LEFEBVRE
Maxime
GMC 01

MN41 : Résolution de systèmes 1D

Table des matières

Ce rapport a pour vocation de détailler la démarche suivie dans la réalisation d'un programme en langage C. Ce programme nous permettra de résoudre des systèmes physiques 1D discrétisés. Nous aborderons donc dans ce rapport les enjeux du projet avant de décrire les différentes étapes de résolution du problèmes posé. Nous finirons par comparer les résultats obtenus avec les résultats théoriques dans trois cas distincts.

I. Description du problème

Dans le cadre de sa profession, un ingénieur doit aujourd'hui être capable de trouver une solution aux problèmes qu'il rencontre et ce quelque soit leur type. Il est ainsi amené à étudier, de façon plus ou moins poussée, des systèmes physiques complexes. Cependant, ces problèmes ne sont pas toujours solvables de façon directe. L'ingénieur se doit donc de simplifier le problème en le modélisant par un système plus simple reproduisant les aspects ou comportements principaux de l'original. Le but de l'UV MN41 est de comprendre la mise en place et les enjeux de la modélisation de ces systèmes physiques complexes.

Dans ce projet nous nous intéresserons à la discrétisation de systèmes 1D, c'est-à-dire des systèmes ne présentant qu'une seule variables d'évolution pour chaque élément qui le compose (par exemple le potentiel électrique entre deux résistances). Ici, nous nous pencherons notamment sur la résolution de problèmes liés aux domaines mécanique, thermique et électrique.

Avant de pouvoir modéliser ces différents problèmes, attachons nous à décrire leur loi de comportement ainsi que toutes les données dont nous auront besoin. C'est à partir de cette loi de comportement, appliquées aux éléments du système, que nous pourrons discrétiser le problème. Les résultats sont donnés dans le tableau suivant.

Type de problème	Loi de comportement	Variables	Matrice élémentaire
Mécanique (chariot-ressort)	$F = k \cdot \Delta U$	F : force externe appliquée k : raideur du ressort ΔU : déplacement du ressort	$\begin{pmatrix} F_{ij} \\ F_{ji} \end{pmatrix} = K_m \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$
Électrique (résistances en série)	$I = \frac{V}{R}$	I : intensité au nœud V : potentiel électrique R : résistance de l'élément	$\begin{pmatrix} I_{ij} \\ I_{ji} \end{pmatrix} = \frac{1}{R_m} \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$
Thermique (couches d'isolation)	$Q = \frac{T}{R}$	Q : flux thermique T : température au nœud R : résistance de l'élément	$\begin{pmatrix} Q_{ij} \\ Q_{ji} \end{pmatrix} = \frac{1}{R_m} \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Figure 1 : description du comportement des système 1D étudiés

Les différentes matrices élémentaires sont créés à partir d'un tableau de connectivité qui décrit l'agencement de tous les éléments du système par rapport aux nœud existants. L'objectif de ce projet est donc d'assembler les matrices élémentaires de tous les éléments afin de créer une matrice décrivant de façon discrète le système étudié. Il nous faudra ensuite résoudre le problème proposé en appliquant les conditions limites et les éventuelles contraintes qui lui sont liées.

NB : La matrice globale étant symétrique, il est impossible de résoudre le système sans conditions limites.

II. Données du système

Afin de répondre aux objectifs de ce projet, il a été décidé que seul le traitement des informations serait automatique, c'est-à-dire que l'on demandera à l'utilisateur de fournir les données du problèmes. Pour cela, nous avons choisi de récupérer lesdites données sous le forme de deux fichiers textes : *DONNEES.txt* et *CONTRAINTES.txt*. Ces fichiers contiennent :

DONNEES.txt :

- Le nombre d'éléments et de nœuds du système,
- Le tableau de connectivité du système,
- Les constantes multiplicatives (résistance ou raideur) des différents éléments,
- Les forces externes s'appliquant sur chaque éléments,
- Les conditions limites à appliquer (nombre, position et valeur)

CONTRAINTES.txt :

- Les contraintes dont on doit tenir compte (nombre et valeur)

Pour permettre de traiter plusieurs problèmes consécutivement et conserver différents jeux de données, les deux fichiers textes seront placés dans un dossier (au nom libre) directement dans le dossier racine du projet. Lorsqu'il sera demandé à l'utilisateur, le chemin d'accès de ces fichiers sera donc : **../nom_du_dossier/**.

NB : Le choix de l'emplacement des fichiers textes est en fait libre. Mais pour simplifier la prise en main du programme il est recommandé de ne pas le modifier, au risque de devoir changer le chemin d'accès à fournir lors du démarrage du programme. Les deux fichiers devront néanmoins toujours être placés dans le même dossier. Ce dossier servira de plus à stocker le fichier de résultats.

Pour finir, on demandera à l'utilisateur de fournir le type de problème qu'il souhaite traiter (cf. partie V. 1)).

III. Algorithme de résolution proposé

Possédant toutes les données nécessaires à la résolution du problème étudié, nous pouvons maintenant proposé une description du fonctionnement de notre programme. De façon simplifiée le programme devra se dérouler de la même façon que celle abordée en TD.

