1-Une matrice est un tableau à deux dimensions contenant dans ses cases des nombres ou lettres

Exemple d’une supermatrice avec des nombres

On a une matrice 3*3 (3lignes, 3colonnes)

```
M[3][3] = {
    {1, 2, 3},     ligne[0]
    {4, 5, 6},     ligne[1]
    {7, 8, 9}      ligne[2]
};
```

2-Une supermatrice est une matrice dont chaque élément est lui-même une matrice. Tu peux la voir comme une matrice de sous-matrices.

Exemple

```
supermatrice sm avec ses éléments A, B, C, D chaque élément est une matrice
Sm = {
    A, B    ligne[0]
    C, D    ligne[1]
};

A = {
    1, 2
    3, 4
};

B = {
    5, 6
    7, 8
};

C = {
    9, 10
    11, 12
};

D = {
    13, 14
    15, 16
};
```

```
// sm->ligne[0][0] = A
// sm->ligne[0][1] = B
// sm->ligne[1][0] = C
// sm->ligne[1][1] = D
```

Cependant dans le fichier du prof une supermatrice ici ne contient pas des sous-matrices A, B, C, D.
Au contraire, c'est une matrice flottante (type double) améliorée, avec :

1. Un descripteur (structure avec nl, nc, et ligne)
2. Un tableau de pointeurs vers les lignes
3. Un bloc mémoire contenant tous les éléments double, rangés par lignes

Donc c'est juste une matrice mais avec gestion de la mémoire.

Le fichier supermat.h

```
typedef unsigned int iQt;
```

iQt est un alias pour unsigned int : utilisé pour les indices, tailles (lignes, colonnes).

- iQt signifie "indice Quantité"
- On a unsigned qui veut dire que les valeurs seront positives donc toutes les valeurs de types iQt seront non négatives

```c
typedef struct {
    iQt nl;       // Nombre de lignes
    iQt nc;       // Nombre de colonnes
    double **ligne; // Tableau de pointeurs vers les lignes
} *SUPERMRT;
```

iQt nl et iQt nc : stockent respectivement le **nombre de lignes** et le **nombre de colonnes** de la supermatrice.

double **ligne : tableau de pointeurs vers les lignes.

- Chaque ligne[i] est un pointeur vers un tableau double de nc colonnes.
- Cela permet d'accéder à un élément avec : ligne[i][j].

Exemple si on a une supermatrice avec les valeurs

```c
double sm[][] = {
    4.0, 5.0, 6.0, ligne[1]
    1.0, 2.0, 3.0,  ligne[0]
    7.0, 8.0, 9.0  ligne[2]
};
```

Ligne[0] pointe vers la première ligne de la matrice, ligne[1] la deuxième et ligne[2] la troisième

Ligne[0] c'est un pointeur vers la première ligne de la matrice et chaque ligne constitue un tableau

```c
1.0, 2.0, 3.0,  ligne[0]
```

Cette ligne constitue un tableau et comme ligne[0] pointe vers ce tableau pour pointer vers le premier élément de ce tableau on fera ligne[0][0]

Pour le deuxième ce sera ligne[0][1]

```c
sm->ligne[0] pointe vers {1.0, 2.0, 3.0}
sm->ligne[1] pointe vers {4.0, 5.0, 6.0}
sm->ligne[2] pointe vers {7.0, 8.0, 9.0}
```

Pourquoi un pointeur sur structure (*SUPERMRT) ?

- On veut manipuler des supermatrices **dynamiques**, donc allouées avec malloc.
- L'utilisateur manipule un **pointeur** vers la supermatrice, ce qui permet de modifier ses champs (taille, contenu) sans copie.

```
#define acces(a, i, j) (a->ligne[i][j])
```

Fournit un **accès simplifié** à l'élément en position (i,j) de la supermatrice a.

- Elle Permet d'écrire du code plus lisible et naturel :

Exemple avec la supermatrice en haut :

- acces(a, 1, 1) = 5.0; au lieu de a->ligne[1][1] = 5.0
- On sait que ligne[i] pointe vers une ligne de la supermatrice donc ligne[i][j] pointe vers la valeur de la supermatrice à la ligne i position j
- Donc a->ligne[1][1] = 5.0 permet d'accéder à la valeur de la matrice mais la macro acces(a, 1, 1) = 5.0; équivaut donc à cela pour une simplification

```
SUPERMRT aLLouerSupermat(iQt QL, iQt Qc);
```

Alloue une supermatrice de QL lignes et Qc colonnes.

