Résolution de systèmes d'équations linéaires (méthodes directes)

Valentin PORTAIL et Jérémie VILLEPREUX

6 octobre 2022

Table des matières

Figure 1 – Carl Friedrich Gauss

1 Pré-requis

1.1 Les bibliothèques nécessaires

Ce projet nécessite l'installation des trois bibliothèques standards suivantes :

- `stdio.h`, pour la gestion des affichages et saisie clavier ;
- `stdlib.h`, pour la gestion des allocations dynamiques ;
- `math.h`, pour l'utilisation de la fonction puissances ;

1.2 Comment compiler notre projet ?

Un fichier `Makefile` est également disponible pour compiler le programme. Le compilateur choisi est `gcc` et l'exécutable généré aura pour nom `prog`. Il s'obtient avec la commande :

```
1 $ make
```

Il est également possible de compiler le programme sans utiliser le `Makefile` à disposition avec la commande :

```
1 $ gcc -Wall -Wextra -o prog *.c -lm
```

Remarque 1. La compilation de tous les fichiers sources est nécessaire à l'obtention de l'exécutable.

2 Rappel rapide des méthodes

2.1 Les systèmes d'équations linéaires

Soient deux matrices $A \in \mathbb{R}^{n\times n}$ et $b \in \mathbb{R}^n$. Notre objectif est alors de trouver un vecteur $x \in \mathbb{R}^n$ tel que $Ax = b$.

Pour $i \in [\![1, n]\!]$ et $j \in [\![1, n]\!]$, on pose $a_{ij} \in \mathbb{R}$ le coefficient de A sur la i-ème ligne et la j-ème colonne, $b_i \in \mathbb{R}$ le coefficient de b sur la i-ème ligne et $x_i \in \mathbb{R}$ le coefficient de x sur la i-ème ligne. L'équation $Ax = b$ s'écrit alors sous forme matricielle de la manière suivante :

$$Ax = b \iff \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \iff \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{pmatrix}$$

Cette équation correspond donc à un système d'équations linéaires.

Il existe deux principales méthodes de résolution pour les systèmes $Ax = b$:

Les méthodes directes : On transforme le système pour obtenir des équations plus simples. La solution obtenue sera exacte.

Les méthodes indirectes : On choisit un point initial, puis à l'aide d'algorithmes, on passe d'un point à un autre jusqu'à atteindre une solution proche de la solution exacte.

Dans ce document, nous ne nous concentrerons que sur une méthode de résolution directe : la méthode de Gauss.

2.2 La méthode de Gauss

La méthode de Gauss simple permet d'obtenir une solution exacte du système $Ax = b$ en un nombre fini d'étapes. Cependant, des erreurs au cours de la résolution peuvent affecter les résultats obtenus à la fin.

2.2.1 La triangularisation

L'objectif est de transformer notre système $Ax = b$ en un système $A'x = b'$ où A' est une matrice triangulaire supérieure, c'est-à-dire, une matrice qui vérifie les condition suivantes :

$$\begin{cases} a_{ij} = \lambda \in \mathbb{R} & \text{si } j \geq i \\ a_{ij} = 0 & \text{si } j < i \end{cases}$$

Pour cela, on utilise l'algorithme de triangularisation (donné dans le cours) :

Algorithm 1: Triangularisation

Input : A : tableau[n,n] de flottant ;
n : entier (désignant la taille de A) ;
B : tableau[n,1] de flottant ;

for $k = 1, \dots, n-1$ **do**
  for $i = k+1, \dots, n$ **do**
    // Calcul du pivot
    $\alpha_i^{(k)} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$
    for $j = k, \dots, n$ **do**
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - \alpha_i^{(k)} a_{kj}^{(k)}$
    end for
    $b_i^{(k+1)} = b_i^{(k)} - \alpha_i^{(k)} b_k^{(k)}$
  end for
end for

Remarque 2.
Quelques précisions sur les notations :

— $a_{ik}^{(j)}$ désigne l'élément de la i-ième ligne et k-ième colonne de la matrice A, lors de la j-ième étape de la boucle `for`.
— $b_i^{(k)}$ désigne l'élément de la 1re ligne et i-ième colonne de la matrice B, lors de la k-ème étape de la boucle `for`.

