Application en Ingénieurie et Programmation Numérique

"Rendu I - Méthodes Directes"

VILLEDIEU Maxance et BESQUEUT Corentin

1 octobre 2023

Table des matières

I Résolution de systèmes linéaires par des Méthodes Directes Méthode de Gauss **3**

I.1 Détail de l'algorithme 3

I.2 Exemples 4

I.2.1 Implémentation du pivot de Gauss sans matrice Augmentée 4

I.3 Pivot de Gauss avec matrice augmentée 4

I.3.1 Code source 4

I.3.2 Commentaires du code 7

I.3.3 Inputs / Outputs 8

I.3.4 Exemples d'exécutions 9

Table des figures

Chapitre I

Résolution de systèmes linéaires par des Méthodes Directes Méthode de Gauss

Dans le cadre de ce premier TP, nous devions implémenter l'algorithme du *Pivot de Gauss* en utilisant le langage de programmation C.

I.1 Détail de l'algorithme

Soit A une matrice $\in \mathcal{M}_{m,m}$ $m \in \mathbb{N}$ et $b \in \mathcal{M}_{m,1}$.
L'algorithme de Gauss se décrit ainsi :

Pour $k = 1, \ldots, n-1$ Faire :
Pour $i = k+1, \ldots, n$ Faire :

$$\alpha_i^{(k)} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$$

Pour $j = k, \ldots, n$ Faire :

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \alpha_i^{(k)} a_{kj}^{(k)}$$

FIN Pour j

$$b_i^{(k+1)} = b_i^{(k)} - \alpha_i^{(k)} b_k^{(k)}$$

FIN Pour i
FIN Pour k

Après cette algorithme permettant l'échelonnage de la matrice, pour trouver les solutions du système, on appliquera la formule suivante :

$x_n = \frac{b_n}{a_{m,m}}$
et

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

La complexité temporelle de cet algorithme est cubique soit $O(n^3)$ avec une complexité exacte de $\frac{2n^3}{3}$.
Pour l'implémentation de cet algorithme, nous allons présenter deux façons de le conceptualiser avec une comparaison algorithmique des deux programmes.

I.2 Exemples

Soit $A \in \mathcal{M}_{m,m}$ et $B \in \mathcal{M}_{m,1}$ et x la matrice des inconnues.
Considérons alors le Système suivant $Ax = b$.
Ce système peut-être représenter sous la forme d'une matrice augmentée M tel que :

$$M = \begin{pmatrix} a_{11} & \dots & a_{1m} & | & b_1 \\ \vdots & \ddots & \vdots & | & \vdots \\ a_{m1} & \dots & a_{mm} & | & b_m \end{pmatrix}$$

Après exécution du pivot de Gauss, M devient

$$M = \begin{pmatrix} 1 & a_{12} & a_{13} & \dots & a_{1m} & | & b_1^{'} \\ 0 & 1 & a_{23}^{'} & \dots & a_{2m}^{'} & | & b_2^{'} \\ 0 & 0 & 1 & \dots & \vdots & | & b_3^{'} \\ \vdots & \vdots & \vdots & \ddots & \vdots & | & \vdots \\ 0 & \dots & 0 & \dots & 1 & | & b_m^{'} \end{pmatrix}$$

Une fois que tout les pivots sont placés, il suffira de reconstituer le système et de le remonter afin de déterminer les inconnues comme suit : Soit $A^{'} = \begin{pmatrix} 1 & a_{12}^{'} & a_{13}^{'} & \dots & a_{1m}^{'} \\ 0 & 1 & a_{23}^{'} & \dots & a_{2m}^{'} \\ 0 & 0 & 1 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 1 \end{pmatrix}$ et $b = \begin{pmatrix} b_1^{'} \\ b_2^{'} \\ b_3^{'} \\ \vdots \\ b_m^{'} \end{pmatrix}$

alors pour $A^{'}x = b$ on a donc $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1+i}^{n} a_{ij} x_j \right) \forall i = n-1, \dots, 1$.

Nous remarquerons que l'implémentation du pivot de Gauss ne necéssitera pas de mettre nos pivot à 1

I.2.1 Implémentation du pivot de Gauss sans matrice Augmentée

I.3 Pivot de Gauss avec matrice augmentée

I.3.1 Code source

Voici le code source de mon implémentation du pivot de Gauss via le passage par la matrice augmentée.
C'est-à-dire que dans mon implémentation, on fera usage de la concaténation des matrices.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
/*
 * PRINT MATRIX WITH RIGHT FORMAT
 */
void printMatrix(float **matrix, int m, int p) {
  printf("PRINTING MATRIX FROM: %p LOCATION : \n", matrix);
```

