Application en Ingénierie et Programmation Numérique

"Rendu I - Méthodes Directes"

VILLEDIEU Maxance et BESQUEUT Corentin

1 octobre 2023

Table des matières

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.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.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
*CREATE A 2D FLOAT MATRIX
*/

float** createMatrix(int row, int column){
        float **mat=NULL;
        mat=malloc(row* sizeof(int*));
```

```
        if(mat==NULL){return NULL;}
        for (int i=0; i<row; i++){
                mat[i]=malloc(column* sizeof(int));
                if(mat[i]==NULL){
                        for(int j=0; j<i; j++){
                                free(mat[j]);
                                return NULL;
                        }
                }
        }
        return mat;
}

/*
*PRINT A 2D FLOAT MATRIX
*/

void printMatrix(float **mat, int row, int column){
        for (int i=0; i<row; i++){
                for(int j=0; j<column; j++){
                        printf("%f    ", mat[i][j]);
                }
                printf("\n");
        }
}

/*
*FREE A 2D FLOAT MATRIX
*/

void freeMatrix(float **mat, int row){
        for(int i=0; i<row;i++){
                free(mat[i]);
        }
        free(mat);
}

/*
*COMPLETE A 2D FLOAT MATRIX FROM USER INPUT
*/

void completeMatrix(float **mat, int row, int column){
        for (int i=0; i<row; i++){
                for(int j=0; j<column; j++){
                        printf("Coefficient at M_%d,%d:    ", i+1, j+1);
                        scanf("%f", &mat[i][j]);
                }
        }
}

/*
*GENERATE A COLUMN VECTOR "B" FROM A 2D FLOAT MATRIX "A"
*/

void generateB(float **matA, float **matB, int row, int column){
        for(int i=0; i<row; i++){
                float sum=0;
                for(int j=0; j<column; j++){
                        sum+=matA[i][j];
                }
                matB[i][0]=sum;
        }
}

/*
```

```c
*PERFORM GAUSSIAN ELIMINATION ON A Ax=B MATRIX SYSTEM OF LINEAR EQUATIONS
*/

void gauss(float** matA, float** matb, int size){
        for(int k=0; k<size-1; k++){
                for(int i=k+1; i<size; i++){
                        float alpha=matA[i][k]/matA[k][k];
                        for(int j=k; j<size; j++){
                                matA[i][j]=matA[i][j]-alpha*matA[k][j];
                        }
                        matb[i][0]=matb[i][0]-alpha*matb[k][0];
                }
        }
}

/*
*SOLVE A MATRIX SYSTEM OF LINEAR EQUATIONS USING BACKWARD SUBSTITUTION
*/
void resolution(float** matA, float** matb, float** matx, int size){
        matx[size-1][0]=matb[size-1][0]/matA[size-1][size-1];
        for (int i=size-2; i>=0; i--){
                float sum=0;
                for(int j=i+1; j<size; j++){
                        sum+=matA[i][j]*matx[j][0];
                }
                matx[i][0]=(1/matA[i][i])*(matb[i][0]-sum);
        }
}

int main(){

        //A Matrix
        int rowA;
        int columnA;
        printf("\nRow count of matrix A : ");
        scanf("%d", &rowA);
        printf("\nColumn count of matrix A : ");
        scanf("%d", &columnA);

        float** Amatrix=createMatrix(rowA, columnA);
        completeMatrix(Amatrix, rowA, columnA);
        puts("\n                A matrix \n");
        printMatrix(Amatrix, rowA, columnA);

        //B Matrix
        float** Bmatrix=createMatrix(rowA, 1);
        generateB(Amatrix, Bmatrix, rowA, columnA);
        puts("\n                B matrix \n");
        printMatrix(Bmatrix, rowA, 1);

        //X Matrix
        float** Xmatrix=createMatrix(rowA, 1);

        //Matrix Triangularization
        puts("\n                TRIANGULARIZATION \n");
        gauss(Amatrix, Bmatrix, rowA);
        puts("\n                A Matrix \n");
        printMatrix(Amatrix, rowA, columnA);
        puts("\n                B Matrix \n");
        printMatrix(Bmatrix, rowA, 1);