Alloue :

- Le descripteur (la structure),

- Le tableau de pointeurs vers les lignes,
- Et les lignes elles-mêmes (tableaux de double).

```
SUPERMRT superProduit(SUPERMRT a, SUPERMRT b);
```

Calcule le **produit matriciel** : a × b.

Vérifie que a->nc == b->nl, sinon retourne NULL.

Alloue une nouvelle supermatrice résultat.

```
void permuterLigQes(SUPERMRT a, iQt i, iQt j);
```

Permute les **lignes i et j** de la supermatrice.

échange simplement les pointeurs ligne[i] et ligne[j].

On a par exemple cette supermatrice

```
1.0, 2.0, 3.0,  ligne[0]
4.0, 5.0, 6.0, ligne[1]
7.0, 8.0, 9.0  ligne[2]
```

On veut permuter les ligne[1] et ligne[0] par **exemple**, on aura

```
double m[][] = {
    4.0, 5.0, 6.0, ligne[0]
    1.0, 2.0, 3.0,  ligne[1]
    7.0, 8.0, 9.0  ligne[2]
};
```

C’est le but de la fonctions

```
SUPERMRT sousMatrice(SUPERMRT a, iQt L1, iQt L2, iQt c1, iQt c2);
```

Crée une **sous-matrice** allant des lignes Ll à L2 et des colonnes cl à c2.

Alloue uniquement le **descripteur et le tableau de pointeurs**, **pas les données** : réutilise les données de a

```
SUPERMRT matSupermat(double *m, iQt QLd, iQt Qcd, iQt QLe, iQt Qce);
```

Convertit une **matrice simple** (tableau 1D) en **supermatrice**.

```
void supermatMat(SUPERMRT sm, double *m, iQt QLd, iQt Qcd);
```

Convertit une **supermatrice vers une matrice simple** (tableau 1D).

Recopie les éléments de sm dans m.

```
iQt coQtiguite(SUPERMRT a);
```

- Vérifie la **contiguïté mémoire** des lignes :
    - Renvoie 2 : lignes contiguës et ordonnées.
    - Renvoie 1 : lignes contiguës mais désordonnées.
    - Renvoie 0 : lignes non contiguës.

```
void reQdreSupermat(SUPERMRT sm);
```

Libère toute la mémoire allouée :

- Chaque ligne,
- Le tableau des pointeurs,
- Le descripteur.

Le fichier supermat.c

1-aLLouerSupermat

```
SUPERMRT aLLouerSupermat(iQt QL, iQt Qc) {
```

But : allouer dynamiquement une structure SUPERMRT avec QL lignes et Qc colonnes.

Entrées :

- QL : nombre de lignes (de type iQt, un alias pour unsigned int)
- Qc : nombre de colonnes

Retour :

- Un pointeur vers une structure de type SUPERMRT allouée dynamiquement, ou NULL si l'allocation échoue.

```
SUPERMRT sm = (SUPERMRT)malloc(sizeof(*sm));
if (sm == NULL) return NULL; // Allocation failed
```

Alloue dynamiquement de la mémoire pour une **structure SUPERMRT**, c'est-à-dire un pointeur vers la structure

Vérifie que l'allocation du descripteur sm a réussi.

Si malloc a échoué → retourne NULL immédiatement.

```
sm->nl = QL;
sm->nc = Qc;
```

On **remplit les champs nl et nc** de la structure avec les dimensions demandées. C’est-à-dire les nl et nc vont prendre les valeurs de QL et Qc fournis en paramètres

```
sm->ligne = (double **)malloc(QL * sizeof(double *));
```

- Alloue un tableau de **pointeurs vers des lignes**.
- double **ligne signifie : "chaque ligne est un tableau de double".
- Il y a QL lignes, donc on réserve de la place pour QL pointeurs.

```
if (sm->ligne == NULL) {
    free(sm);
    return NULL; // Allocation échoué
}
```

