'un parseur, les strings sont convertis en tableaux de chaînes de charactères, qui ont le même ordre.
Finalement, la fonction *fetchParamValue<T>()* va s'en charger de chercher dans le tableau de valeurs, la valeur corréspondante au paramètre passé en argument. En plus, selon le paramètre template T, cette valeur est reconvertie au type souhaité.
Par exemple, le morceau de code suivant cherche la valeur (dans le tableau values) corréspondante au paramètre "Phi" du tableau params et la stocke dans la variable _phi.

```
_phi = fetchParamValue<double>("Phi",params,values);
```

Construction du système linéaire

Il faut remarquer que dans le cas stationnaire, résoudre l'équation discrétisée revient à résoudre un système linéaire $AT = F$, avec A une matrice tridiagonale.
Dans le cas instationnaire, à chaque pas de temps un système linéaire de ce type est résolu, considérant itérativement les solutions du pas de temps antérieur.
En tout cas, la matrice A a la même forme pour les deux modèles. Pour cela, les objets des deux modèles seront initialisés dans cette partie, selon les paramètres qui corréspondent au modèle.
Pour la construction numérique de ce système, les classes Vector et Tridiagonal sont utilisées. A est stockée dans un objet Tridiagonal, et F dans un objet Vector.

Pour économiser en mémoire et conserver la performance, la classe Tridiagonal stocke uniquement les trois diagonales de la matrice.

Résolution du système linéaire

Le système linéaire de ce style peut être résolu en décomposant la matrice A en $A = LU$. Avec L triangulaire inférieure, U triangulaire supérieure avec des 1 sur la diagonale. Il faut noter que L et U consèrvent la forme tridiagonale.
La matrice initialisée antérieurement est factorisée à l'aide de la méthode *decompositionLU()* de la classe Tridiagonal. Cette decomposition se fait "inplace" pour des questions de performance.
Ensuite, le système est résolu (par la méthode *solveLU(F)*) en effectuant une méthode de descente, puis de remontée. C'est à dire, on sépare le système $LUT = F$ en $UT = Y$ puis $LY = F$. Cette méthode de la classe Tridiagonal ne marche que si l'objet depuis où elle est appelée est déjà sous forme factorisée.
La résolution dans le cas instationnaire équivaut à la solution pour l'instant $t = 0$. Finalement, la solution est stockée dans un vecteur.

Modèle stationnaire

Puisque la solution numérique de ce modèle a été calculée dans la section antérieure, cette partie est dédiée au calcul d'une solution exacte, et

à l'enregistrement des solutions.
Une fois la solution exacte est calculée, un objet CSV est crée avec les deux solutions en colonne, et les valeurs de la coordonné en x corréspondantes aux points de discrétisation. L'objet CSV est ensuite enregistré sous forme de csv.
En ce qui concèrne la visualisation en 3D, un objet de type Ailette est instancié avec les dimensions de l'ailette et le maillage. À l'aide de la méthode *setTemperatures()*, les températures sont associées au points du maillage en interpolant linéairement la solution. Ensuite, un objet de type VTK est créé à partir de l'objet ailette, qui permetra d'enregistrer les solutions en tel format.

Modèle instationnaire

Les deux scénarios mentionnés auparavant sont traités simultanément, pour éviter de travailler avec deux boucles distinctes.
En premier, on construit deux vecteurs T_1 et T_2, qui contiendront la solution du système à chaque pas de temps antérieur au courant. Ces vecteurs sont initialisés avec des coefficients égaux à T_e. Ensuite, on crée les vecteurs vides T_1^{n+1} et T_2^{n+1} qui contiendront les solutions à chaque étape de la boucle, pour le scénario 1 et 2.

Les vecteurs F_1 et F_2 correspondant au vecteur second membre du système linéaire. Ils sont identiques au début de la boucle. À chaque étape, on associe à F:

$$F_1 = (\Phi_p, \frac{h_c p}{S} T_e + \frac{\rho C_p}{\Delta t} T_{i,1}, \cdots, \frac{h_c p}{S} T_e + \frac{\rho C_p}{\Delta t} T_{i,M-1}, 0) \tag{4}$$

Pour l'activation et désactivation des flux de chaleur, F_2 a le même valeurs que F_1, sauf que chaque 30 secondes, $\Phi_p = 0$. À chaque pas de temps, on utilise la solution du pas de temps antérieur pour trouver la solution actuelle, en résolvant les systèmes $AT_1^{n+1} = F_1$ et $AT_2^{n+1} = F_2$

Chaque solution est ensuite ajoutée à des objets CSS, pour chaque scénario. Ce CSV est enregistré au final du programme.
On ajoute a des objets Ailette les solutions en interpolant de la même façon que pour le cas stationnaire. À chaque pas, tel objet est converti en objet VTK et enregistré.