- Étape 1 : Choix du type de système
- Étape 2 : Choix du dossier de données
- Étape 3 : Lecture des fichiers et initialisation des variables
- Étape 4 : Création de la matrice globale à partir du tableau de connectivités
- Étape 5 : Réduction de la matrice (application des conditions limites)
- Étape 6 : Résolution du système
- Étape 7 : Identification et application des contraintes
- Étape 8 : Résolution finale du système (si des contraintes existent)
- Étape 9 : Affichage des résultats
- Étape 10 : Sauvegarde des résultats dans le dossier.

Remarquons que nous ne définissons pas les matrices élémentaires du système. En effet par soucis de réduire l'espace mémoire alloué au programme, nous avons souhaité n'allouer que la matrice globale et la remplir directement à l'aide de la table de connectivités. Cela ne change cependant rien au fonctionnement

du programme.

Dans la suite du rapport, nous définirons les différentes fonctions mises en place à chaque étape.

IV. Initialisation

Après avoir donné les grandes lignes de l'algorithme utilisé pour notre programme, tachons maintenant de développer un peu plus la façons dont il travaille.

1) Allocations des variables

En langage C, il existe deux possibilités pour allouer la mémoire nécessaire au stockage des matrices et des vecteurs : l'allocation à partir d'une constante, vue en TP, ou l'allocation dynamique. La taille des systèmes étudiés n'étant pas fixe (variable donnée dans le fichier de données), nous devrons obligatoirement passer par l'allocation dynamique. Nous avons donc créer deux fonctions, une pour les vecteurs, l'autre pour les matrices. Ces fonctions prendront en argument la taille de l'espace mémoire à allouer pour stocker les données.

Pour un vecteur, le principe est simple : pour chaque valeur nous allons allouer une case mémoire de la taille souhaitée (4 octets pour un `float`). Nous aurons donc besoin d'un espace de 4N octets en mémoire pour pouvoir stocker un vecteur de `float` de taille N.

```
float* allocateVector(int nLine){
        float* vect = (float*)malloc(nLine*sizeof(float));     //allocation valeurs
        return vect;
}
```

Pour une matrice, on alloue en fait plusieurs vecteurs : on alloue d'abord un vecteur qui représentera la première ligne de notre matrice puis pour chacune de ses cases on alloue un nouveau vecteur. En d'autres termes, on crée les différentes colonnes à partir de l'espace réservé pour la première ligne. On insère pour cela une seconde boucle dans la première. Cette dernière va nous permettre de créer les cases manquantes.

```
float** allocateMatrix(int nLine, int nCol){
        float** mat = (float**)malloc(nLine*sizeof(float*));   //allocation 1e ligne
        for (int i = 0; i < nLine; i++ ) {
               mat[i] = (float*)malloc(nCol*sizeof(float));    //allocation colonnes
        }
        return mat;
}
```

Ces deux fonctions retournent un pointeur vers la zone mémoire allouée. On récupère ainsi un espace que nous allons pouvoir remplir avec les valeurs des données contenues dans les fichiers utilisateurs.

NB : Certaines matrices et vecteurs sont définis à l'aide de structures de type `SYSTEMS`. Les structures sont en fait des variables qui contiennent d'autres variables de type différents. Nous utiliserons ici,

```
typedef struct{
    float** matrix;
    float* vector;
    int size;
}SYSTEMS;
```

Chaque système S contiendra donc une matrice (`S.matrix`), un second membre (`S.vector`) et une taille (`S.size`). On considère dans notre cas des systèmes dont les matrices sont carrés, c'est pourquoi nous n'avons pas besoin de définir le nombre de ligne ET le nombre de colonnes.

2) *Lecture des fichiers*

Pour remplir ces éléments, nous devons donc récupérer les données que l'utilisateur a stocké dans les fichiers *DONNEES.txt* et *CONTRAINTES.txt*. Pour cela intéressons nous d'abord à la structure de ces fichiers :

```
Nombre de noeuds  :     5

-----------------------------------

Connectivités :

1       2       2       3       4
2       4       3       4       5

-----------------------------------

Vecteur des valeurs de K ou de R :

1       2       3       4       5
```

Figure 2 : structure du fichier pour une valeur, une matrice et un vecteur

Nous pouvons constater que les différents éléments (vecteurs, matrices et valeurs simples) sont précédés d'un texte informatif se terminant par deux points. C'est grâce à cette structure particulière que nous pouvons facilement lire les données. En effet, avant de pouvoir lire les données numériques, nous devons positionner le curseur de lecture au bon endroit. Pour cela, à l'aide de la fonction `void readVoid(FILE* fp)` nous allons lire les caractères un par un jusqu'à rencontrer « : ». Après avoir lu ce caractère, nous sommes sûrs d'être devant la prochaine valeur numérique.

Il nous suffit alors d'utiliser la fonction adéquate (`void readMatrix()`, `void readVector()` ou `int readValue()`) afin de récupérer les valeurs souhaitées. Les deux premières fonctions prennent pour arguments la taille des éléments à remplir (N x M pour une matrice) et nécessitent un pointeur vers ledit élément. La fonction `int readValue()` ne prend pas d'argument et renvoie directement la valeur lue dans le fichier.

