Rapport

Marie Sengler

December 13, 2023

1 Modèle stationnaire

1.1 Description de la classe

La classe **Stationnaire** est constituée de six attributs : cinq tableaux double* M, L, U, F et T, ainsi que la taille Mtaille. La matrice M et le vecteur F correspondent à ceux de notre système. L et U représentent la décomposition LU de M. Enfin, T est le vecteur solution de notre système. M, L et U sont des matrices carrées de taille (Mtaille+1)×(Mtaille+1), tandis que F et T sont des vecteurs de taille Mtaille +1. Pour accéder à l'élément (i, j) d'une matrice, on utilise l'indice j*(Mtaille +1) + i du tableau correspondant à cette matrice.

Lors de la construction de l'objet, les coefficients de M et F sont mis en place en appelant les méthodes `void set_M` et `void set_F`.

Ensuite, la fonction `void decomposition_LU(double* M, double* L, double* U, int taille)` est appelée, implémentant les coefficients de L et U. Cette fonction prend en paramètre les matrices M, L et U, ainsi que la taille du système. Elle utilise l'algorithme de décomposition LU adapté à une matrice tri-diagonale.

Les valeurs de notre solution T sont complétées à l'aide de la fonction `double* resoudre(double* L, double* U, double* F, int taille)`. Cette fonction externe renvoie le vecteur solution, nommé X. Elle utilise un algorithme de descente et de remontée, en considérant que L ne comporte qu'une diagonale et une sous-diagonale, tandis que U ne comporte qu'une diagonale et une sur-diagonale. Cette fonction est appelée depuis des méthodes de notre classe.

La classe **Stationnaire** comprend plusieurs accesseurs pour obtenir la taille du système (`void taille()`), ainsi que pour visualiser les matrices M (`void matrice()`), L (`void decompL()`) et U (`void decompU()`), ainsi que le vecteur F (`void vecteurF()`). Pour afficher ces matrices, on fait appel à une fonction externe (`void printMatrix(double* M, int ligne, int colonne)`) qui prend en paramètre la matrice à visualiser, ainsi que son nombre de lignes et de colonnes.

De plus, elle contient deux méthodes qui vérifient que le système est bien résolu :
1) `void LU()` qui calcule le produit des matrices LU grâce à la fonction externe `double* matmul (double* A, double* B, int ligA, int colA, int colB)`, puis affiche le résultat. On est censé retrouver notre matrice M.
2) `void MT()`, qui effectue le produit de MT, et affiche le résultat. On est censé retrouver notre vecteur F.

La méthode statique `static double Texact(double x)` renvoie la solution exacte à la distance x.

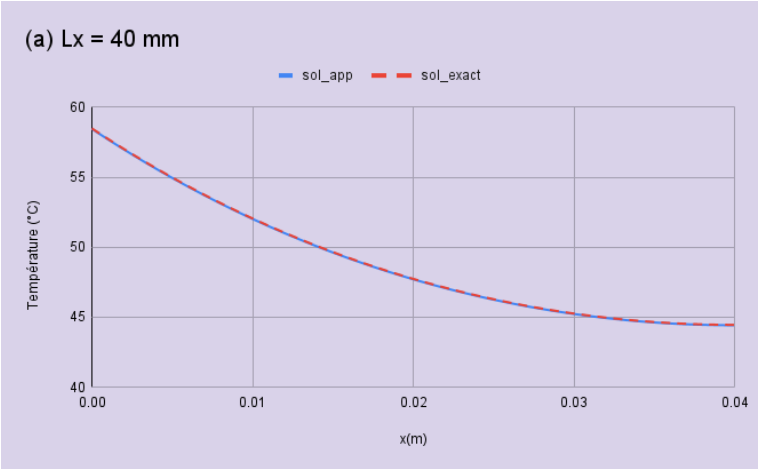
La méthode `double erreur()` revoie l'erreur global: $e_{global} = ||T_{exact} - T_{app}||$. Pour ce faire, elle utilise la fonction résoudre et évalue la méthode Texact en $i * \frac{L_x}{M_{taille}}, \forall i \in [0, M_{taille}]$.