2 Analyse des résultats

2.1 Modèle stationnaire

On considère les paramètres:

nom	valeur	unité	nom	valeur	unité
ρ	2700	$kg/(m^3)$	h_c	200	$W/(m^2 \cdot K)$
C_p	940	$J/(kg \cdot K)$	L_x	0.04	m
κ	164	$W/(m \cdot K)$	L_y	0.004	m
T_e	20	°C	L_z	0.05	m
Φ_p	$1.25 \cdot 10^5$	W/m^2			

Figure 2: Valeurs des paramètres physiques et géométriques

Ci-dessous, on observe la solution numérique (bleu) du modèle stationnaire avec ces paramètres, ainsi que la solution exacte (rouge) à chaque point de discrétisation. 10 000 points de discrétisation sont utilisés.

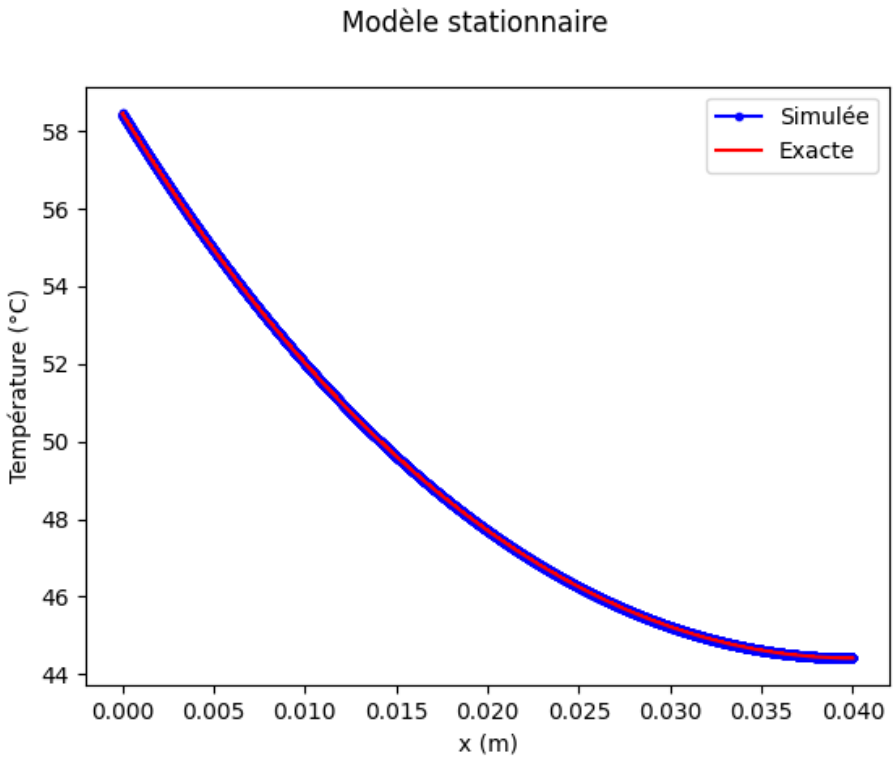

Figure 3: Représentation graphique de la solution numérique et exacte du modèle stationnaire, avec $L_x = 0.04m$

Les solutions semblent bien corréspondre, ce qui implique un comportement sosuhaité du programme. On remarque que la température au point le plus proche du micro-processeur est de 58°, pendant que la température au point le plus éloignée descend à environ 44°.
On s'intéresse a voir ce qui se passe lorsque la taille de l'ailette varie. Sur ce, on voit à continuation les graphiques pour des longueurs $L_x = 0.02m$ et $L_x = 0.04m$.