2.2.2 Résolution facile du système

Une fois le système $Ax = b$, transformé avec A une matrice triangulaire supérieure, trouver la matrice x solution du système est simple. On a :

$$x_n = \frac{b_n}{a_{nn}}$$

et

$$\forall i \in \{n-1, n-2, \ldots, 1\}, \quad x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$$

2.2.3 Méthode du pivot maximal : Choix du pivot partiel

Cette méthode permet de gérer au mieux le cas où un pivot est nul ou presque nul (*i.e.* très proche de 0).

En effet, si un élément de la diagonale de notre matrice A est nul, on obtiendra une division par 0 à la ligne 4 (dans l'algorithme de triangularisation). Pour empêcher cela, on a mis en place une méthode dite « *Cas du pivot partiel* ». Elle consiste à parcourir la colonne sur laquelle on a notre pivot nul et de permuter la ligne (où le pivot est nul) avec une ligne inférieure dans la matrice respectant la propriété suivante : il faut que cette ligne permutée ait, dans la même colonne que le pivot nul, le plus grand scalaire en valeur absolue.

On traitera aussi le cas où le pivot est presque nul. En effet, des erreurs d'arrondis pour l'obtention des pivots peuvent engendrer l'apparition d'un pivot très petit mais non nul, alors qu'il est censé être nul mathématiquement. On considérera arbitrairement qu'un pivot est presque nul si sa valeur absolue est inférieure à un certain ε fixé à 10^{-15}. Plus concrètement, l'algorithme est le suivant :

Algorithm 2: Choix du pivot maximal - Cas du pivot partiel

Input : A : tableau[n,n] de flottant ;
n : entier (désignant la taille de A) ;
ε : flottant ;

```
1  // A l'étape i de la première boucle "pour" de l'agorithme 1
2  if valeurAbsolue(a_ii) < ε then
3  |  indicePivotMax = i
4  |  valeurPivotMax = valeurAbsolue(a_ii)
5  |  for j = i, i+1, ..., n do
6  |  |  if valeurAbsolue(a_ij) > valeurPivotMax and valeurAbsolue(a_ij) >= ε then
7  |  |  |  indicePivotMax = j
8  |  |  |  valeurPivotMax = valeurAbsolue(a_ij)
9  |  |  end if
10 |  end for
11 |  if indicePivotMax ≠ i then
12 |  |  permute(Ligne i, Ligne indicePivotMax)
13 |  end if
14 end if
```

Remarque 3.
Les matrices considérées ici, ont toutes une unique solution. Il est alors possible d'appliquer la méthode de Gauss, car les matrices sont forcément inversibles. De plus, le choix du pivot maximal est possible, car la méthode de Gauss est applicable. Plus particulièrement, il va exister une ligne avec qui la permutation sera possible. On a cependant traité le cas où cette permutation n'est pas possible. l'algorithme de triangularisation va s'arrêter, et afficher un message indiquant que la matrice possède une infinité ou aucune solution. Il n'est donc pas possible dans se cas de connaître une matrice x, solution du problème.

3 Présentation des programmes commentés

3.1 Structure générale du programme

Le programme est divisé en en trois fichiers principaux :

- `gauss.c` : contient l'implémentation de la méthode de Gauss : triangularisation, résolution facile, pivot maximal, permutation... ;
- `MatriceTest.c` : contient principalement l'implémentation des matrices de test, affichage, libération de mémoire ;
- `main.c` : contient le programme principal faisant appel aux autres fichiers ;

Remarque 4. A chaque fichier `.c` correspond un fichier `.h`, qui contient essentiellement les signatures des fonctions (sauf pour le `main.c`) ;

3.2 Commantaire du programme

Pour en savoir plus sur nos fonctions, il est possible de se référer aux commentaires dans le code. Il est aussi possible de consulter une documentation plus détaillée. Elle se trouve dans le dossier rendu sous le nom de `documentationCplt.pdf`.