Enfin, la méthode `void save_csv(std::string format = "csv")` permet d'enregistrer les résultats dans un fichier nommé "Stationnaire". On peut choisir le format du fichier, le format par défaut étant .csv, mais on peut également opter pour .txt, .dat selon nos besoins. Cette méthode utilise `std::ofstream`. Les résultats sont enregistrés sous forme de trois colonnes nommées : x, sol_part, sol_exact, de taille $M_{taille}+2$. Sur la ligne i, on retrouve dans l'ordre la distance: $i * \frac{L_x}{M_{taille}}$, la température approchée: $T[i]$, la température exacte: $Texact(i * \frac{L_x}{M_{taille}})$.

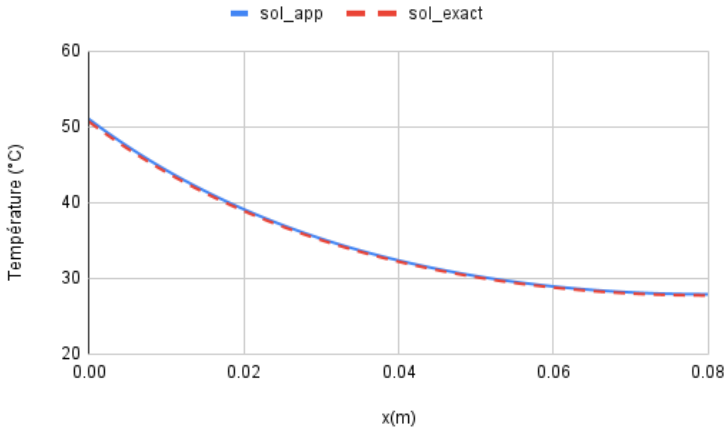
1.2 Analyse des résultats

On lance notre programme avec le paramètre M = 1000. On enregistre les résultats au format CSV en utilisant la méthode `save_csv()`. On peut ainsi le lire avec un tableur. Dans les paramètres régionaux du tableur, choisissez "États-Unis" pour assurer que les nombres soient interprétés correctement en format américain. On importe les données dans le tableur en utilisant un espace comme séparateur. On peut enfin tracer un graphique pour visualiser les résultats.

(a) Lx = 40 mm



(b) Lx = 80 mm



On remarque que les courbes sol_exact et sol_app sont presque confondues. De plus, notre fonction d'erreur semble converger vers 0, lorsque M devient grand. Voici quelques résultats de la fonction d'erreur pour Lx = 40 mm:
10: 3.07437
100: 0.272509
500: 0.0539596
1000: 0.0269464
5000: 0.00538393
Pour de petites valeurs de M, on constate que la précision n'est pas très élevée. On remarque que pour Lx = 80 mm, la courbe ressemble à celle avec Lx = 40 mm, mais avec des température plus basse.

2 Modèle non stationnaire

2.1 Construction du modèle

On réécrit la discrétisassions pour séparer les différents points en fonction du temps. Ce qui nous donne :

$\rho C_p \frac{T_i^{n+1}-T_i^n}{\Delta t} - \kappa \frac{T_{i-1}^{n+1}-2T_i^{n+1}+T_{i+1}^{n+1}}{h^2} + \frac{h_c p}{S}(T_i^{n+1} - T_e) = 0$
donc $-\frac{\kappa}{h^2}T_{i-1}^{n+1} + (\frac{\rho C_p}{\Delta t} + \frac{2\kappa}{h^2} + \frac{h_c p}{S})T_i^{n+1} - \frac{\kappa}{h^2}T_{i-1}^{n+1} = \frac{\rho C_p}{\Delta t}T_i^n + \frac{h_c p T_e}{S}$

On retrouve un système de la même forme que le modèle stationnaire avec une matrice tri-diagonale, mais avec des coefficients différents. On remarque que le vecteur F est variable, et dépend du temps précédent.