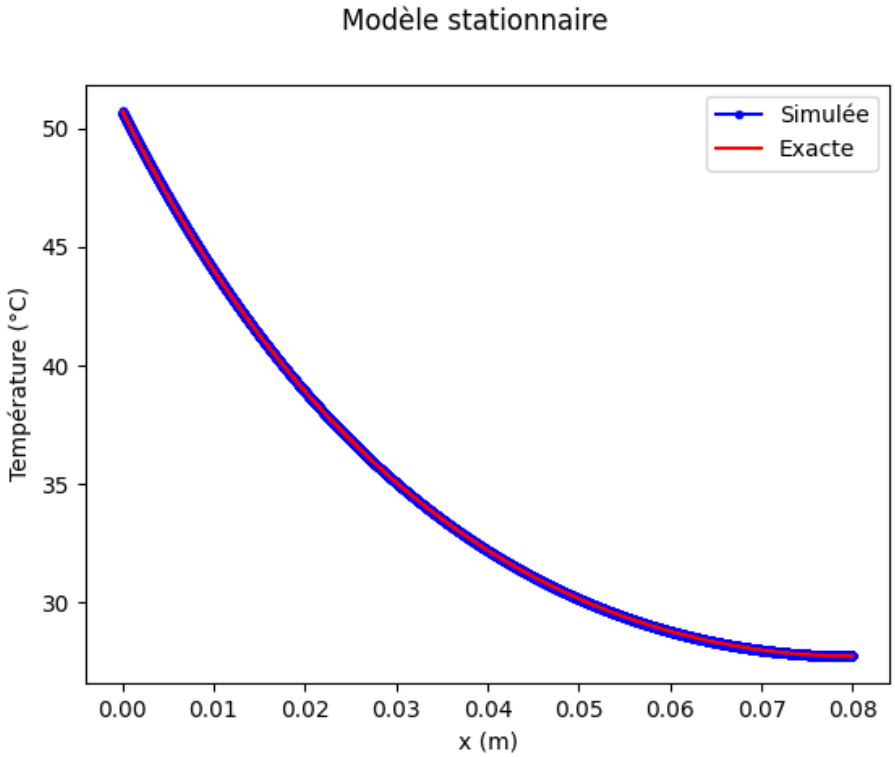

Figure 4: $L_x = 0.08m$.

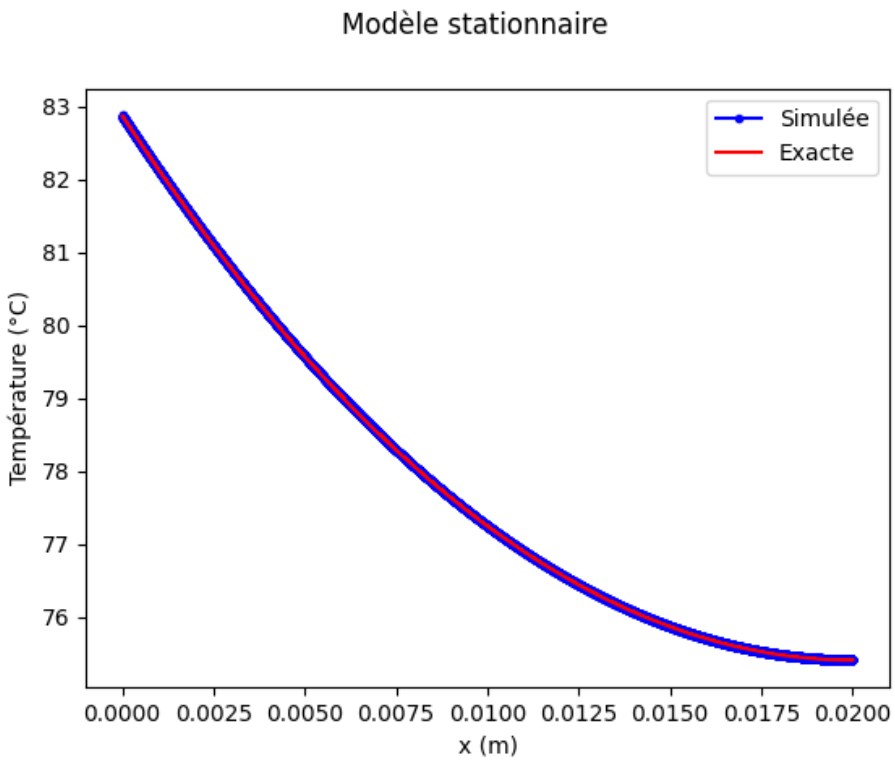

Figure 5: $L_x = 0.02m$.

On peut en conjeturer que plus l'ailette est longue, moins la température de celle-ci augmente. En plus, la température semble descendre plus rapidement quand l'ailette est longue, au cours de celle-ci. La température ne peut pas descendre plus que la température ambiente.
Lorsque l'on augmente les autres dimensions de l'ailette, on obsere (dans la figure suivante) que la température de l'ailette augmente assez rapidement.