```c
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
      (j <= p - 2) ? printf("%f ", matrix[i][j]) : printf("%f", matrix[i][j]);
    }
    puts("");
  }
}
/*
 * ALLOCATE MEMORY FOR MATRIX
 */
float **allocate(int m, int n) {
  float **T = malloc(m * sizeof *T);
  for (int i = 0; i < m; i++) {
    T[i] = malloc(n * sizeof *T[i]);
    if (T[i] == NULL) {
      for (int j = 0; j < i; j++) {
        free(T[i]);
      }
      free(T);
      puts("ALLOCATION ERROR");
      exit(-1);
    }
  }
  return T;
}
/*
 * FILL MATRIX BY USER INPUT
 */
void fillM(int m, int p, float **T) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
      T[i][j] = 0;
      printf("Enter coefficient for %p[%d][%d]", T, i, j);
      scanf("%f", &T[i][j]);
    }
  }
}
/*
 * FREE MATRIX
 */
void freeAll(float **T, int m) {
  for (int i = 0; i < m; i++) {
    free(T[i]);
  }
  free(T);
}
/*
 * IMPLEMENTATION OF '.' OPERATOR FOR MATRIX
 */
float **multiplication(float **M1, float **M2, int m, int q) {
  float **R = allocate(m, q);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < q; j++) {
      for (int k = 0; k < q; k++) {
```

```
        R[i][j] += M1[i][k] * M2[k][j];
      }
    }
  }
  return R;
}
/*
 * BUILD AUGMENTED MATRIX
 */
float **AugmentedMatrix(float **M1, float **M2, int m, int n) {
  float **A = allocate(m, m + 1);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n + 1; j++) {
      (j != n) ? (A[i][j] = M1[i][j]) : (A[i][j] = M2[i][0]);
    }
  }
  return A;
}
/*
 * PERFORM GAUSS ALGORITHM ONLY ON AUGMENTED MATRIX
 */
void gauss(float **A, int m, int p) {
  if (m != p) {
    puts("La matrice doit etre carree !");
    return;
  }
  for (int k = 0; k <= m - 1; k++) {
    for (int i = k + 1; i < m; i++) {
      float pivot = A[i][k] / A[k][k];
      for (int j = k; j <= m; j++) {
        A[i][j] = A[i][j] - pivot * A[k][j];
      }
    }
  }
}
/*
 * DETERMINE ALL UNKNOWNS VARIABLES
 */
float *findSolutions(float **A, int m) {
  float *S = calloc(m, sizeof *S);
  S[m - 1] = A[m - 1][m] / A[m - 1][m - 1];
  for (int i = m - 1; i >= 0; i--) {
    S[i] = A[i][m];
    for (int j = i + 1; j < m; j++) {
      S[i] -= A[i][j] * S[j];
    }
    S[i] = S[i] / A[i][i];
  }
  return S;
}
int main() {
  int m, n, p, q;
  float **P, **Q, **B, **A, *S;
  clock_t start, end;
```

```
    double execution;
    puts("Nombre de ligne suivit du nombre de colonne pour la matrice 1:");
    scanf("%d%d", &m, &p);
    puts("Nombre de ligne suivit du nombre de colonne pour la matrice 2:");
    scanf("%d%d", &n, &q);
    start = clock();
    P = allocate(m, p);
    Q = allocate(n, q);
    fillM(m, p, P);
    fillM(n, q, Q);
    printMatrix(P, m, p);
    printMatrix(Q, n, q);
    B = multiplication(P, Q, m, n);
    printMatrix(B, m, q);
    A = AugmentedMatrix(P, B, m, p);
    gauss(A, m, p);
    printMatrix(A, m, m + 1);
    S = findSolutions(A, m);
    puts("SOLUTIONS");
    for (int i = 0; i < m; i++)
      printf("x%d = %f\n", i, S[i]);
    freeAll(P, m);
    freeAll(Q, n);
    freeAll(B, m);
    freeAll(A, m);
    free(S);
    end = clock();
    execution = ((double)(end - start) / CLOCKS_PER_SEC);
    printf("RUNTIME: %f seconds", execution / 10);
    return 0;
}
```

I.3.2 Commentaires du code

Mon implémentation utilise strictement l'algorithme de Gauss rappelé précédemment avec seulement quelques changements d'indices puisque au lieu de travailler sur une matrice carrée et un vecteur colonne, mon programme utilise une matrice augmentée ayant m lignes et $m+1$ colonnes, $m \in \mathbb{N}^*$.

Détail des fonctions non conventionnelles :

Comme mentioné précédemment, je ne détaillerai pas les fonction gauss() et findSolutions() puisque ces fonctions permettent strictement que d'une part d'implémenter l'algorithme de Gauss et d'autre part à "remonter" la matrice échelonnée afin de récupérer les valeurs des inconnus.

-float **AugmentedMatrix(float **M1, float **M2, int m, int n) : cette fonction permet de créer une matrice $A \in \mathcal{M}_{m,m+1}$ à partir de la concaténation de $M1 \in \mathcal{M}_{mm}$ et $M2 \in \mathcal{M}_{m,1}$.

Soient a_{ij} les coefficients peuplant A, b_{ij} les coefficients peuplant $M1$ et c_{i0} les coefficients peuplant $M2$.

On obtient alors $a_{ij} = b_{ij} \forall i \in \mathbb{N}_m, \forall j \in \mathbb{N}_m$ et $a_{ij} = c_{i0}$ si $j = m+1$.

Cette fonction renvoie alors A, la matrice de flottants créee dynamiquement.

Les fonctions *fillM()*, *printMatrix()*, *freeAll()* et *multiplication()* et *allocate()* sont quatre fonctions utilitaires qui permettent respectivement : de remplir une matrice, d'afficher convenablement une matrice, de libérer les matrices en mémoires, de définir la multiplication matricielle et enfin d'allouer de la mémoire pour déclarer les matrices (avec quelques légères sécurités permettant d'être sûr que les matrices sont bien instanciées convenablement).

I.3.3 Inputs / Outputs

Mon programme demande d'abord 4 entiers m, p, n, q qui correspondent aux dimensions de la première matrice $A \in \mathcal{M}_{mp}$ et de la seconde matrice $X \in \mathcal{M}_{nq}$. Le but étant de résoudre le système $AX = b$, nous initialiserons X à 1. Ce choix de valeur permettra de controller la validité du programme, ainsi si à la fin du programme $\forall x_i \neq 1, \forall i \in \mathbb{N}_n$, on pourra affirmer que le programme est faux.
Sur le $m \times p$ prochaines lignes, le programme demande les coefficients de A.
Sur les $n \times q$ prochaines lignes, le programme demandera les coefficient du vecteur colonne X, que l'utilisateur initialisera à 1.
On peut alors automatiser les entrées en utilisant des fichiers.
Ainsi la matrice : $M = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix}$ et $X = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
sont représentés par ce fichier d'entrées :

Listing I.1 – input.txt