Si l’allocation du tableau de pointeurs a échoué :

- On libère le descripteur sm (car il a été alloué plus tôt),
- Et on retourne NULL.

```c
for (iQt i = 0; i < QL; i++) {
    sm->ligne[i] = (double *)malloc(Qc * sizeof(double));
    if (sm->ligne[i] == NULL) {
        for (iQt j = 0; j < i; j++) free(sm->ligne[j]);
        free(sm->ligne);
        free(sm);
        return NULL; // Allocation échoué
    }
}
```

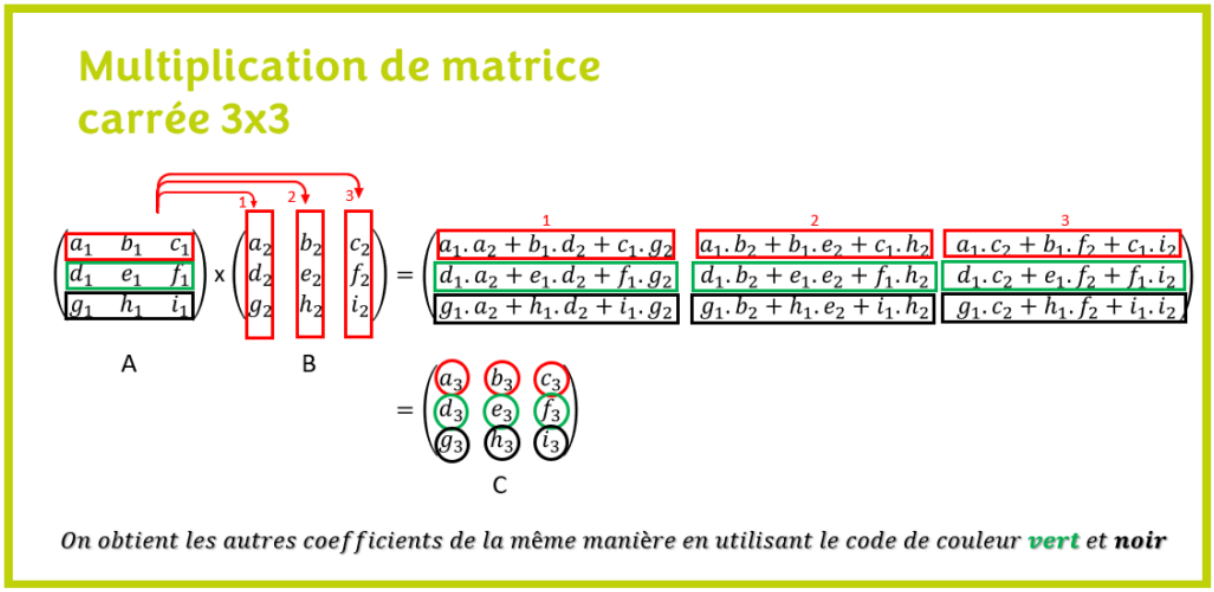
- Pour chaque ligne i de 0 à QL - 1 :
    - On alloue un tableau de Qc double → une vraie **ligne de matrice**.
    - Si l'allocation échoue à la ligne i :
        - On libère toutes les lignes déjà allouées jusqu'à i - 1
        - On libère le tableau de pointeurs sm->ligne
        - On libère le descripteur sm
        - Puis on retourne NULL

3. superProduit

```c
if (a->nc != b->nl) return NULL;
```

- On vérifie que le nombre de **colonnes de a** correspond au nombre de **lignes de b** → condition nécessaire pour la multiplication matricielle.

Exemple de multiplication d'una matrice



Les lignes de la matrice A doivent être = aux colonnes de la matrice B sinon la multiplication ne marche pas. C'est cette condition qu'on vérifie Si elle est fausse on retourne null

```
SUPERMRT c = aLLouerSupermat(a->nl, b->nc);
```

On alloue la matrice c, résultat du produit :
Elle aura autant de **lignes que a** et autant de **colonnes que b**.

```
if (!c) return NULL;
```

Et on vérifie si l'allocation a réussi ou échoué

```
for (int i = 0; i < a->nl; i++) {
    for (int j = 0; j < b->nc; j++) {
        acces(c, i, j) = 0.0;
```