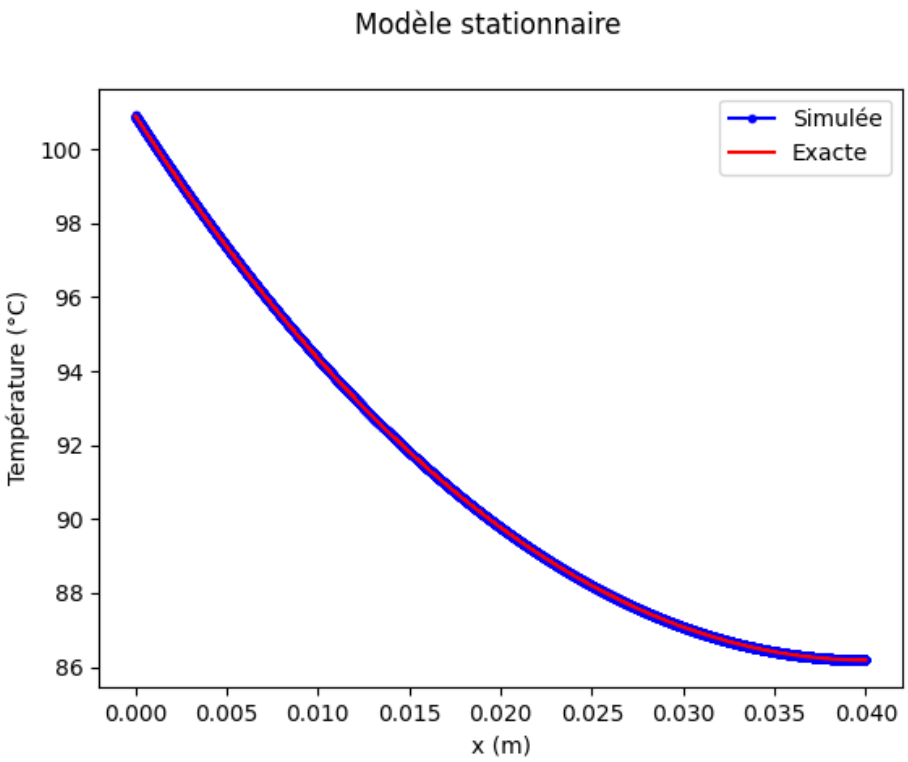

Figure 6: $L_x = 0.04m, L_y = 0.01m, Lz = 0.1m$.

Voici une visualisation tridimensionelle de la temperature de l'ailette du dissipateur.

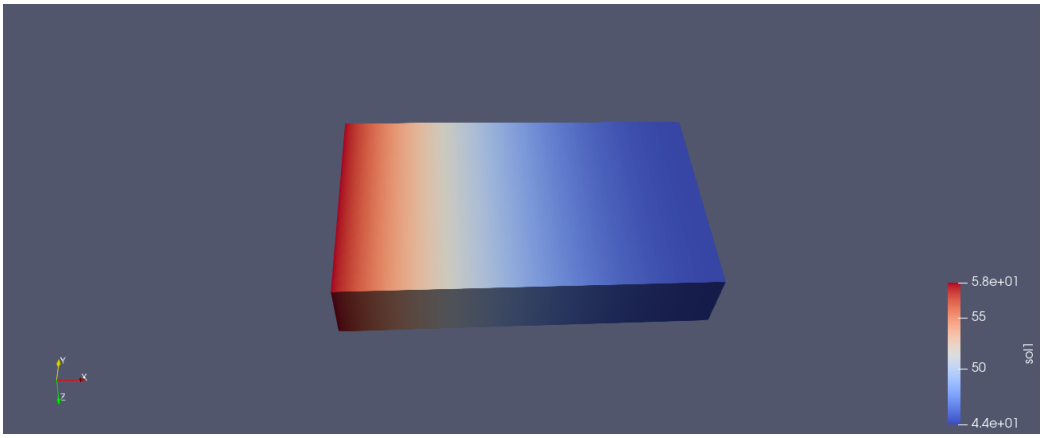

Figure 7:Visualisation 3D de solution stationnaire.