```
3 3
3 1
3 0 4
7 4 2
-1 1 2
1 1 1
```

Pour ce qui est des résultats produits par mon programme, une fois injecté dans un fichier texte, une input "type" ressemble à ceci.

Listing I.2 – Gauss elimination with M and X matrix

```
PRINTING MATRIX FROM: 0x556d938672c0 LOCATION :
3.000000 0.000000 4.000000
7.000000 4.000000 2.000000
-1.000000 1.000000 2.000000
PRINTING MATRIX FROM: 0x556d93867340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x556d938673c0 LOCATION :
7.000000
13.000000
2.000000
PRINTING MATRIX FROM: 0x556d93867440 LOCATION :
3.000000 0.000000 4.000000 7.000000
0.000000 4.000000 -7.333333 -3.333332
0.000000 0.000000 5.166667 5.166667
SOLUTIONS
x0 = 1.000000
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

On remarquera que le programme affiche dans cet ordre :
- La Matrice A
- La Matrice X
- La Matrice B trouvé avec les valeur de X
- La Matrice augmentée en triangle supérieur

- Les solutions
- Un timer permettant de contrôler le temps d'exécution approximatif de mon programme

I.3.4 Exemples d'exécutions

Soient $A_2 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}$, $A_4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}$, $A_6 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}$
On obtient respectivement ces résulats :

Listing I.3 – Matrix 2 results

```
PRINTING MATRIX FROM: 0x55f604fb32c0 LOCATION :
-3.000000 3.000000 -6.000000
-4.000000 7.000000 8.000000
5.000000 7.000000 -9.000000
PRINTING MATRIX FROM: 0x55f604fb3340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x55f604fb33c0 LOCATION :
-6.000000
11.000000
3.000000
PRINTING MATRIX FROM: 0x55f604fb3440 LOCATION :
-3.000000 3.000000 -6.000000 -6.000000
0.000000 3.000000 16.000000 19.000000
0.000000 0.000000 -83.000000 -83.000000
SOLUTIONS
x0 = 1.000000
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

Listing I.4 – Matrix 4 results

```
PRINTING MATRIX FROM: 0x55f7afd662c0 LOCATION :
7.000000 6.000000 9.000000
4.000000 5.000000 -4.000000
-7.000000 -3.000000 8.000000
PRINTING MATRIX FROM: 0x55f7afd66340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x55f7afd663c0 LOCATION :
22.000000
5.000000
-2.000000
PRINTING MATRIX FROM: 0x55f7afd66440 LOCATION :
7.000000 6.000000 9.000000 22.000000
0.000000 1.571428 -9.142858 -7.571429
0.000000 0.000000 34.454552 34.454548
SOLUTIONS
x0 = 1.000001
x1 = 0.999999
```

```
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

Listing I.5 – Matrix 6 results

```
PRINTING MATRIX FROM: 0x557fdaa552c0 LOCATION :
3.000000 -1.000000 0.000000
0.000000 3.000000 -1.000000
0.000000 -2.000000 3.000000
PRINTING MATRIX FROM: 0x557fdaa55340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x557fdaa553c0 LOCATION :
2.000000
2.000000
1.000000
PRINTING MATRIX FROM: 0x557fdaa55440 LOCATION :
3.000000 -1.000000 0.000000 2.000000
0.000000 3.000000 -1.000000 2.000000
0.000000 0.000000 2.333333 2.333333
SOLUTIONS
x0 = 1.000000
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

On remarquera que sur le calcul de A_4, on tombe sur des valeur extrêmement proche de 1. Ceci est provoqué à cause des erreurs d'arrondis provoqués par l'encodage des nombres flottants.