NB : Si nous ne passions pas les matrices ou les vecteurs par pointeur, nous n'aurions pas la possibilité des les remplir car il est impossible de retourner un tableau dans une fonction. Les matrices seraient donc inchangées.

3) *Gestion des erreurs*

Nous avons maintenant récupéré toutes les données issues des fichiers utilisateurs. Néanmoins, nous ne savons pas si ces données sont cohérentes, non seulement d'un point de vue algébrique mais aussi entre elles. Nous avons donc mis en place un certain nombre de vérifications tout au long du programme afin de rendre le programme robuste (insensible aux erreurs). En effet, nous souhaitons que le programme ne puisse pas s'arrêter de façon impromptue en cas d'erreur de l'utilisateur. On crée alors les vérifications suivantes sous forme de conditions ou de fonctions propres :

- **Lecture des fichiers :** Si le chemin des fichiers de données est faux ou si les fichiers ne se sont pas ouverts de façon correcte, le programme s'arrête (dossier ou *DONNEES.txt*) ou continue de manière tronquée (*CONTRAINTES.txt*).
- **Table de connectivités :** On vérifie que le tableau de connectivités ne contient pas de nœuds inexistants ou de liaisons entre un nœud et lui même. En cas d'erreur, le programme s'arrête → `int checkConnectivity()`
- **Résistance des éléments :** On vérifie que les valeurs des résistances ne sont pas nulles ou négatives. Si c'est le cas, le programme s'arrête → `int checkKVector()`
- **Conditions :** Connaissant la position des conditions, on vérifie que le vecteur est correct. Une condition ne peut pas exister si la valeur n'est pas censée être connue. En cas d'erreur, les conditions limites ne sont pas appliquées → `int checkConditions()`
- **Contraintes :** On vérifie qu'il n'existe aucune contraintes dans le cas où il existe une condition limite sur le nœud. En d'autres termes, la contrainte doit être nulle si la condition limite existe. En cas d'erreur, le programme n'applique pas les contraintes → `int checkConstraintsCoherence()`

Toutes ces vérifications nous permettent de ne traiter le problème que si les fichiers de données sont cohérents. Si tout est correct, les fonctions retournent la valeur 1. Dans le cas contraire, le programme peut suivre son déroulement normal ou s'arrêter selon l'erreur reconnue.

Nous avons d'autre part créé d'autres fonctions de vérifications qui interviennent lors de la résolution du système. Elles nous permettent en effet de vérifier que tout se passe correctement lors de certaines étapes intermédiaires. Nous les aborderons ci-après dans la description pas à pas du programme.

V. Le programme pas à pas

Dans cette partie, nous nous attacherons à détailler le fonctionnement de chaque sous-partie du programme. Pour cela nous rappellerons les différents mécanismes de calculs mais aussi de vérification mis en jeu.

NB : Toujours dans un soucis de gérer correctement l'espace mémoire, les opération de lecture de fichiers n'ont lieues que si elles sont nécessaires. Elles sont donc utilisées aux endroits auxquels nous nécessitons de récupérer des données (on ne lit pas tout le fichier dès le début). De même nous n'allouerons les matrices et vecteurs que lorsque que nous en aurons besoin.

1) *Menu : nature du problème et dossiers de données*

Lors du lancement du programme un menu apparaît. Ce menu permet d'initialiser la recherche de solution en deux étapes. Dans un premier temps l'utilisateur doit sélectionner la nature du problème en rentrant un

choix (a, b ou c). Le programme appliquera plus tard les opérations nécessaires à la prise en compte de ce choix :

- Si le choix est a (cas mécanique) alors le programme se déroule normalement.
- Sinon le vecteur des résistance est inversé et le programme continue (cas thermique ou électrique).

Notons que le choix effectué est vérifié grâce à une boucle `while`, c'est-à-dire que tant que le choix n'est pas dans les options proposé le programme attend. Cela nous permet ainsi d'être sûrs que l'utilisateur n'a pas tapé une valeur au hasard. Cependant, n'ayant aucune information sur le type réel de problème implémenté dans le dossier de données, il nous est impossible de vérifier que ce choix est cohérent avec les problème traité. Cette situation ne générera pas d'erreur critique, mais les résultats de calculs seront faux. L'utilisateur devra donc faire attention à ne pas commettre d'erreur à cette étape. Nous noterons également que pour éviter tout problème de dépassement de pile (stack overflow) sur la variable de choix, nous vidons le cache de l'entrée clavier (`stdin`) avant chaque saisie.

Dans un deuxième temps, l'utilisateur spécifie le dossier contenant les fichiers de données et de contraintes. Son choix est soumis à une première vérification. En effet, si le traitement peut s'effectuer sans appliquer de contraintes, il est impossible de résoudre un système qui n'existe pas. Donc si le dossier spécifié n'existe pas ou si le fichier de données n'est pas correctement ouvert, le programme retourne une erreur et s'arrête. Si toutes les informations sont correctes, le programme commence la lecture des données et le traitement de celles-ci.