3.3 Choix d'implémentation

3.3.1 `matriceTest.c`

On a décidé de créer une fonction pour chaque matrice. Ce choix d'implémentation permet d'avoir un gain de place en mémoire. En effet, cela permet d'allouer la fonction uniquement si on a l'intention de s'en servir après, et non d'allouer toutes les fonctions au début du programme.

La coordonnée pour une ligne de la matrice b est obtenue en faisant la somme des coefficients de cette même ligne dans la matrice A. Nous avons décidé de calculer ces valeurs à la main, puis de rentrer le résultat directement et non de laisser la machine le calculer, afin d'éviter les potentielles erreurs d'arrondis. Cela permet donc d'éviter des approximations du résultat, avant même d'appliquer la méthode de Gauss.

3.3.2 `gauss.c`

Pour plus de détails, sur les méthodes, se référer aux algorithmes de la section 2.2.

3.3.3 `main.c`

Plutôt que d'afficher toutes les matrices et la résolution associée à ses matrices, nous avons décidé de laisser le choix à l'utilisateur de sélectionner la matrice. On a implémenté ce "choix" de matrice, à l'aide d'un `switch`. Les différents choix possibles sont rappelés grâce à un menu de sélection. Tout ceci se veut être le plus intuitif possible.

3.4 Graphe d'appel des fonctions

Voici également le graphe d'appel de nos fonctions en figure 2.