        //Solve the system
        puts("\n                SOLVING \n");
        resolution(Amatrix, Bmatrix, Xmatrix, rowA);
        puts("\n                SOLUTION VECTOR X \n");
```

```
        printMatrix(Xmatrix, rowA, 1);

        //Free
        freeMatrix(Amatrix, rowA);
        freeMatrix(Bmatrix, rowA);
        freeMatrix(Xmatrix, rowA);
        return 0;
}
```

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 clefs 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 :
   $a_{1,1}, a_{1,2}, ..., a_{1,n},$ puis $a_{2,1}, ...a_{2,n},$ jusque $a_{n,1}, ..., a_{n,n}$
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 $B \in \mathcal{M}_{n,1}, \forall i \in \{1, n\}, b_{i,1} = \sum_{j=1}^{p} a_{i,p}$.
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 :

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

est représenté par l'entrée utilisateur :

Listing I.1 – User Input

```
Row count of matrix A : 3

Column count of matrix A : 3

                    FILL IN THE VALUE OF MATRIX A

Value for a_1,1:    3
Value for a_1,2:    0
Value for a_1,3:    4
Value for a_2,1:    7
Value for a_2,2:    4
Value for a_2,3:    2
Value for a_3,1:    -1
Value for a_3,2:    1
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

```
                    A matrix

3.000000     0.000000     4.000000
7.000000     4.000000     2.000000
-1.000000     1.000000     2.000000

```

```
                B matrix

7.000000
13.000000
2.000000

                TRIANGULARIZATION
                A Matrix

3.000000    0.000000    4.000000
0.000000    4.000000    -7.333333
0.000000    0.000000    5.166667

                B Matrix

7.000000
-3.333332
5.166667

                SOLVING
                SOLUTION VECTOR X

1.000000
1.000000
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

```
                A matrix

-3.000000    3.000000    -6.000000
-4.000000    7.000000    8.000000
5.000000    7.000000    -9.000000

                B matrix

-6.000000
11.000000
3.000000

                TRIANGULARIZATION
                A Matrix

-3.000000    3.000000    -6.000000
0.000000    3.000000    16.000000
0.000000    0.000000    -83.000000

                B Matrix

-6.000000
19.000000
-83.000000

                SOLVING
                SOLUTION VECTOR X

1.000000
1.000000
1.000000

```

```
Temps d'execution : 0.000250 secondes
```

Listing I.4 – $A_4X = B$ results

```
                A matrix

7.000000   6.000000   9.000000
4.000000   5.000000   -4.000000
-7.000000   -3.000000   8.000000

                B matrix

22.000000
5.000000
-2.000000

                TRIANGULARIZATION
                A Matrix

7.000000   6.000000   9.000000
0.000000   1.571428   -9.142858
0.000000   0.000000   34.454552

                B Matrix

22.000000
-7.571429
34.454548

                SOLVING
                SOLUTION VECTOR X

1.000001
0.999999
1.000000

Temps d'execution : 0.000231 secondes
```

Listing I.5 – $A_6X = B$ results

```
                A matrix

-3.000000   3.000000   -6.000000
-4.000000   7.000000   8.000000
5.000000   7.000000   -9.000000

                B matrix

-6.000000
11.000000
3.000000

                TRIANGULARIZATION
                A Matrix

-3.000000   3.000000   -6.000000
0.000000   3.000000   16.000000
0.000000   0.000000   -83.000000

                B Matrix

-6.000000
19.000000
-83.000000

                SOLVING
                SOLUTION VECTOR X

1.000000
1.000000
1.000000

Temps d'execution : 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

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

```c
 */
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];
  }
```

```c
  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.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 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 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 de ce dernier, si $\forall x_i \neq 1, \forall i \in \mathbb{N}_n$, 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 les matrices : $M = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix}$ et $X = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
sont représentés par ce fichier d'entrées :

Listing I.6 – input.txt

```
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.7 – 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 trouvée 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

```
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.9 – 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.10 – 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 obtient sur des valeurs extrêmement proches de 1. Ceci est provoqué par des erreurs d'arrondis issus par l'encodage des nombres flottants.
Nous remarquerons que l'implémentation utilisant la matrice augmentée est sensiblement meilleure, en effet le temps d'exécution est multiplié par 100 sur la premère implémentation.