2) *Table de connectivités et matrice globale*

Après avoir initialisé le système, nous commençons la lecture du fichier de données afin d'obtenir les spécifications du systèmes : nombre d'éléments, nombre de nœuds et leur agencement les uns par rapport aux autres. Nous obtenons alors deux entiers (`nElement` et `global.size`) et un tableau (`tabConnec`) de longueur `global.size` et de hauteur 2. Ce tableau est le tableau de connectivités du système. Il nous permettra de remplir la matrice globale à partir de la liste des liaisons entre les nœuds.

Lors de la lecture de ce tableau, on fait varier i de 0 à `global.size` et j entre 0 et 1, avec i correspondant au numéro de l'élément attaché entre les nœuds `tabConnec[i][0]` et `tabConnec[i][1]`. C'est ce i que nous pourrons ensuite utiliser pour obtenir la valeur de résistance ou de raideur de l'élément. Mais avant cela le programme vérifie que la table de connectivités est cohérente (cf. IV.3)). Si la fonction de vérification retourne 1, alors le programme s'exécute normalement.

Le programme vient ensuite lire deux nouveaux vecteurs : les résistances (`kVector`), puis le vecteur des forces externes appliquées au système (`global.vector`). Après avoir vérifié que les valeurs étaient bien supérieures à zéro, nous faisons intervenir ici le choix de la nature du problème : si le problème est un problème autre que mécanique, les valeurs `kVector[i]` sont inversées à l'aide de la fonction `void flipVector()`. Sinon, rien ne change.

Nous avons donc maintenant toutes les informations nécessaires afin de construire la matrice globale de notre système. Pour cela nous allons lire en parallèle la table de connectivités ainsi que le vecteur des K : pour le $i^{ème}$ élément on connaît ainsi la valeur de résistance et les nœuds auxquels il est rattaché. La matrice global `global.matrix` étant de taille `global.size x global.size` est d'abord initialisées à zéro, puis il nous suffit de récupérer le numéro des nœuds et ainsi d'ajouter à la valeur précédente de `global.matrix` la valeur de résistance correspondante. Cela revient en fait à définir implicitement les matrices élémentaires et d'injecter leurs composantes à la matrice globale initialement nulle.

On obtient ainsi la matrice et le vecteur sur lesquels nous allons appliquer nos conditions limites.

NB : On ne vérifie pas que les matrice est symétrique avec une diagonale positive. En effet, la vérification du vecteur des résistances ainsi que le fait d'avoir coder nous même la fonction d'écriture de la matrice globale, nous permet d'être assuré que la matrice est correcte.

3) Matrice réduite

La matrice globale créée, nous allons pouvoir commencer à résoudre le système nouvellement créé. Pour cela nous devons d'abord récupérer les valeurs des conditions limites à appliquer. Nous continuons donc la lecture de notre fichier de données et allouons les vecteurs nécessaires : `knownConditions` et `finalResults`. Nous obtenons ainsi le nombre de conditions (`nCondition`), leur position et leur valeur. Deux cas se présentent alors à nous :

- Le nombre de conditions est nul : le système global est considéré comme système réduit.
- Il existe des conditions et elles sont correctes (cf. IV.3)) : on calcule le système réduit.

Rappelons à ce stade que la non-existence de conditions limites est aberrante. En effet, si il n'y a aucune condition limite, le système n'est pas solvable. Il existe dans ce cas plus d'inconnues que d'équations dans le système car au moins une ligne est la combinaison linéaire des autres.

Une fois les vérifications faites, il est possible de calculer le système réduit après avoir alloué la matrice (`reduced.matrix`) ainsi que les vecteurs de forces externes (`reduced.vector`) réduits. La taille de ces deux éléments est fonction du nombre de conditions : `reduced.size = global.size-nCondition`.

Nous utilisons ensuite la fonction `applyCondition()` qui va calculer les deux éléments précédents à l'aide de la matrice globale et du vecteur de forces extérieures initial. Nous aurons ici besoin de quatre curseurs : deux pour le système global (i et j) et deux pour le système réduit (r et c). Le principe est alors le suivant,

```
r = 0
Pour 0 < i < n : Lecture des positions des conditions limites
|      c = 0
|      Si la ligne doit être supprimée, ne rien faire
|      Sinon : initialiser r^ième valeur du vecteur réduit
|      |      Pour 0 < j < n :
|      |      |      Si condition sur j^ème valeur : affectée aux forces réduites
|      |      |      Sinon
|      |      |      |      La valeur courante appartient à la matrice réduite R_r,c
|      |      |      |      Incrémentation de c : colonne réduite suivante
|      |      |      Fin.si
|      |      Fin.pour
|      |      Incrémentation de r : ligne réduite suivante
|      Fin.si
Fin.pour
```

Ce mécanisme nous permet ainsi de lire une à une les valeur de la matrice globale et de les traiter de la façon adéquate, c'est à dire de les supprimer ou de les garder selon les valeurs des conditions appliquées. La matrice et le vecteur obtenu représente le système que nous allons maintenant résoudre pour calculer les résultats du problème posé.

