### 0.0.1 Implémentation grâce à une matrice augmentée

**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 il y a une concaténation des matrices.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.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
 */
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  tre  carr e !");
    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;
  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);
  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);
    return 0;
}
```

**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 $A \in \mathcal{M}_{m,m+1}$ 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 floattant créee dynamiquement.