

Application en Ingénierie 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.3	Implémentation de l'algorithme de Gauss en passant par le système d'équations linéaires	4
I.3.1	Code source	4
I.3.2	Commentaires	7
I.3.3	Interaction Utilisateur/Console	7
I.3.4	Exemples d'exécutions	9
I.4	Pivot de Gauss avec matrice augmentée	10
I.4.1	Code source	10
I.4.2	Commentaires du code	13
I.4.3	Inputs / Outputs	14
I.4.4	Exemples d'exécutions	15

Table des figures

Chapitre I

Résolution de systèmes linéaires par des Méthodes Directes : Méthode de Gauss

Dans le cadre de ce premier TP, nous devons 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 $k = 1, \dots, n-1$ Faire :
 Pour $i = k+1, \dots, n$ Faire :

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

 Pour $j = k, \dots, n$ 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 :

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

et

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

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 :

$$M = \left(\begin{array}{cccc|c} a_{11} & \dots & a_{1m} & & b_1 \\ \vdots & \ddots & \vdots & & \vdots \\ a_{m1} & \dots & a_{mm} & & b_m \end{array} \right)$$

Après exécution du pivot de Gauss, M devient

$$M = \left(\begin{array}{ccccc|c} 1 & a_{12} & a_{13} & \dots & a_{1m} & b'_1 \\ 0 & 1 & a'_{23} & \dots & a'_{2m} & b'_2 \\ 0 & 0 & 1 & \dots & \vdots & b'_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & 1 & b'_m \end{array} \right)$$

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 nécessitera pas de mettre nos pivot à 1

I.3 Implémentation de l'algorithme de Gauss en passant par le système d'équations linéaires

I.3.1 Code source

Voici mon implémentation de l'algorithme de Gauss, qui n'utilise pas la matrice augmentée. En effet, l'algorithme travaille directement avec le système d'équations linéaires $Ax=B$.

```

0 #include <stdio.h>
1 #include <string.h>
2 #include <stdlib.h>
3
4 /*
5 *CREATE A 2D FLOAT MATRIX
6 */
7
8 float** createMatrix(int row, int column){
9     float **mat=NULL;
10    mat=malloc(row* sizeof(int*));

```

```

11     if(mat==NULL){return NULL;}
12     for (int i=0; i<row; i++){
13         mat[i]=malloc(column* sizeof(int));
14         if(mat[i]==NULL){
15             for(int j=0; j<i; j++){
16                 free(mat[j]);
17                 return NULL;
18             }
19         }
20     }
21     return mat;
22 }
23
24 /*
25 *PRINT A 2D FLOAT MATRIX
26 */
27
28 void printMatrix(float **mat, int row, int column){
29     for (int i=0; i<row; i++){
30         for(int j=0; j<column; j++){
31             printf("%f ", mat[i][j]);
32         }
33         printf("\n");
34     }
35 }
36
37 /*
38 *FREE A 2D FLOAT MATRIX
39 */
40
41 void freeMatrix(float **mat, int row){
42     for(int i=0; i<row; i++){
43         free(mat[i]);
44     }
45     free(mat);
46 }
47
48 /*
49 *COMPLETE A 2D FLOAT MATRIX FROM USER INPUT
50 */
51
52 void completeMatrix(float **mat, int row, int column){
53     for (int i=0; i<row; i++){
54         for(int j=0; j<column; j++){
55             printf("Coefficient at M_%d,%d: ", i+1, j+1);
56             scanf("%f", &mat[i][j]);
57         }
58     }
59 }
60
61 /*
62 *GENERATE A COLUMN VECTOR "B" FROM A 2D FLOAT MATRIX "A"
63 */
64
65 void generateB(float **matA, float **matB, int row, int column){
66     for(int i=0; i<row; i++){
67         float sum=0;
68         for(int j=0; j<column; j++){
69             sum+=matA[i][j];
70         }
71         matB[i][0]=sum;
72     }
73 }
74
75 /*