2.2 Modèle instationnaire: Flux de chaleurs constants

On s'interesse à analyser le comportement thermique de l'ailette au cours du temps. Dans la figure à continuation, on observe la température au long de l'ailette, dans des certains temps, lorsque les flux de chaleur sont constants:

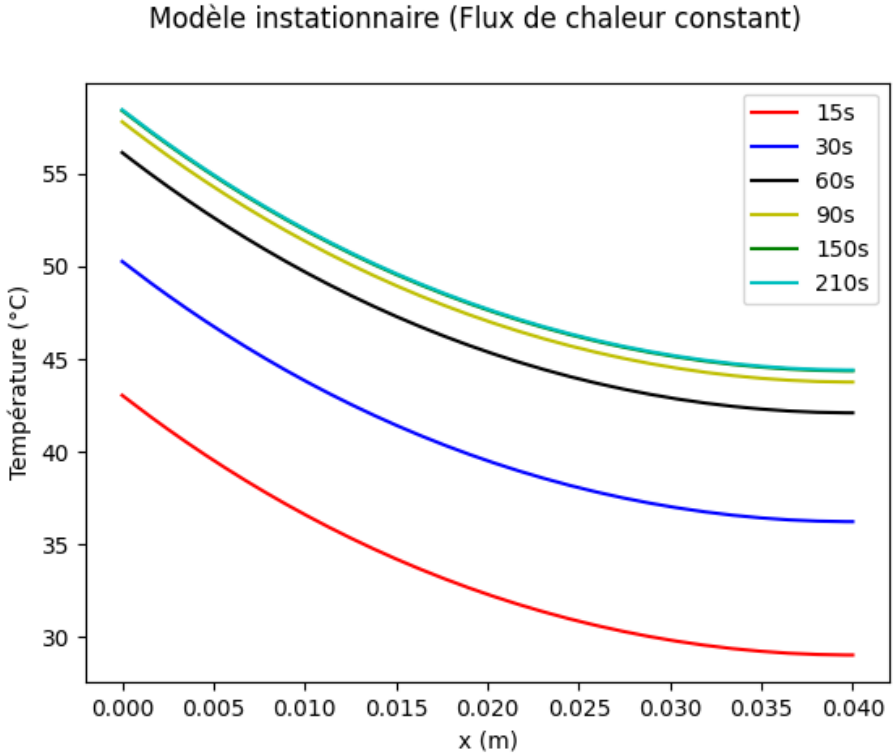

Figure 8: Solution du modèle instationnaire au cours du temps.

On remarque, au cours du temps, la température au long de l'ailette augmente et semble converger vers la solution stationnaire du problème. Au début de la simulation, la variation de température est remarquable, mais vers la fin celle-ci varie peu.
Lorsque l'ailette a une longueur selon la composante x plus grande, la température semble converger au cours du temps vers une température plus basse que le cas antérieur (fig. 9).