- Initialisation de chaque case c[i][j] à 0.0 avant d'accumuler les produits.

```
for (int k = 0; k < a->nc; k++) {
    acces(c, i, j) += acces(a, i, k) * acces(b, k, j);
    // c[i][j] += a[i][k] * b[k][j];
}
```

Boucle sur `k` : on applique la formule :

$$c[i][j] = \sum_{k} a[i][k] \times b[k][j]$$

Exemple d'exécution de la fonction

```
/*
a = [   [1, 2],
        [3, 4] ]    → 2x2

b = [   [5, 6],
        [7, 8] ]    → 2x2

c[0][0] = 1×5 + 2×7 = 5 + 14 = 19
c[0][1] = 1×6 + 2×8 = 6 + 16 = 22
c[1][0] = 3×5 + 4×7 = 15 + 28 = 43
c[1][1] = 3×6 + 4×8 = 18 + 32 = 50

Résultat :
c = [   [19, 22],
        [43, 50] ]    → 2x2

*/
```

4. permuterLigQes

```
void permuterLigQes(SUPERMRT a, iQt i, iQt j)
```

la fonction. Elle reçoit :

- une supermatrice a,
- deux indices i et j qui désignent les lignes à échanger.

```
double *temp = a->ligne[i];
```

- On sauvegarde le pointeur de la ligne i dans une variable temporaire temp.

```
a->ligne[i] = a->ligne[j];
```

On remplace le pointeur de la ligne i par celui de la ligne j.

```
a->ligne[j] = temp;
```

Et enfin, on met dans la ligne j le pointeur initial de la ligne i (stocké dans temp).

Exemple d’exécution de la fonction

```
/*
Exemple de permutation de lignes :

On a par exemple cette supermatrice
a = [ [1.0, 2.0, 3.0],   // ligne 0
      [4.0, 5.0, 6.0],   // ligne 1
      [7.0, 8.0, 9.0] ]  // ligne 2

permuterLigQes(a, 0, 1);
On permute la ligne 0 avec la ligne 1, ce qui donne :

a = [ [4.0, 5.0, 6.0],   // ligne 0 (anciennement ligne 1)
      [1.0, 2.0, 3.0],   // ligne 1 (anciennement ligne 0)
      [7.0, 8.0, 9.0] ]  // ligne 2 (inchangée)
```

5. sousMatrice

```
SUPERMRT sousMatrice(SUPERMRT a, iQt L1, iQt L2, iQt C1, iQt C2) {
```

Extraire une **sous-matrice** à partir d'une supermatrice a, entre les lignes L1 à L2 et les colonnes C1 à C2, **sans recopier les données**.

On déclare une fonction qui retourne une nouvelle supermatrice **sub** représentant la sous-matrice de a.

```
if (L1 < 0 || L2 >= a->nl || C1 < 0 || C2 >= a->nc || L1 > L2 || C1 > C2)
    return NULL;
```

Vérification des bornes :

- Les indices L1, L2, C1, C2 doivent être **dans les limites** de la matrice a.
- L1 ≤ L2 et C1 ≤ C2 sinon on n'a pas une vraie sous-matrice.

Condition	Signification	Pourquoi c'est interdit
`L1 < 0`	Tu demandes une ligne avant la première ligne	Hors limite négative ❌
`L2 >= a->nl`	Tu demandes une ligne après la dernière ligne	Dépassement de la matrice ❌
`C1 < 0`	Tu demandes une colonne négative	Idem, pas de colonne -1 ❌
`C2 >= a->nc`	Tu dépasses le nombre de colonnes	Tu vises une colonne qui n'existe pas ❌
`L1 > L2`	Tu veux que la première ligne soit après la dernière	Logique incorrecte ❌
`C1 > C2`	Pareil avec les colonnes	Impossible à représenter ❌

Exemple pour la vérification

```
a->nl = 3;
a->nc = 3;

sousMatrice(a, 1, 4, 0, 2);
Ici L2 = 4, or a->nl = 3 ⇒ donc ligne 4 n'existe pas
Donc on va retourner NULL
```

```
SUPERMRT sub = (SUPERMRT)malloc(sizeof(*sub));
if (!sub) return NULL;
```

Allocation mémoire pour la **structure de la nouvelle supermatrice** sub.