```

```

76  *PERFORM GAUSSIAN ELIMINATION ON A Ax=B MATRIX SYSTEM OF LINEAR EQUATIONS
77  */
78
79  void gauss(float** matA, float** matb, int size){
80      for(int k=0; k<size-1; k++){
81          for(int i=k+1; i<size; i++){
82              float alpha=matA[i][k]/matA[k][k];
83              for(int j=k; j<size; j++){
84                  matA[i][j]=matA[i][j]-alpha*matA[k][j];
85              }
86              matb[i][0]=matb[i][0]-alpha*matb[k][0];
87          }
88      }
89  }
90
91  /*
92  *SOLVE A MATRIX SYSTEM OF LINEAR EQUATIONS USING BACKWARD SUBSTITUTION
93  */
94  void resolution(float** matA, float** matb, float** matx, int size){
95      matx[size-1][0]=matb[size-1][0]/matA[size-1][size-1];
96      for (int i=size-2; i>=0; i--){
97          float sum=0;
98          for(int j=i+1; j<size; j++){
99              sum+=matA[i][j]*matx[j][0];
100          }
101          matx[i][0]=(1/matA[i][i])*(matb[i][0]-sum);
102      }
103  }
104
105  int main(){
106
107      //A Matrix
108      int rowA;
109      int columnA;
110      printf("\nRow count of matrix A : ");
111      scanf("%d", &rowA);
112      printf("\nColumn count of matrix A : ");
113      scanf("%d", &columnA);
114
115      float** Amatrix=createMatrix(rowA, columnA);
116      completeMatrix(Amatrix, rowA, columnA);
117      puts("\n      A matrix \n");
118      printMatrix(Amatrix, rowA, columnA);
119
120      //B Matrix
121      float** Bmatrix=createMatrix(rowA, 1);
122      generateB(Amatrix, Bmatrix, rowA, columnA);
123      puts("\n      B matrix \n");
124      printMatrix(Bmatrix, rowA, 1);
125
126      //X Matrix
127      float** Xmatrix=createMatrix(rowA, 1);
128
129      //Matrix Triangularization
130      puts("\n      TRIANGULARIZATION \n");
131      gauss(Amatrix, Bmatrix, rowA);
132      puts("\n      A Matrix \n");
133      printMatrix(Amatrix, rowA, columnA);
134      puts("\n      B Matrix \n");
135      printMatrix(Bmatrix, rowA, 1);
136
137      //Solve the system
138      puts("\n      SOLVING \n");
139      resolution(Amatrix, Bmatrix, Xmatrix, rowA);
140      puts("\n      SOLUTION VECTOR X \n");

```

```
141     printMatrix(Xmatrix, rowA, 1);
142
143     //Free
144     freeMatrix(Amatrix, rowA);
145     freeMatrix(Bmatrix, rowA);
146     freeMatrix(Xmatrix, rowA);
147     return 0;
148 }
```

I.3.2 Commentaires

Fonctions usuelles de manipulation de matrices

Ce code implémente diverses fonctions pour travailler avec des matrices à coefficients en nombre flottants.

- La fonction ***createMatrix*** alloue dynamiquement de la mémoire pour créer une matrice de nombres flottants avec un nombre spécifié de lignes et de colonnes.
- La fonction ***printMatrix*** affiche les éléments d'une matrice de nombres flottants.
- La fonction ***freeMatrix*** libère la mémoire allouée pour une matrice de nombres flottants.
- La fonction ***completeMatrix*** permet à l'utilisateur de saisir des valeurs pour remplir les éléments d'une matrice de nombres flottants.
- La fonction ***generateB*** génère un vecteur colonne B en fonction de la somme des éléments de chaque ligne de la matrice A .

Fonctions résolvant notre système linéaire $Ax = B$ à l'aide de l'algorithme de Gauss

Dans le cadre de notre résolution de systèmes d'équations linéaires, deux fonctions jouent des rôles clés dans ce code : la fonction ***gauss*** et la fonction ***resolution***.