4) *Résolution du système réduit*

Lors des premières séances de TP, nous avons pu aborder trois manières différentes de résoudre une système matriciel : la méthode de Thomas, la méthode du pivot de Gauss et la méthode LU. Nous avons donc décider de laisser à l'utilisateur le choix de la méthode à appliquer. Pour cela, nous mettons une nouvelle fois le programme en pause dans l'attente d'un choix correct (a, b et éventuellement c). Ce choix sera plus tard conservé pour toutes les étapes de résolution (cf. Application des contraintes). Nous ferons néanmoins attention à ce que la méthode puisse être appliquée notamment car la méthode de Thomas nécessite que la matrice soit tribande (trois bandes non-nulles). Nous nous attacherons par conséquent à vérifier la forme de la matrice avant de proposer cette solution.

Il est donc en théorie possible d'utiliser la méthode que l'on souhaite afin de mener à bien l'étude. Nous recommandons cependant dans l'idéal l'utilisation de la méthode du Pivot de Gauss qui offre de meilleurs résultats d'un point de vue informatique et ce pour les raisons suivantes.

- La méthode de Thomas fonctionne très bien mais seulement dans le cas ou la matrice est tribande, c'est-à-dire que les nœuds liés à un élément ne peuvent être que consécutifs, ce qui n'est pas le cas dans tous les systèmes. La vérification de la matrice surcharge donc le programme.
- Pour la méthode LU ce n'est pas la forme de la matrice qui pose problème mais plutôt l'espace mémoire nécessaire au bon fonctionnement de la méthode. En effet, en plus des matrices réduites, il faut créer des matrices temporaires L, U et Y. Ce problème va à l'encontre du choix que nous souhaitons faire d'optimiser l'espace mémoire utilisé.
- La méthode du pivot de Gauss semble donc la plus adaptée pour répondre à nos besoins car elle permet de résoudre tous les types de problème tout en n'augmentant pas l'espace mémoire nécessaire de façon trop importante.

L'utilisateur restera néanmoins seul décisionnaire de la méthode à appliquer au travers de la fonction `void solveSystem()`.

NB : Lors de la descente de la matrice dans la méthode de Gauss, nous aurions pu vérifier que le pivot n'est pas nul. Mais en pratique si le pivot est nul, alors le nœud correspondant n'existe pas et il n'y donc pas d'autres valeurs à annuler pour ce pivot.

5) *Récupération des résultats*

Pour récupérer les résultats globaux dans le vecteur X, on réécrit le vecteur de résultats initial de façon à conserver les conditions limites tout en ajoutant les résultats bruts. La méthode est la suivante :

- Si il existe une condition limite pour la $i^{ème}$ valeur, alors X_i prend la valeur de la condition,
- Sinon, X_i prend la valeur du prochain résultat brut et on passe à la valeur brute suivante.

Soit pour l'exemple suivant,

Position des conditions	Conditions limites	Résultats bruts	X
$\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$	$\begin{pmatrix}20\\0\\0\\15\end{pmatrix}$	$\begin{pmatrix}5\\7\end{pmatrix}$	$\begin{pmatrix}20\\5\\7\\15\end{pmatrix}$

Figure 3 : Exemple de récupération des résultats finaux

Le vecteur X des résultats finaux est donc bien la combinaison de deux vecteurs : celui des résultats bruts et celui des conditions limites. Ces derniers nous permettent de clore la première partie du programme qui consistait à résoudre le système proposé.

6) *Application des contraintes*

Après avoir obtenu un premier résultat de notre système, nous devons vérifier que ce dernier est libre d'atteindre l'état dans lequel il doit se trouver. En d'autres termes nous devons vérifier qu'il n'existe pas d'autres contraintes sur le système global, mais aussi savoir si ces contraintes s'appliquent.

Tout d'abord, nous nous assurons que le fichier de contraintes a bien été ouvert à l'aide d'un `if`. Si c'est le cas, alors on lit le nombre de contraintes (`nConstraint`) ainsi que leurs valeurs (`constraints`) puis on vérifie qu'elles sont cohérentes avec le vecteur des conditions limites (cf. IV.3)).

Dans le cas où tout est correct, le programme se met en attente d'un choix utilisateur : doit-on ou non appliquer les contraintes ? Ce choix s'effectue seulement par rapport au nombre de contraintes car on ne sait pas encore si des contraintes s'appliquent réellement. On sait juste qu'il existe de potentielles contraintes. Ici, deux fins possibles :

- « non »: on ne tient pas compte des contraintes et on affiche le résultat final.
- « oui » : on regarde quelles sont les contraintes et on les applique si besoin.

Il nous reste donc à savoir si les contraintes doivent réellement être appliquées à notre système.

Pour cela, nous calculons pour tous les éléments l'inverse du temps mis par ce dernier pour atteindre l'état de contrainte. Nous conservons alors l'index de position de l'élément pour lequel la contrainte est la plus forte. La fonction `int checkConstraints()` nous renvoie alors deux valeurs possibles : soit le numéro de l'élément sur lequel la contrainte s'applique, soit -1 si aucune contrainte n'a été trouvée.

Une fois cette vérification terminée, on modifie les vecteurs de conditions ainsi que le vecteur des forces extérieures. En effet, si une contrainte s'applique on peut considérer que le système a une condition limite nouvelle. On incrémente donc le nombre de conditions limites, on remplace le 0 de l'élément contraint par un 1 et on modifie également la force qui lui est appliquée dans le second membre.