On vérifie si l'allocation a réussi.

```
sub->nl = L2 - L1 + 1;
sub->nc = C2 - C1 + 1;
```

- On définit les dimensions de la sous-matrice :
    - Exemple : L1 = 0, L2 = 1 → 2 lignes ⇒ 1 - 0 + 1 = 2.

```
sub->ligne = (double **)malloc(sub->nl * sizeof(double *));
if (!sub->ligne) {
    free(sub);
    return NULL;
}
```

On alloue un tableau de pointeurs pour chaque ligne de la sous-matrice.

Si l'allocation échoue, on libère et on retourne NULL.

```
for (iQt i = 0; i < sub->nl; i++)
{
    sub->ligne[i] = &a->ligne[L1 + i][C1];
}
```

Chaque ligne[i] pointe directement dans a->ligne à partir de l’indice C1.

C’est une **vue** sur la matrice originale.

Exemple d’exécution de la fonction

```
/*
a = [ [1, 2, 3],    // ligne 0
      [4, 5, 6],    // ligne 1
      [7, 8, 9] ]   // ligne 2

SUPERMRT s = sousMatrice(a, 1, 2, 1, 2);

Cela veut dire :

lignes 1 à 2 → [4, 5, 6] et [7, 8, 9]
colonnes 1 à 2 → 5, 6 et 8, 9

sub = [     [5, 6],    // ligne 0
            [8, 9] ]   // ligne 1

*/
```

6. matSupermat

```
SUPERMRT matSupermat(double *m, iQt QLd, iQt Qcd, iQt QLe, iQt Qce) {
```

Paramètres :

- m : pointeur vers une **matrice classique** allouée en mémoire de façon **contiguë** (1D).
- QLd : nombre de lignes **dans la matrice d'origine** (souvent inutile ici, pas utilisé).
- Qcd : nombre de **colonnes dans la matrice d'origine** (important pour le calcul de l'offset).
- QLe : nombre de lignes qu'on veut dans la **supermatrice**.
- Qce : nombre de colonnes qu'on veut dans la **supermatrice**.

```
SUPERMRT sm = (SUPERMRT)malloc(sizeof(*sm));
if (sm == NULL) return NULL; // L'allocation a échoué
```

Allocation dynamique de la structure SUPERMRT et on Vérifie si l'allocation a échoué → retourne NULL

```
sm->nl = QLe;
sm->nc = Qce;
```

On stocke les dimensions de la supermatrice.

```
sm->ligne = (double **)malloc(QLe * sizeof(double *));
if (sm->ligne == NULL) {
    free(sm);
    return NULL; // L'allocation a échoué
}
```

On crée un **tableau de pointeurs de lignes** (chaque ligne[i] pointera vers la ligne i de m).

Si l’allocation du tableau de pointeurs échoue → on libère sm et on retourne NULL.

```
for (iQt i = 0; i < QLe; i++)
{
    sm->ligne[i] = m + i * Qcd; // Ligne i
}
```

ci, on va **recréer un accès en 2D** sur une matrice allouée de façon contiguë (tableau m).

Chaque sm->ligne[i] pointe vers le début de la i-ème ligne dans le tableau m.

Exemple d’exécution de la fonction

```
/*
double m[6] = {1, 2, 3, 4, 5, 6}; // Qcd = 3
SUPERMRT sm = matSupermat(m, ..., 2, 3);

sm->ligne[0] → m + 0 * 3 → &m[0] → [1, 2, 3]
sm->ligne[1] → m + 1 * 3 → &m[3] → [4, 5, 6]

sm[][] = {
    1, 2, 3
    4, 5, 6}

*/
```

7. supermatMat

```
void supermatMat(SUPERMRT sm, double *m, iQt QLd, iQt Qcd)
{
```

Paramètres :

- sm : pointeur vers une supermatrice (SUPERMRT).
- m : tableau contigu (1D) où l'on veut **copier** les données.
- QLd : nombre de lignes de la matrice m (pas utilisé ici).
- Qcd : nombre de colonnes dans m → utilisé pour le bon indexage.

```
for (iQt i = 0; i < sm->nl; i++)
{
```

On Parcours de toutes les **lignes de la supermatrice**.

```
for (iQt j = 0; j < sm->nc; j++)
{
```

Parcours des **colonnes de la supermatrice**.

```
m[i * Qcd + j] = acces(sm, i, j);
```

On **copie chaque élément** de la supermatrice dans la matrice.

acces(sm, i, j) accède à l'élément ligne i, colonne j de sm.

i * Qcd + j calcule l'**index dans le tableau m**.