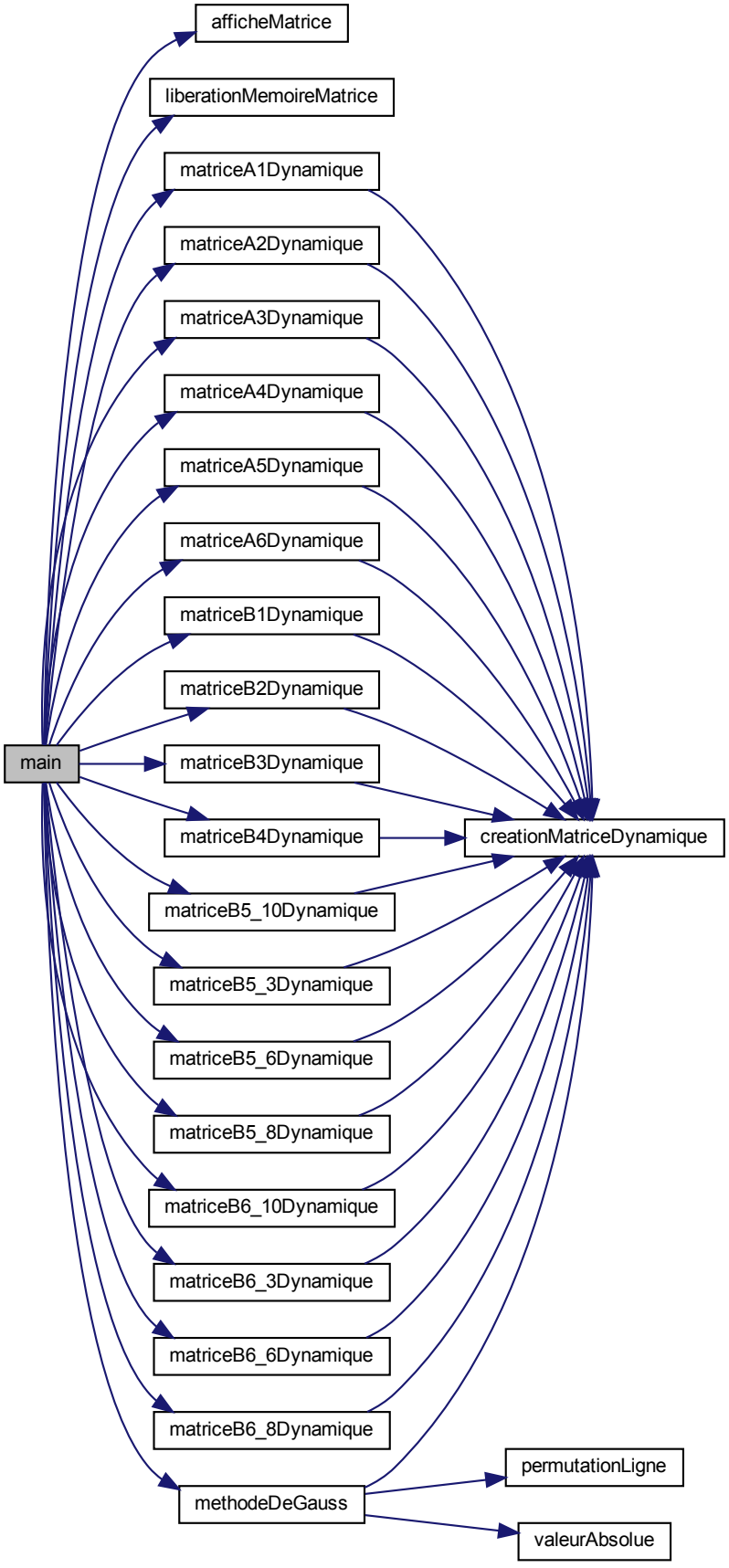

Figure 2 – Graphe d'appel des fonctions

4 Commentaires sur les jeux d'essais

On implémente dans le fichier `matriceTest.c` décrit dans la section 3.3.1 des matrices de test :

$$A_1 = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix}, \; A_2 = \begin{pmatrix} -3 & 3 & 6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}, \; A_3 = \begin{pmatrix} 4 & 1 & 1 \\ 2 & -9 & 0 \\ 0 & -8 & 6 \end{pmatrix}, \; A_4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}$$

Les matrices A_5 et A_6 sont définies différemment. Au lieu de définir un à un tous les coefficients, ces derniers sont déterminés grâce à certaines conditions.

Ainsi, pour $k \in \{5, 6\}$, on a $A_k = (a_{ij})_{i,j \in [\![1,n]\!]}$ avec $n = 3, 6, 8, 10$ telle que :

$$A_5 = \begin{cases} a_{ii} = 1 \\ a_{1j} = a_{j1} = 2^{1-j} \\ 0 \text{ sinon} \end{cases} \quad \text{et } A_6 = \begin{cases} a_{ii} = 3 \\ a_{ij} = -1 \text{ si } j = i+1, i < n \\ a_{ij} = -2 \text{ si } j = i-1, i > 1 \\ 0 \text{ sinon} \end{cases}$$

On choisit les vecteurs b afin que la solution exacte du système d'équations soit : $\bar{x}_i = 1, \forall i \in [\![1, n]\!]$. Pour cela, le i-ième coefficient de la matrice b sera égal à la somme des coefficients sur la i-ième ligne de la matrice A. Autrement dit :

$$\forall i \in [\![1, n]\!], \;\; b_i = \sum_{j=1}^{n} a_{ij}$$

Une fois les matrices définies, on résout le pivot de Gauss avec les différentes matrices A_k.

Avec les programmes présentés ci-dessus, on obtient pour chacune d'entre elles une matrice $\bar{x}$ avec des coefficients tous égaux à 1.0. Les programmes ont donc bien le comportement souhaité.

Le choix du type `double` au lieu du type `float` dans les programmes permet d'augmenter la précision des résultats et donc de limiter les erreurs.

5 Conclusion générale

Pour conclure, la méthode de Gauss fonctionne de manière plutôt efficace. En effet, en l'implémentant en `C` et en la testant avec les matrices de test, on obtient exactement les résultats souhaités.

Cependant, pour une matrice de taille n, la méthode de Gauss effectue $\frac{2n^3}{3}$ opérations. Elle a donc une complexité en $O(n^3)$. D'autres méthodes permettent de résoudre des systèmes d'équations linéaires avec une complexité moindre. On a, par exemple, des méthodes directes comme celle de Cholesky ($\frac{n^3}{2}$ opérations) et des méthodes indirectes comme celle de Jacobi ou celle de Gauss-Seidel qui eux ont une complexité moins importante, notamment pour les matrices *creuses*.

Nous aborderons ces deux dernières méthodes de résolution dans le prochain TP.