On réutilise ensuite les trois étapes précédentes afin de calculer le nouveau système réduit et le résoudre. Ici le choix de la méthode n'a pas lieu : on reprend la méthode sélectionnée précédemment afin de calculer le résultat du système.

On termine par regarder si il existe une autre contrainte sur le résultat obtenue. Si oui, on recommence la recherche ainsi que la modification du vecteur des conditions. Finalement on résout une dernière fois le système.

Si aucune autre contrainte n'existe, on affiche directement le résultat final.

7) *Affichage du résultat final et fin du programme*

Toutes les étapes s'étant déroulées correctement, on affiche le résultat puis on vérifie qu'il est cohérent. Pour cela, on utilise la fonction `int checkResult()`. Cette fonction prend en argument la matrice globale et le résultat final de notre système et multiplie les deux en stockant le résultat dans un vecteur temporaire.

On réalise alors la somme des valeurs de ce vecteur.

Si tout est correct, la valeur de cette somme devrait être nulle (équilibre du système). Cependant, les `float` sont des variables qui ont une tolérance de précision, c'est-à-dire qu'elles ne retournent pas la valeur exacte qui leur a été affecté. Prenons par exemple un test simple :

```
float t = 0.1;

if(t == 0.1) printf("TRUE");
else printf("FALSE");
```

Ce programme affichera toujours `FALSE` car la valeur stockée dans `t` n'est pas exactement 0.1 : elle a été affectée avec un résidu liée à son type. Ce phénomène entraîne donc l'accumulation d'erreur de calcul tout au long de la résolution du système.

Pour remédier à cela nous ne vérifions donc pas que la somme finale est nulle mais plutôt qu'elle est inférieure à un seuil de tolérance. Nous choisirons ici un tolérance au $10000^{ième}$, ce qui est largement suffisant pour modéliser notre système. En effet, il est rare que nous donnions plus de deux chiffres décimaux lors de la résolution manuelle.

Pour avoir une tolérance plus faible, nous aurions pu utiliser des variables de type `double float` qui ont une précision plus élevée. Cependant cela voudrait dire utiliser deux fois plus d'espace mémoire pour obtenir une précision dont nous n'avons pas forcément besoin. Nous avons donc conservés nos variables en type `float`.

Si le résultat est correct un message indiquant que l'équilibre a été atteint s'affiche.

8) Sauvegarde des résultats

Nous terminons le programme par la sauvegarde des données calculées. Grâce à la fonction `void writeResults()`, nous écrivons la matrice globale, le second membre ainsi que les résultats dans le fichier *RESULTATS.txt* placé dans le dossier spécifié par l'utilisateur (cf II.). Cette fonction est en fait une succession de *fput()* qui nous permet d'écrire dans le fichier des chaînes de caractères ou des nombres. Nous sauvegardons de plus la réponse de l'utilisateur quant aux contraintes ce qui lui permettra de savoir si les résultats sauvegardés prennent en compte les contraintes ou non.

NB : Si le fichier n'existe pas il est créé puis rempli. Si il existe, le fichier est d'abord vidé puis re-rempli avec les valeurs obtenues.

Lorsque toutes les opérations sont terminées, les espaces mémoires précédemment alloués sont libérés (gestion au cas par cas) puis tous les fichiers de données sont fermés. Pour finir, le programme se met une nouvelle fois en attente : l'utilisateur souhaite t-il relancer le programme avec un autre jeu de données ? Si oui, l'écran est effacé à l'aide de la fonction `system("cls")` et le menu s'affiche de nouveau. Si non, la programme s'arrête et conserve les résultats à l'écran.

9) Algorithme final

Nous l'avons vu tout au long du rapport, l'algorithme proposé a été modifié petit à petit pour répondre aux

besoins d'optimisation et de robustesse de notre programme. Nous proposons donc dans la suite un algorithme qui résume la forme finale de l'application.

```
choix = "oui";
while(choix == "oui"){

    effacer écran
    afficher menu

    if(fichier de données ouvert) {
        lire et afficher nombre d'éléments et de nœuds
        lire et afficher table de connectivités
        vérifier table de connectivités

        if(connectivités OK) {
            lire vecteur des résistances
            afficher vecteur des résistances
            lire et afficher vecteur des forces extérieures

            vérifier vecteur des résistance

            if(résistances OK){
                if(problème non mécanique){
                    inverser vecteur des résistances
                }

                remplir et afficher matrice globale
                lire les vecteurs de conditions limites
                vérifier conditions limites

                if((il existe des conditions)&&(conditions OK)) {
                    récupération du système réduit
                }
                else {
                    système global devient le système réduit
                }

                résolution du système grâce à la méthode choisie
                récupération des résultats globaux

                if(fichier de contraintes ouvert){
                    lecture des contraintes
                    vérification des contraintes

                    if(il existe des contraintes){
                        if(les contraintes sont correctes){
                            attente choix utilisateur : appliquer les contraintes ?
                                if(choix == "oui"){
                                for(j = 0 au nombre de contraintes){
                                    récupération contrainte à appliquer en premier

                                    if(si contrainte effective){
                                        modification conditions limites
                                        récupération système réduit
                                        résolution du système grâce à la méthode choisie
                                        récupération des résultats globaux
                                    }
                                }
                            }
                        }
```

```
                    else{
                        afficher erreur sur les contraintes
                    }
                }
                libérer vecteur de contraintes
                fermer fichier de contraintes
            }
            else {
                erreur sur fichier de contraintes
            }
            affichage et vérification du résultat final
            libérer système réduit et conditions limites
        }
        else{
            erreur sur les résistances
        }
        libérer système global
    }
    else {
        erreur sur la table de connectivités
    }
    libérer table de connectivités
    fermer fichier de données
}
else {
    erreur sur fichier de données
}
attendre choix utilisateur : recommencer ?
}
```