On applique la formule pour passer **de l'index en 1D** pour un tableau 2D [ligne][colonne] :

m[i * nb_colonnes + j]

Exemple d’exécution de la fonction

```
/*
sm->nl = 2;
sm->nc = 3;
Qcd = 4;

m[0 * 4 + 0] = sm->ligne[0][0];
m[0 * 4 + 1] = sm->ligne[0][1];
m[0 * 4 + 2] = sm->ligne[0][2];

m[1 * 4 + 0] = sm->ligne[1][0];
m[1 * 4 + 1] = sm->ligne[1][1];
m[1 * 4 + 2] = sm->ligne[1][2];
```

8- coQtiguite

```
iQt coQtiguite(SUPERMRT a)
```

Retourne 1 si les lignes sont contiguës mais désordonnées.

Retourne 2 si elles sont contiguës et bien ordonnées.

Retourne 0 si elles **ne sont pas contiguës**.

```
iQt contigu = 1;
```

On **suppose d'abord** que les lignes sont contiguës mais peut-être désordonnées.

1 signifie *contigu mais pas nécessairement dans l'ordre*.

```
for (iQt i = 1; i < a->nl; i++)
{
```

- On commence à comparer **chaque ligne** à la précédente, dès la deuxième ligne.

```
if (a->ligne[i] == a->ligne[i - 1] + a->nc)
{
    continue; // Toujours dans l'ordre attendu
}
```

Si ligne[i] est **juste après** ligne[i - 1], alors :

- Les **lignes sont bien ordonnées en mémoire**, l'une à la suite de l'autre.
- On continue simplement.

```
else if (a->ligne[i] > a->ligne[0] && (a->ligne[i] - a->ligne[0]) % a->nc == 0)
{
    contigu = 1; // Contigu mais hors de l'ordre attendu
}
```

- Sinon, on vérifie une autre forme de contiguïté :
    - Est-ce que ligne[i] appartient **au même bloc mémoire** que ligne[0] ?
    - Et est-elle bien **alignée par rapport au nombre de colonnes** ?

- Si oui → C’est contigu, mais **pas dans l’ordre**.

Donc on garde contigu = 1, mais on ne passe pas à 2

```
else
{
    return 0; // Pas contigu du tout
}
```

Si aucune des deux conditions n’est remplie → **les lignes ne sont pas contiguës du tout**.

```
return contigu == 1 ? 1 : 2;
```

Si on n’a détecté **aucun désordre**, et que toutes les lignes étaient parfaitement ordonnées :

- contigu reste à sa valeur initiale (1), mais comme on ne l’a jamais passé à 2, on retourne 2.

Sinon, on retourne 1 (contigu mais pas trié).

Exemple d’exécution de la fonction

```
/*
Cas 1 - Contigu et ordonné :
a->ligne[0] = m
a->ligne[1] = m + nc
a->ligne[2] = m + 2 * nc
On retourne 2 car Contigu en mémoire, dans l'ordre

Cas 2 - Contigu mais désordonné :
a->ligne[0] = m + 2 * nc
a->ligne[1] = m
a->ligne[2] = m + nc
On retourne 1 car Contigu en mémoire, mais dans un ordre différent

Cas 3 - Pas contigu :
a->ligne[0] = malloc(nc * sizeof(double));
a->ligne[1] = malloc(nc * sizeof(double));
On retourne 0 car Chaque ligne vient d’un bloc mémoire séparé
```

9-reQdreSupermat

```
if (sm)
{
```

Avant toute chose, on vérifie que le pointeur sm n’est **pas NULL**.

```
if (sm->ligne)
{
```

On vérifie maintenant que le tableau sm->ligne a bien été alloué.