On prend pour la matrice M :
$a_i = \frac{-\kappa}{h^2}, \forall x \in [1, M-1]$
$a_M = -\frac{-\kappa}{k}$
$b_i = (\frac{\rho C_p}{\Delta t} + \frac{2\kappa}{h^2} + \frac{h_c p}{S}), \forall x \in [1, M-1]$
$b_0 = \frac{-\kappa}{k}$
$b_M = \frac{-\kappa}{k}$
$c_i = \frac{\kappa}{h^2}, \forall x \in [1, M-1]$
$c_0 = -\frac{-\kappa}{k}$

On prend pour le vecteur F :
$F_i = \frac{\rho C_p}{\Delta t}T_i^n + \frac{h_c p T_e}{S}, \forall x \in [1, M-1]$
$F_0 = \phi_p$
$F_M = 0$

2.2 Description de la classe

Pour implémenter ce schéma, nous avons créé une nouvelle classe, la classe **NonStationnaire**, qui partage des attributs communs tels que les tableaux `double* M, L, U, F`, ainsi que la taille `Mtaille`. Nous ajoutons également l'entier `Mtemps`, qui correspond au nombre de secondes pendant lesquelles nous voulons étudier le système. De plus, deux tableaux `double* Ti` et `Ttps` sont ajoutés. Dans `Ti` de taille $Mtaille + 1$, nous enregistrons les résultats correspondant à la température en fonction de la distance pour un flux de chaleur constant. Dans le deuxième tableau `Ttps` de taille $Mtemps*3$, nous enregistrons les résultats correspondant à la température en fonction du temps, mesurés en x=0, x=M/2 et x=M du flux de chaleur avec activation ou désactivation. Enfin, pour déterminer comment gérer l'activation du flux de chaleur, nous avons ajouté un dernier attribut booléen : `phi_const`.

Le constructeur de la classe prend trois arguments dont un obligatoire la taille, deux optionnels: le temps initialisé par défaut à 300 secondes, et un

booléen pour `phi_const` initialiser par défaut à true, indiquant que le flux de chaleur est constant. Ensuite, on utilise l'allocation dynamique sur nos différents tableaux. Et, on fait appelle aux méthodes `set_M`, `set_F`, et `set_Ti`. Enfin, pour résoudre le système, on appelle la fonction `void decompositio_LU(double* M, double* L, double* U, int taille)` que l'on a déjà utiliser et la méthode `iteration()`.

On a un peu près les mêmes assesseurs que dans la première classe, auxquels on a ajouté `void vecteurTtps()` qui permet de visualiser les résultats en fonctions du temps.

La méthode `set_Ti` initialise le vecteur Ti à la température T_e. Les méthodes `set_M` et `set_F` fonctionnent de la même manière que dans la classe Stationnaire, réadaptées avec les valeurs des coefficients de M et F calculées. On a surcharger la méthode `set_F` avec un paramètre booléen permettant de modifier les valeurs du vecteur F lorsque l'on désactive le flux de chaleur.

Ensuite, la méthode `void iteration()` nous permet de résoudre concrètement le système. Pour cela, on utilise une boucle for qui va de 1 jusqu'à Mtemps. À chaque itération, on appelle la fonction amie `double* resoudre (double* L, double* U, double* F, int taille)` déjà présentée. On remplace Ti par le tableau rendu par la fonction. Ensuite, on récupère les 3 valeurs d'une colonne de Ttps à l'emplacement x=0, x=M/2 et x=M. Enfin, on change les valeurs de F dépendant de Ti, en fonction de l'activation ou non du flux de chaleur, et on passe à l'itération suivante.
La méthode `void uneiteration()` modifie les résultats en espaces après un pas de temps. Elle fonctionne sur le même principe que la méthode `itération`. Elle va nous servir pour visualiser les solutions en 3D.

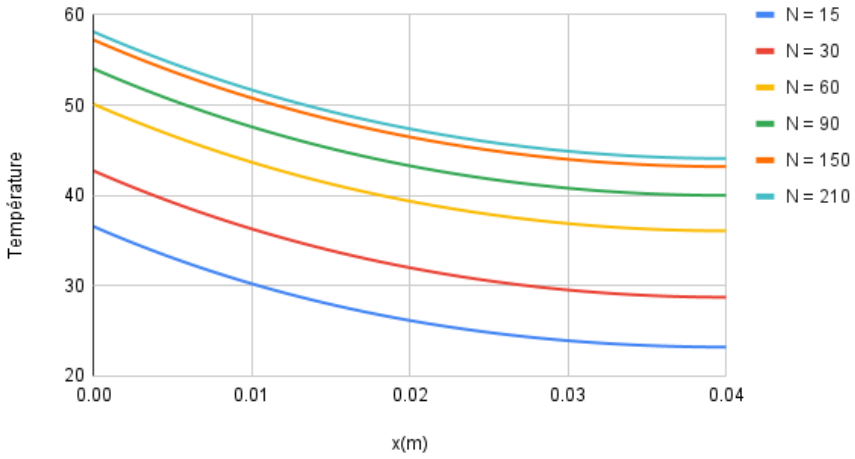
Notre classe a une méthode `double erreur()` similaire à celle de la classe précédente, permettant de montrer la convergence de la solution vers celle du modèle stationnaire.