- La fonction ***gauss*** joue un rôle important dans la préparation de la résolution de notre système d'équations linéaires. En effectuant l'élimination de Gauss sur la matrice A , elle la transforme en une matrice triangulaire supérieure. Cela signifie que les éléments sous la diagonale principale de la matrice deviennent tous des zéros, simplifiant ainsi la résolution du système. De plus, la fonction met également à jour la matrice B en conséquence, garantissant que notre système $Ax = B$ reste équilibré.
- La fonction ***resolution***, quant à elle, prend en charge la résolution effective du système linéaire une fois que la matrice A a été triangulée par la fonction ***gauss***. Elle utilise la méthode de substitution pour calculer la solution et stocke le résultat dans le vecteur X . Cette étape finale permet d'obtenir les valeurs des variables inconnues du système, fournissant ainsi la solution recherchée pour le problème initial.

En combinant ces deux fonctions avec celles citées dans la sous-section I.3.2, le code réalise un processus complet de résolution de systèmes d'équations linéaires de manière efficace et précise (aux erreurs d'arrondies près).

I.3.3 Interaction Utilisateur/Console

Entrées utilisateur

En premier lieu dans notre programme, nous avons besoin de spécifier le système $Ax = B$ à l'ordinateur. Pour ce faire, nous allons dans l'ordre :

1. Allouer une matrice A en mémoire. Cette matrice verra sa taille définie par la première entrée utilisateur du programme (nous demanderons consécutivement le nombre de lignes, puis le nombre de colonnes de la matrice).
2. Définir les coefficients de la matrice A . Il s'agira de la deuxième entrée utilisateur de notre programme.
Par définition de notre fonction ***completeMatrix***, nous remplirons la matrice dans l'ordre suivant :
 $a_{1,1}, a_{1,2}, \dots, a_{1,n}$, puis $a_{2,1}, \dots, a_{2,n}$, jusque $a_{n,1}, \dots, a_{n,n}$
3. Allouer une matrice B en mémoire. À noter que la taille de B est définie automatiquement en fonction de la taille de A . Nous avons $A \in \mathcal{M}_{n,p} \Rightarrow B \in \mathcal{M}_{n,1}$.
4. Définir les coefficients de la matrice B . Chaque coefficient prendra la valeur de la somme des éléments de la ligne respective de la matrice A .
Nous avons donc :
Soient $A \in \mathcal{M}_{n,p}$ et $B \in \mathcal{M}_{n,1}, \forall i \in \{1, n\}, b_{i,1} = \sum_{j=1}^p a_{i,j}$.
5. Allouer une matrice X en mémoire. Cette matrice aura la même taille que la matrice B . Ces coefficients ne seront pas définis pour le moment.

En guise d'exemple, le système matriciel $AX = B$ suivant :

$$\begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 13 \\ 2 \end{pmatrix} \quad (\text{I.1})$$

est représenté par l'entrée utilisateur :

Listing I.1 – User Input

```

0 Row count of matrix A : 3
1
2 Column count of matrix A : 3
3
4 FILL IN THE VALUE OF MATRIX A
5
6 Value for a_1,1: 3
7 Value for a_1,2: 0
8 Value for a_1,3: 4
9 Value for a_2,1: 7
10 Value for a_2,2: 4
11 Value for a_2,3: 2
12 Value for a_3,1: -1
13 Value for a_3,2: 1
14 Value for a_3,3: 2

```

Une fois toutes les matrices initialisées et complétées, nous pouvons attaquer la résolution du système par la triangularisation du système. Ceci fait, nous résolverons le système obtenu pour obtenir notre vecteur X solution.

Affichage Console

Dès lors le système $AX = B$ connu par l'ordinateur, ce dernier peut retrouver les valeurs de la matrice X . Voici l'affichage produit par notre programme en console :

Listing I.2 – Console Display of the Gauss elimination for the AX

```

0 A matrix
1
2 3.000000 0.000000 4.000000
3 7.000000 4.000000 2.000000
4 -1.000000 1.000000 2.000000
5

```

```

6          B matrix
7
8      7.000000
9      13.000000
10     2.000000
11
12          TRIANGULARIZATION
13          A Matrix
14
15     3.000000    0.000000    4.000000
16     0.000000    4.000000   -7.333333
17     0.000000    0.000000    5.166667
18
19          B Matrix
20
21     7.000000
22    -3.333332
23     5.166667
24
25          SOLVING
26          SOLUTION VECTOR X
27
28     1.000000
29     1.000000
30     1.000000