VI. Diverses applications

Appliquons pour terminer notre programme à différents système proposés dans le projet ou en examen. Pour montrer que nos résultats pratiques sont corrects nous commencerons par calculer de façon succincte les résultats théoriques pour le premier cas puis nous donnerons le résultat pour tous les autres cas.

1) *Système mécanique du projet, cas 4*

Schéma du système

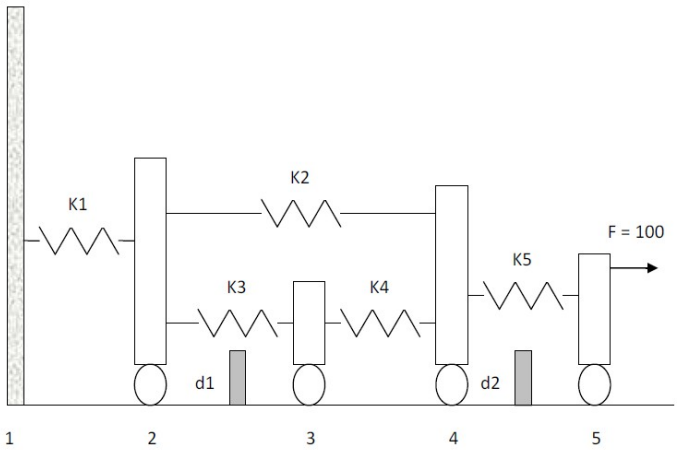

Figure 4 : Schéma du système mécanique étudié

Données

Il y a 5 éléments (ressorts) et 5 nœuds (1 mur et 4 chariots) .

Chaque élément K_i répond à la loi de comportement : $F = K_i \cdot \Delta U$
avec $K_1 = 1$, $K_2 = 2$, $K_3 = 3$, $K_4 = 4$ et $K_5 = 5$.

Le système est alors définit par :

| Vecteur des déplacements U : $(0, U_2, U_3, U_4, U_5)$
| Vecteur des forces extérieures F : $(R_1, 0, 0, 0, 100)$

Table de connectivités

Élément	K_1	K_2	K_3	K_4	K_5
i	1	2	2	3	4
j	2	4	3	4	5

Matrices élémentaires

Élément 1 : $\begin{pmatrix} F_{12} \\ F_{21} \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Élément 2 : $\begin{pmatrix} F_{24} \\ F_{42} \end{pmatrix} = 2 \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Élément 3 : $\begin{pmatrix} F_{23} \\ F_{32} \end{pmatrix} = 3 \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Élément 4 : $\begin{pmatrix} F_{34} \\ F_{43} \end{pmatrix} = 4 \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Élément 5 : $\begin{pmatrix} F_{45} \\ F_{54} \end{pmatrix} = 5 \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Matrice globale

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 6 & -3 & -2 & 0 \\ 0 & -3 & 7 & -4 & 0 \\ 0 & -2 & -4 & 11 & -5 \\ 0 & 0 & 0 & -5 & 5 \end{pmatrix} \begin{pmatrix} 0 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix} = \begin{pmatrix} R_1 \\ 0 \\ 0 \\ 0 \\ 100 \end{pmatrix}$$

Conditions limites

$U_1 = 0$

Matrice réduite

Suppression de la première ligne et passage de la colonne correspondante dans le second membre.

$$\begin{pmatrix} 6 & -3 & -2 & 0 \\ -3 & 7 & -4 & 0 \\ -2 & -4 & 11 & -5 \\ 0 & 0 & -5 & 5 \end{pmatrix}\begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 0 \\ 100 \end{pmatrix}$$

$R_1 = U_1 - U_2 = - U_2$

Résolution par le pivot de Gauss

Système triangularisé

$$\begin{pmatrix} 6 & -3 & -2 & 0 \\ 0 & 5{,}5 & -5 & 0 \\ 0 & 00 & 5{,}79 & -5 \\ 0 & 0 & 0 & 0{,}68 \end{pmatrix}\begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 0 \\ 100 \end{pmatrix} \Leftrightarrow \begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 100 \\ 115{,}38 \\ 126{,}92 \\ 146{,}92 \end{pmatrix}$$

Application des contraintes

Il existe des contraintes sur notre système telles que $U_{2\max} = 20\text{m}$ et $U_{4\max} = 30\text{m}$. Regardons par conséquent quelle condition s'applique en premier. Pour cela, comparons les rapport entre la distance parcourue et la distance maximale pour les chariots.