Les deux dernière méthodes sont `void save_csv(std::string format = "csv")` et void save_csv_temps(std::string format = "csv") qui fonctionnent de la même manière que celle de la classe Stationnaire. Les noms des fichiers changent en fonction du temps choisi. Les colonnes de la première fonctions sont x et sol_app, et celle de la deuxième t, x_0, x_M2 et x_M.

2.3 Analyse des résultats

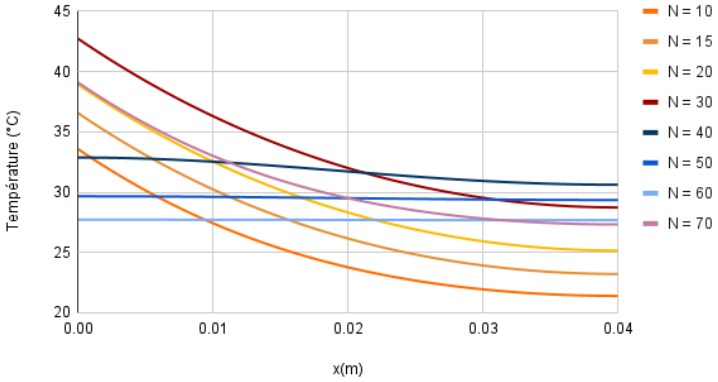
Comme pour la première partie, on importe nos fichiers dans un tableur. On va d'abord traiter nos données en espace, puis en temps, en prenant en compte l'activation et la désactivation du flux de chaleur.

(a) Flux de chaleur constant



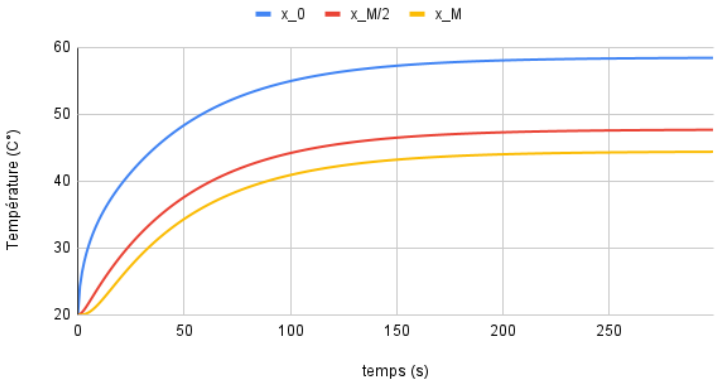
On observe sur le premier graphique, avec un flux constant, que la courbe se translate vers des températures plus chaudes avec le temps, jusqu'à converger vers une courbe qui ressemble à notre modèle stationnaire. En effet, on observe qu'à une différence en temps plus espacé, l'écart entre les courbes se resserre, pour des N grands.
On peut vérifier que notre solution converge en l'infini vers la solution stationnaire :
N = 50 : erreur = 10.2967
N = 100 : erreur = 3.57607
N = 250 : erreur = 0.126219
N = 500 : erreur = 0.0261533
N = 1000 : erreur = 0.0269463
On a pris M constant égal à 1000. L'erreur devient de plus en plus petit jusqu'à un certain point. C'est normal pour que la méthode converge, on doit avoir $N << M$. Il faudrait alors augmenter M, pour avoir une meilleur convergence.