```

Il est à repérer que le programme affiche dans cet ordre :

- La Matrice A
- La Matrice B
- La Matrice A une fois triangulée supérieure
- La Matrice B une fois mise à jour en conséquence pour que le système reste équilibré
- La Matrice X solution du système

Remarque : le temps d'exécution de ce programme a été de 0.000237 secondes

I.3.4 Exemples d'exécutions

Soient les matrices suivantes données dans le TP :

$$A_2 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}, A_4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}, A_6 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}$$

On obtient respectivement les résultats suivants :

Listing I.3 – $A_2X = B$ results

```

0          A matrix
1
2     -3.000000    3.000000   -6.000000
3     -4.000000    7.000000    8.000000
4     5.000000    7.000000   -9.000000
5
6          B matrix
7
8     -6.000000
9     11.000000
10     3.000000
11
12          TRIANGULARIZATION
13          A Matrix
14
15     -3.000000    3.000000   -6.000000
16     0.000000    3.000000   16.000000
17     0.000000    0.000000  -83.000000
18
19          B Matrix
20
21     -6.000000
22     19.000000
23    -83.000000
24
25          SOLVING
26          SOLUTION VECTOR X
27
28     1.000000
29     1.000000
30     1.000000
31

```

32 | Temps d'exécution : 0.000250 secondes

Listing I.4 – $A_4X = B$ results

```

0      A matrix
1
2      7.000000    6.000000    9.000000
3      4.000000    5.000000   -4.000000
4      -7.000000   -3.000000    8.000000
5
6      B matrix
7
8      22.000000
9      5.000000
10     -2.000000
11
12     TRIANGULARIZATION
13     A Matrix
14
15     7.000000    6.000000    9.000000
16     0.000000    1.571428   -9.142858
17     0.000000    0.000000    34.454552
18
19     B Matrix
20
21     22.000000
22     -7.571429
23     34.454548
24
25     SOLVING
26     SOLUTION VECTOR X
27
28     1.000001
29     0.999999
30     1.000000
31
32     Temps d'exécution : 0.000231 secondes

```

Listing I.5 – $A_6X = B$ results

```

0      A matrix
1
2      -3.000000    3.000000   -6.000000
3      -4.000000    7.000000    8.000000
4      5.000000    7.000000   -9.000000
5
6      B matrix
7
8      -6.000000
9      11.000000
10     3.000000
11
12     TRIANGULARIZATION
13     A Matrix
14
15     -3.000000    3.000000   -6.000000
16     0.000000    3.000000   16.000000
17     0.000000    0.000000  -83.000000
18
19     B Matrix
20
21     -6.000000
22     19.000000
23     -83.000000
24
25     SOLVING
26     SOLUTION VECTOR X
27
28     1.000000
29     1.000000
30     1.000000
31
32     Temps d'exécution : 0.000246 secondes

```

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.

I.4 Pivot de Gauss avec matrice augmentée

I.4.1 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.

```

0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <string.h>

```

```

3 #include <time.h>
4 /*
5  * PRINT MATRIX WITH RIGHT FORMAT
6  */
7 void printMatrix(float **matrix, int m, int p) {
8     printf("PRINTING MATRIX FROM: %p LOCATION : \n", matrix);
9     for (int i = 0; i < m; i++) {
10         for (int j = 0; j < p; j++) {
11             (j <= p - 2) ? printf("%f ", matrix[i][j]) : printf("%f", matrix[i][j]);
12         }
13         puts("");
14     }
15 }
16 /*
17  * ALLOCATE MEMORY FOR MATRIX
18  */
19 float **allocate(int m, int n) {
20     float **T = malloc(m * sizeof *T);
21     for (int i = 0; i < m; i++) {
22         T[i] = malloc(n * sizeof *T[i]);
23         if (T[i] == NULL) {
24             for (int j = 0; j < i; j++) {
25                 free(T[j]);
26             }
27             free(T);
28             puts("ALLOCATION ERROR");
29             exit(-1);
30         }
31     }
32     return T;
33 }
34 /*
35  * FILL MATRIX BY USER INPUT
36  */
37 void fillM(int m, int p, float **T) {
38     for (int i = 0; i < m; i++) {
39         for (int j = 0; j < p; j++) {
40             T[i][j] = 0;
41             printf("Enter coefficient for %p[%d][%d]", T, i, j);
42             scanf("%f", &T[i][j]);
43         }
44     }
45 }
46 /*
47  * FREE MATRIX
48  */
49 void freeAll(float **T, int m) {
50     for (int i = 0; i < m; i++) {
51         free(T[i]);
52     }
53     free(T);
54 }
55 /*
56  * IMPLEMENTATION OF ' . ' OPERATOR FOR MATRIX