Ici pour le chariot 2, la contrainte est de $\frac{U_2}{U_{2\max}} = \frac{100}{20} = 5$

Pour le chariot 4 on trouve un contrainte de $\frac{U_4}{U_{4\max}} = \frac{126{,}92}{30} = 4{,}23$

La contrainte est donc plus forte sur le chariot 4. Nous allons donc reprendre notre étude en prenant en compte ce résultat. Ainsi nous trouvons le système réduit suivant :

$$\begin{pmatrix} 7 & -4 & 0 \\ -4 & 11 & -5 \\ 0 & -5 & 5 \end{pmatrix}\begin{pmatrix} U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 60 \\ 40 \\ 100 \end{pmatrix}$$

$R_2 = - U_1 + 6U_2 - 3U_3 - 2U_4$

Ce qui donne,

$$\begin{pmatrix} 7 & -4 & 0 \\ 0 & 8{,}71 & -5 \\ 0 & 0 & 2{,}13 \end{pmatrix}\begin{pmatrix} U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 60 \\ 74{,}29 \\ 142{,}62 \end{pmatrix} \Leftrightarrow \begin{pmatrix} U_3 \\ U_4 \\ U_5 \end{pmatrix}=\begin{pmatrix} 35{,}38 \\ 46{,}92 \\ 66{,}92 \end{pmatrix}$$

La contrainte sur le chariot 1 existe toujours on l'applique donc.

Ce qui donne finalement, $\vec{U}=\begin{pmatrix} 0 \\ 20 \\ 25{,}71 \\ 30 \\ 50 \end{pmatrix}$ et $\vec{F}=\begin{pmatrix} -20 \\ -17{,}14 \\ 0 \\ -62{,}86 \\ 100 \end{pmatrix}$

Comparaison avec la pratique

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      0.00              F =      -20.00
        20.00                       -17.14
        25.71                       -0.00
        30.00                       -62.86
        50.00                       100.00
```

<u>Figure 5 :</u> Résultats pratiques sur le système mécanique, cas 4

Les résultats obtenus sont cohérents avec les résultats attendus.

2) *<u>Système mécanique du projet, cas 1</u>*

Pas de contraintes. Utilisation de la méthode du pivot de Gauss.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      0.00              F =      -100.00
        100.00                      -0.00
        115.38                      0.00
        126.92                      0.00
        146.92                      100.00
```

<u>Figure 6 :</u> Résultats pratiques sur le système mécanique, cas 1

3) *Système mécanique du projet, cas 2*

Contraintes : d_1 = 80m et d_2 = 20m. Utilisation de la méthode du pivot de Gauss.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      0.00              F =      -15.76
        15.76                       0.00
        18.18                       0.00
        20.00                      -84.24
        40.00                      100.00
```

Figure 7 : Résultats pratiques sur le système mécanique, cas 2

4) *Système mécanique du projet, cas 3*

Contraintes : d_1 = 20m et d_2 = 80m. Utilisation de la méthode du pivot de Gauss.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      0.00              F =      -20.00
        20.00                      -80.00
        35.38                       0.00
        46.92                      -0.00
        66.92                      100.00
```

Figure 8 : Résultats pratiques sur le système mécanique, cas 3

5) *Système électrique du projet, cas 1*

Schéma du système

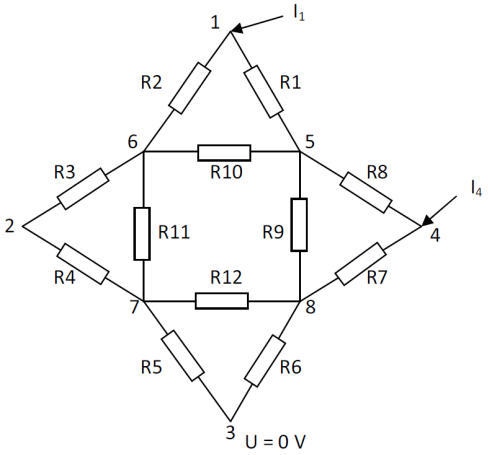

Figure 9 : Schéma du système électrique étudié

Utilisation de la méthode LU.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      266.67           F =      20.00
         133.33                    0.00
         0.00                      -20.00
         133.33                    0.00
         166.67                    -0.00
         166.67                    0.00
         100.00                    -0.00
         100.00                    -0.00
```

Figure 10 : Résultats pratiques sur le système électrique, cas 1

6) *Système électrique du projet, cas 2*

Utilisation de la méthode LU.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      133.33           F =      0.00
         100.00                    0.00
         0.00                      -20.00
         233.33                    20.00
         150.00                    -0.00
         116.67                    0.00
         83.33                     -0.00
         116.67                    -0.00
```

Figure 11 : Résultats pratiques sur le système électrique, cas 1

7) *Système thermique du médian, A2012*

Utilisation de la méthode de Thomas.

```
----------------------
 RESULTAT FINAL
----------------------
-- Vecteur d'etat -----   -- Vecteur des efforts -----
X =      601.00           F =      100.00
         501.00                    -0.00
         301.00                    0.00
         1.00                      -100.00
```

Figure 12 : Résultats pratiques sur le système thermique