(b) Avec activation/désactivation du flux de chaleur en espace

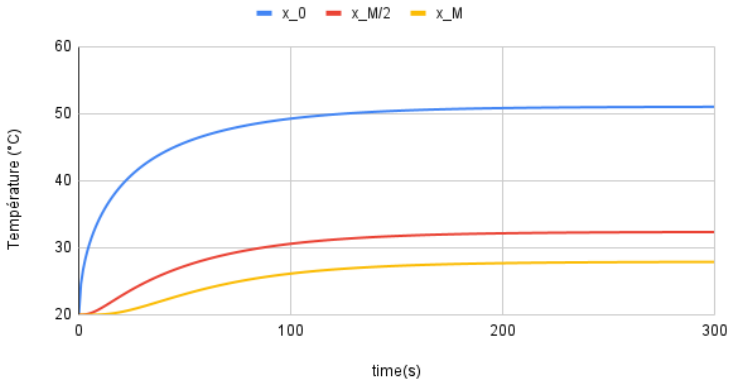


Sur ce graphique, on observe l'évolution de la température avec activation/ désactivation du flux de chaleur. Les courbes en bleu correspondent au moment de désactivation. On observe que la température pendant le moment de désactivation diminue, et que les courbes ont tendance à s'aplatir.

(a) Flux de chaleur constant

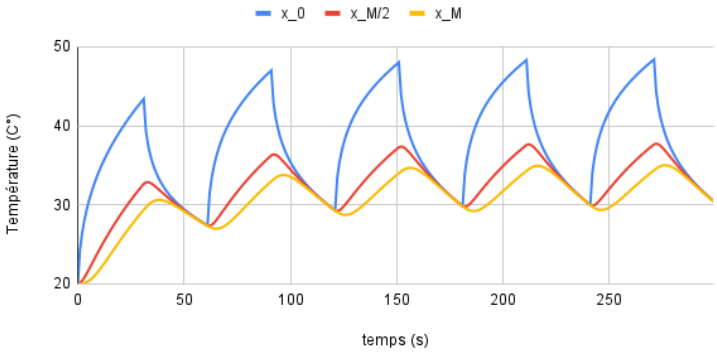


(a) Flux de chaleur constant avec Lx = 80 mm

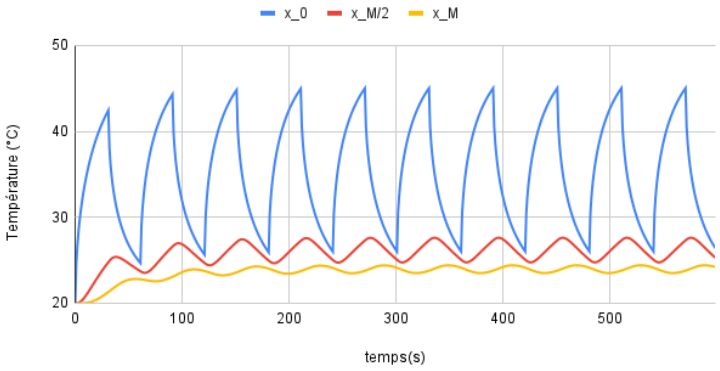


On voit ici l'évolution de la température avec un flux constant pour certaine valeur de x en fonction du temps. On observe une plus grande différence entre x_0 et x_M/2, qu'entre x_M/2 et x_M. Cette différence s'accentue avec Lx = 80 mm. On a des pertes thermiques.

(b) Avec activation/désactivation du flux de chaleur en fonction du temps



(b) Avec activation/désactivation du flux de chaleur (Lx = 80 mm)



On observe que la courbe de température avec activation / désactivation du flux de chaleur forme par pic, de crochet, où le maximum est atteint lors ou légèrement après le moment de désactivation du flux.
Étant donnée que x0 est le plus proche de la source de chaleur et du ventilateur, sa courbe subit de plus forte variation, que celle placé à la moitié et à la fin de l'ailette. Celle-ci sont plus amortie, le pic devient plus arrondi et présentent un léger décalage par rapport aux moments d'activation et de désactivation : la propagation de la chaleur et en effet pas instantanée. Avec Lx = 80 mm, les pics de la courbe x_0 sont plus accentué qu'avec Lx = 40 mm, par rapport à x_M/2 et x_M qui ressemblent plus à des petites vaguelettes. En effet, avec Lx=80 mm, le temps de diffusion est plus lent pour atteindre x_M/2 et x_M.