```

```

57  */
58  float **multiplication(float **M1, float **M2, int m, int q) {
59      float **R = allocate(m, q);
60      for (int i = 0; i < m; i++) {
61          for (int j = 0; j < q; j++) {
62              for (int k = 0; k < q; k++) {
63                  R[i][j] += M1[i][k] * M2[k][j];
64              }
65          }
66      }
67      return R;
68  }
69  /*
70  * BUILD AUGMENTED MATRIX
71  */
72  float **AugmentedMatrix(float **M1, float **M2, int m, int n) {
73      float **A = allocate(m, m + 1);
74      for (int i = 0; i < m; i++) {
75          for (int j = 0; j < n + 1; j++) {
76              (j != n) ? (A[i][j] = M1[i][j]) : (A[i][j] = M2[i][0]);
77          }
78      }
79      return A;
80  }
81  /*
82  * PERFORM GAUSS ALGORITHM ONLY ON AUGMENTED MATRIX
83  */
84  void gauss(float **A, int m, int p) {
85      if (m != p) {
86          puts("La matrice doit etre carree !");
87          return;
88      }
89      for (int k = 0; k <= m - 1; k++) {
90          for (int i = k + 1; i < m; i++) {
91              float pivot = A[i][k] / A[k][k];
92              for (int j = k; j <= m; j++) {
93                  A[i][j] = A[i][j] - pivot * A[k][j];
94              }
95          }
96      }
97  }
98  /*
99  * DETERMINE ALL UNKNOWN VARIABLES
100 */
101 float *findSolutions(float **A, int m) {
102     float *S = calloc(m, sizeof *S);
103     S[m - 1] = A[m - 1][m] / A[m - 1][m - 1];
104     for (int i = m - 1; i >= 0; i--) {
105         S[i] = A[i][m];
106         for (int j = i + 1; j < m; j++) {
107             S[i] -= A[i][j] * S[j];
108         }
109         S[i] = S[i] / A[i][i];
110     }

```

```

111     return S;
112 }
113 int main() {
114     int m, n, p, q;
115     float **P, **Q, **B, **A, *S;
116     clock_t start, end;
117     double execution;
118     puts("Nombre de ligne suivit du nombre de colonne pour la matrice 1:");
119     scanf("%d%d", &m, &p);
120     puts("Nombre de ligne suivit du nombre de colonne pour la matrice 2:");
121     scanf("%d%d", &n, &q);
122     start = clock();
123     P = allocate(m, p);
124     Q = allocate(n, q);
125     fillM(m, p, P);
126     fillM(n, q, Q);
127     printMatrix(P, m, p);
128     printMatrix(Q, n, q);
129     B = multiplication(P, Q, m, n);
130     printMatrix(B, m, q);
131     A = AugmentedMatrix(P, B, m, p);
132     gauss(A, m, p);
133     printMatrix(A, m, m + 1);
134     S = findSolutions(A, m);
135     puts("SOLUTIONS");
136     for (int i = 0; i < m; i++)
137         printf("x%d = %f\n", i, S[i]);
138     freeAll(P, m);
139     freeAll(Q, n);
140     freeAll(B, m);
141     freeAll(A, m);
142     free(S);
143     end = clock();
144     execution = ((double)(end - start) / CLOCKS_PER_SEC);
145     printf("RUNTIME: %f seconds", execution / 10);
146     return 0;
147 }