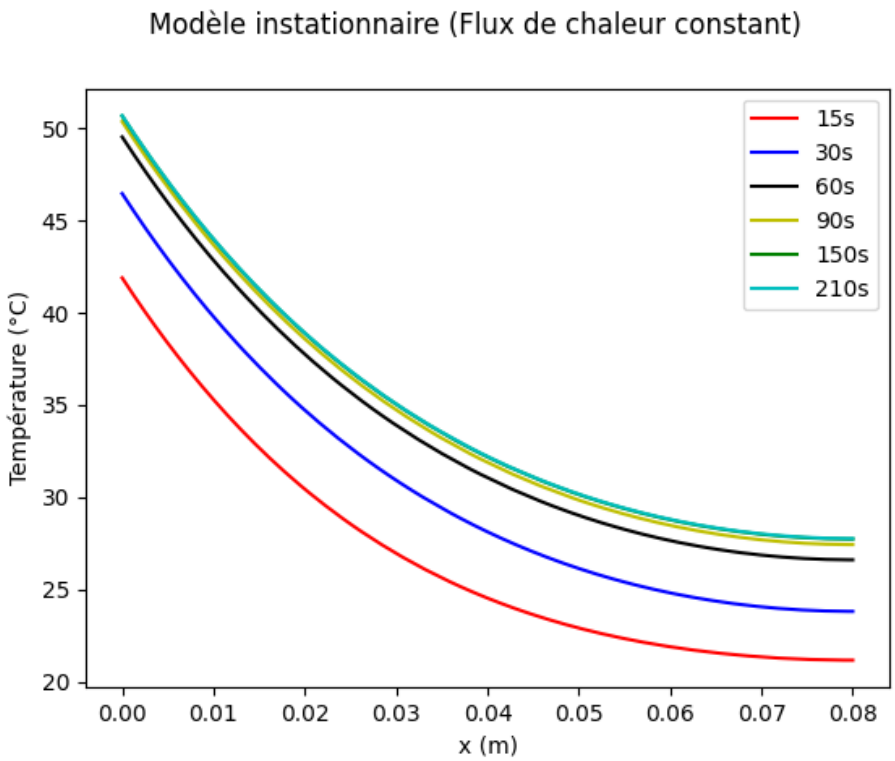

Figure 9: Solution du modèle instationnaire au cours du temps, $L_x = 0.08m$.

À continuation, les visualisations 3D de la solution instationnaire pour au bout de 30s, 100s et 300s.

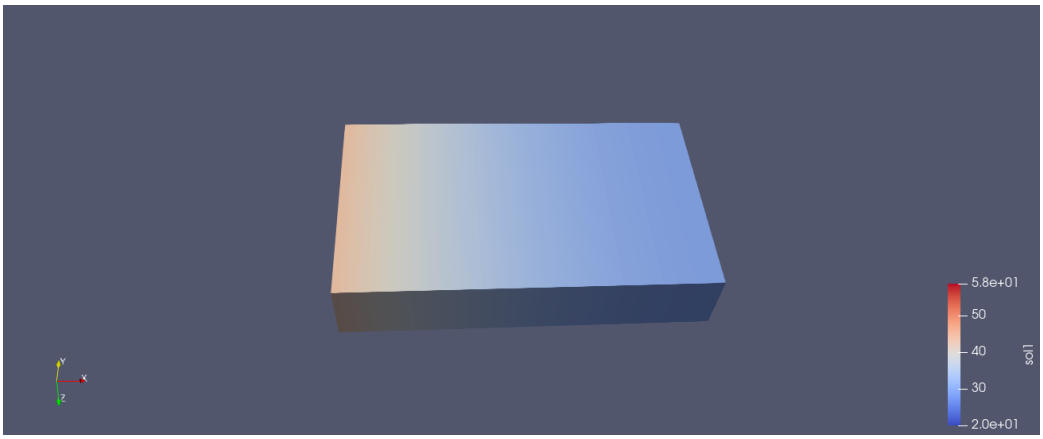

Figure 10: Représentation 3D du modèle instationnaire, t = 15s.

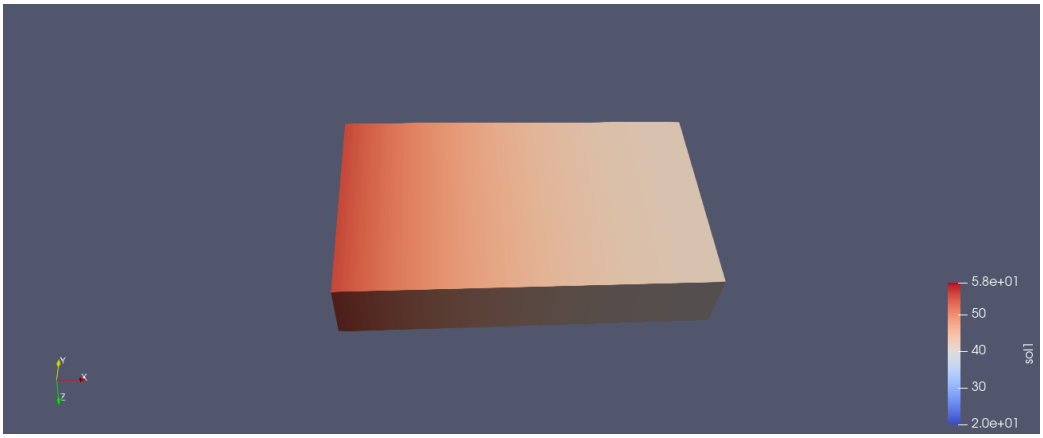

Figure 11: Représentation 3D du modèle instationnaire, t = 50s.