3 Visualisation 3D

3.1 Interpolation

Calcul de $\hat{T}_i$: On remarque que calculer les coefficients de la droite dans l'algorithme revenait à avoir :
$a = \frac{T_{k+1}-T_k}{x_{k+1}-x_k}$ et $b = \frac{x_{k+1}T_k - x_k T_{k+1}}{x_{k+1}-x_k}$
On cherche k tel que $x_k < x_{i00} < x_{k+1}$. Ce qui revient sur les différents discrétisassions de taille M et M_x : $\frac{kL_x}{M+1} < \frac{iL_x}{M_x+1} < \frac{(k+1)L_x}{M+1}$
On a alors : $\frac{i(M+1)}{M_x+1} - 1 < k < \frac{i(M+1)}{M_x+1}$
On peut alors prendre la partie entière de $\frac{i(M+1)}{M_x+1}$ pour trouver notre k.

3.2 Description des classes

3.2.1 Classe Points

Notre classe points contient nos coordonnées x, y, z et un tableau de pointeur double qui représente la température en trois dimensions. Le constructeur prend

en paramètre le nombre de points (Mx, My, Mz) pour chaque coordonnées et alloue un tableau de pointeur temperature de taille Mx+1 dont chaque coordonnée contient un tableau de taille (My +1)*(Mz+1). Cette classe permet en plus d'accéder au tableau en lecteur/écriture, grâce à une surcharge de l'opérateur () prenant en argument 3 entier (i,j,k).

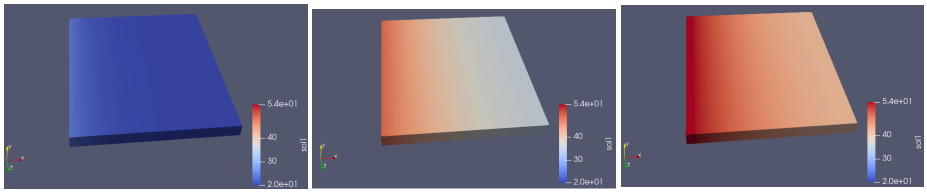
3.2.2 Classe visualisation3D

Cette classe contient 8 attributs : M la taille du maillage sur x, N le temps, (Mx, My, Mz) notre maillage 3D, T_3D notre vecteur solution qui est un pointeur sur la classe Points, syst qui est un pointeur sur la classe Nonstationnaire. Le pas de temps données à syst est de 1. On va utiliser la méthode uneiteration(), pour pouvoir faire une itération à une itération, à la place de plusieurs d'un coup.
Le constructeur de la classe est donné par `visualisation3D(int M, int temps, int Mx, int My, int Mz, bool phiconst = true)`

La première méthode importante de cette classe est `void calculTijk()` qui calcul les coefficients de notre solution 3D, stockés dans le vecteur T_3D, en utilisant l'algorithme 2 Calcul de $\hat{T}_i$ décrit en 3.1 .

La seconde méthode est `void save_vtk()` qui va nous permettre de visualiser notre solution sur paraview. Elle génère N fichier, où N est le temps choisi. Chaque nom de fichier est "solution3D" auquel on ajoute le numéro correspondant au temps enregistré. A chaque itération de la boucle, elle calcule la solution au temps i+1, en appelant la méthode uneiteration() de la classe NonStationnaire sur l'attribut syst.

3.3 Analyse des résultats



Choisissons N = 150. Avec un flux de chaleur constant, on voit évoluer par vague uniforme les différentes couleurs sur notre pavé allant du bleu au rouge. Au début notre pavé est complètement bleu (image 1), les température sont vers les 20 °C. Puis, on se trouve avec un dégrader de couleur allant du rouge vers le bleu (image 2), les températures prennent des valeurs intermédiaires. Vers la fin, de notre simulation, notre pavé a des teintes rouge-orangé, les températures y sont entre 54 et 40 °C.

Considérons maintenant que ce flux est non constant. Lors de son activation les températures évoluent de la même manière que lorsque le flux est constant.

Mais, lors de sa désactivation, les couleurs reviennent à des couleurs plus froides (blanc, bleu) assez rapidement.

4 Le programme principale

Le programme se compile avec g++, et lors de l'exécution on peut ajouter un fichier .cfg ressemblant à celui du sujet, pour initialiser les paramètres. Ce fichier est lu dans le main.