```

I.4.2 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 mentionné 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 créée 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.4.3 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 contrôler 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 coefficients 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.6 – input.txt

```
0 3 3
1 3 1
2 3 0 4
3 7 4 2
4 -1 1 2
5 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.7 – Gauss elimination with M and X matrix

```
0 PRINTING MATRIX FROM: 0x556d938672c0 LOCATION :
1 3.000000 0.000000 4.000000
2 7.000000 4.000000 2.000000
3 -1.000000 1.000000 2.000000
4 PRINTING MATRIX FROM: 0x556d93867340 LOCATION :
5 1.000000
6 1.000000
7 1.000000
8 PRINTING MATRIX FROM: 0x556d938673c0 LOCATION :
9 7.000000
10 13.000000
11 2.000000
12 PRINTING MATRIX FROM: 0x556d93867440 LOCATION :
13 3.000000 0.000000 4.000000 7.000000
14 0.000000 4.000000 -7.333333 -3.333332
15 0.000000 0.000000 5.166667 5.166667
16 SOLUTIONS
17 x0 = 1.000000
18 x1 = 1.000000
19 x2 = 1.000000
20 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.4.4 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 résultats :

Listing I.8 – Matrix 2 results

```

0 PRINTING MATRIX FROM: 0x55f604fb32c0 LOCATION :
1 -3.000000 3.000000 -6.000000
2 -4.000000 7.000000 8.000000
3 5.000000 7.000000 -9.000000
4 PRINTING MATRIX FROM: 0x55f604fb3340 LOCATION :
5 1.000000
6 1.000000
7 1.000000
8 PRINTING MATRIX FROM: 0x55f604fb33c0 LOCATION :
9 -6.000000
10 11.000000
11 3.000000
12 PRINTING MATRIX FROM: 0x55f604fb3440 LOCATION :
13 -3.000000 3.000000 -6.000000 -6.000000
14 0.000000 3.000000 16.000000 19.000000
15 0.000000 0.000000 -83.000000 -83.000000
16 SOLUTIONS
17 x0 = 1.000000
18 x1 = 1.000000
19 x2 = 1.000000
20 RUNTIME: 0.000002 seconds

```

Listing I.9 – Matrix 4 results

```

0 PRINTING MATRIX FROM: 0x55f7afd662c0 LOCATION :
1 7.000000 6.000000 9.000000
2 4.000000 5.000000 -4.000000
3 -7.000000 -3.000000 8.000000
4 PRINTING MATRIX FROM: 0x55f7afd66340 LOCATION :
5 1.000000
6 1.000000
7 1.000000
8 PRINTING MATRIX FROM: 0x55f7afd663c0 LOCATION :
9 22.000000
10 5.000000
11 -2.000000
12 PRINTING MATRIX FROM: 0x55f7afd66440 LOCATION :
13 7.000000 6.000000 9.000000 22.000000

```



```

14 0.000000 1.571428 -9.142858 -7.571429
15 0.000000 0.000000 34.454552 34.454548
16 SOLUTIONS
17 x0 = 1.000001
18 x1 = 0.999999
19 x2 = 1.000000
20 RUNTIME: 0.000002 seconds

```

Listing I.10 – Matrix 6 results

```

0 PRINTING MATRIX FROM: 0x557fdaa552c0 LOCATION :
1 3.000000 -1.000000 0.000000
2 0.000000 3.000000 -1.000000
3 0.000000 -2.000000 3.000000
4 PRINTING MATRIX FROM: 0x557fdaa55340 LOCATION :
5 1.000000
6 1.000000
7 1.000000
8 PRINTING MATRIX FROM: 0x557fdaa553c0 LOCATION :
9 2.000000
10 2.000000
11 1.000000
12 PRINTING MATRIX FROM: 0x557fdaa55440 LOCATION :
13 3.000000 -1.000000 0.000000 2.000000
14 0.000000 3.000000 -1.000000 2.000000
15 0.000000 0.000000 2.333333 2.333333
16 SOLUTIONS
17 x0 = 1.000000
18 x1 = 1.000000
19 x2 = 1.000000
20 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.