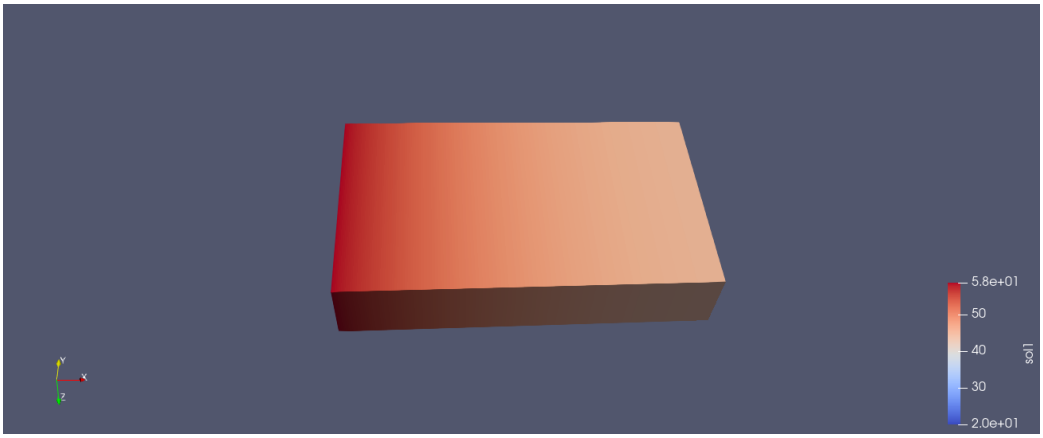

Figure 12: Représentation 3D du modèle instationnaire, t = 150s.

2.3 Modèle instationnaire: Variation du flux de chaleur

Lorsque les flux de chaleurs sont activés et désactivés chaque 30s, on observe le comportement suivant:

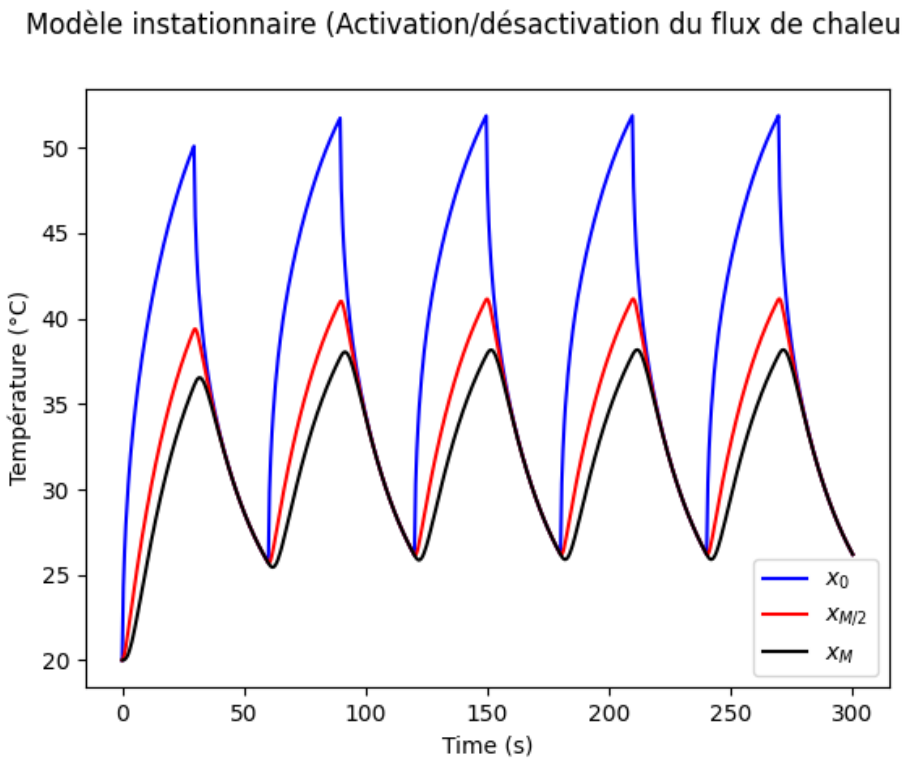

Figure 12: Activation/désactivation des flux de chaleur.

Lorsqu'on désactive les flux de chaleur, la température de l'ailette dans tous ces points diminue notablement. Par contre, cette diminution est plus présente au points plus proches du micro-processeur que pour les points éloignés. Quand les flux sont reactivés, la température augmente assez rapidement.

Conclusion

En conclusion, le comportement thermique d'une ailette de dissipateur a été simulée en effectuant une discrétisation en une dizaine de miliers de points pour l'espace, et en quelques centaines pour le temps.
Même s'il s'agit d'un nombre considérable d'itérations, le programme s'execute en un temps considérablement rapide. La plupart du temps est consacrée a l'écriture des fichiers.
Comme attendu, la température de l'ailette descend selon le long de celle-ci.

Ses dimensions ont un grand impact dans sa température. En plus, si les flux de chaleur sont activés et désactivés périodiquement, la température maximale atteinte va être inférieure à celle où les flux de chaleur sont constants. Plus de recherches peuvent être faites sur le comportement thermique de tout le dissipateur, en tant que ensemble d'ailettes, pour vérifier l'impact du nombre d'ailettes, ainsi que de la séparation entre chaqu'une.
Il serait encore intéressant d'évaluer la performance de la résolution de ce problème lorsque le programme est excécuté en parallèle.

Annexe: documentation

Vector

Cette classe a comme attributs un tableau de doubles contenant le vecteur et la taille de celcui-ci.
Elle a été construite dans le but de faciliter le traitement des vecteurs en tant que tableaux C++.
La classe compte avec les opérateurs classiques dans le calcul vectoriel, ainsi qu'une méthode pour calculer sa norme, et la surcharge de l'opérateur d'affichage dans la librairie standard de C++.

Tridiagonal

La classe Tridiagonal créé des matrices tridiagonales, en stockant uniquement les élément de la diagonal.
Il est possible d'effectuer les operations classiques avec les matrices tridiagonales (somme, multiplication, etc.). Par contre, seule les opérations entre objets du même type ont été programmées. Il est aussi possible d'afficher ces matrices, et d'acceder aux éléments avec les opérateurs (i,j).
Les méthodes les plus importantes sont les suivantes:

- decompositionLU(inplace=True):
  Cette fonction prend un paramètre booléan *inplace*, vrai par défaut. Si celui-ci est vrai, la décomposition LU de la matrice tridiagonale est stockée dans le même objet Tridiagonal, modifiant ainsi ses attributs. Si le booléan est faux, la fonction retourne un tuple (objet de la classe LUWrapper), contenant les matrices L et U.
  La décomposition est faite par la méthode de Gauss.
  Si inplace est vrai, la diagonale inférieure de L est stockée dans la diagonale inférieure de l'objet en question. La diagonale de L est stockée dans la diagonale de l'objet. Finalement, la diagonale supérieur de U est stockée dans la diagonale supérieur de l'objet, et la fonction retourne un tuple vide après avoir modifié les attributs

- solveLU(Vector const& F):
  Prend en paramètre le vecteur second membre du système linéaire

$AX = F$ et retourne le vecteur solution X. La résolution est faite en utilisant l'algorithme de descente et remontée.
La méthode retourne la solution souhaitée dans un objet Vector que si une factorisation LU a été faite auparavant.

TextFile

Il s'agit de la classe mère des classes CSV et VTK. Elle sert a manipuler des objets contenant que des chaînes de charactères.
Les objets peuvent être concatenés verticalement à l'aide des opérateurs + et +=. La concaténation peut se faire avec des objets du même type, ou avec des objets std::string. Il est possible d'enregistrer les objets dans un endroit souhaité.

CSV

Classe qui hérite de TextFile, elle sert a manipuler des objets sous la forme d'un csv. Un objet csv peut être construit à l'aide d'une liste d'objets Vector (std::initializer_list). Les objets CSV peuvent être concatenés horizontalement (dans l'axe des colonnes), avec les opérateurs * et *=.

VTK

Hérite de TextFile, permet de construire un fichier VTK avec un format spécifique.
Une méthode importante de cette classe est *buildMesh(int dim1, int dim2,int dim3)*. Elle prend en paramètres les dimensions du maillage d'un pavé, et construit un maillage sous la forme souhaitée des formats vtk.
Un objet de cette clase peut être construit avec un objet de type Ailette. Un maillage est construit, avec les attributs de cet objet, ainsi que les données nécessaires sont ajoutées.
Les méthodes *addData(Vector T)* et *appendData(Vector T)* servent a ajouter des données à l'objet à l'aide d'un vecteur. La différence entre les méthodes est que la première ajoute le début des données dans la partie corréspondante du format VTK.

Ailette

Représente une ailette d'un dissipateur de chaleur.
Possède comme attributs les dimensions de lobjet physique, ainsi que les dimensions du maillage souhaité et un vecteur avec les températures de l'objet dans les points du maillage. La fonction *setTemperatures(Vector T)* fait une interpolation linéaire du vecteur T, pour trouver les valeurs corréspondantes aux points du maillage. L'interpolation est faite que sur la composante x de l'ailette.