Ce tableau contient les **pointeurs vers les lignes** de la supermatrice.

```
free(sm->ligne[0]);
```

Ici, on libère le **bloc mémoire principal** de données (les vraies valeurs double de la matrice).

On suppose que toutes les lignes pointent vers le **même bloc mémoire contigu** (comme dans matSupermat), donc **un seul free suffit**.

```
free(sm->ligne);
```

Ensuite, on libère le **tableau de pointeurs** vers les lignes.

```
free(sm);
```

Enfin, on libère la **structure principale elle-même** (SUPERMRT), qui contient le nombre de lignes nl, de colonnes nc,et le pointeur ligne

Fichier main.c

```c
void afficherSupermat(SUPERMRT sm) {
    for (iQt i = 0; i < sm->nl; i++) {
        for (iQt j = 0; j < sm->nc; j++) {
            printf("%.2f ", acces(sm, i, j)); //sm[i][j]
        }
        printf("\n");
    }
}
```

- On parcours les ligne puis les colonnes
- Affiche chaque élément de la supermatrice en utilisant la fonction acces (probablement un macro ou une fonction pour accéder à sm->ligne[i][j]).
- Les éléments sont affichés avec 2 décimales (%.2f).

```c
iQt ligne = 3, colonne = 3;  //3*3
SUPERMRT A = aLLouerSupermat(ligne, colonne);
```

- On crée une supermatrice de 3 lignes et 3 colonnes (3x3).
- aLLouerSupermat alloue dynamiquement la matrice.

```c
if (A == NULL) {
    printf("Échec d'allocation de la matrice A\n");
    return 1;
}
```

Si l'allocation échoue, on arrête le programme.

```
printf("Matrice A :\n");
for (iQt i = 0; i < ligne; i++) {
    for (iQt j = 0; j < colonne; j++) {
        acces(A, i, j) = (i + 1) * (j + 1);
    }
}
afficherSupermat(A);
```

On Remplit la matrice avec les produits ligne * colonne.

Exemple : position [0][0] = (1)*(1) = 1*,

[1][2] = (2)(3) = 6

```
SUPERMRT B = aLLouerSupermat(colonne, ligne);
if (B == NULL) {
    printf("Échec d'allocation de la matrice B\n");
    reQdreSupermat(A);
    return 1;
}
```

```
printf("\nMatrice B :\n");
for (iQt i = 0; i < colonne; i++) {
    for (iQt j = 0; j < ligne; j++) {
        acces(B, i, j) = (i + 1) + (j + 1);
    }
}
afficherSupermat(B);
```

B est une autre matrice (3x3) mais transposée pour le produit.

Chaque élément est la somme des indices + 1 (par ex, [0][1] = 1+2 = 3).

```
SUPERMRT C = superProduit(A, B);
if (C == NULL) {
    printf("Échec du produit matriciel.\n");
} else {
    printf("\nProduit A * B :\n");
    afficherSupermat(C);
    reQdreSupermat(C);
}
```

On crée une supermatrice C on vérifie si elle est nulle on la soumet à la fonction qui gère le produit et on l'affiche

```
// Permutation de lignes
printf("\nPermutation des lignes 0 et 1 dans A :\n");
permuterLigQes(A, 0, 1);
afficherSupermat(A);
```

- On échange les lignes 0 et 1 de A.

Eton affiche

```
// Extraction d'une sous-matrice
printf("\nSous-matrice de A (0,1) -> (1,2) :\n");
SUPERMRT subA = sousMatrice(A, 0, 1, 0, 2);
if (subA) {
    afficherSupermat(subA);
    reQdreSupermat(subA);
} else {
    printf("Échec de l'extraction de sous-matrice\n");
}
```

- Extraction d'un bloc de A : lignes [0 à 1], colonnes [0 à 2].

```c
// Vérification de la contiguïté
printf("\nVérification de la contiguïté de A : %d\n", coQtiguite(A));
```

Vérifie si les lignes sont contiguës en mémoire (pour savoir si on peut la manipuler comme une matrice simple).

```c
double matrice[] = {
    1.0, 2.0, 3.0,
    4.0, 5.0, 6.0,
    7.0, 8.0, 9.0
};
```

On déclare une matrice simple qui va servir dans conversion matrice vers supermatrice

```c
// Conversion matrice → supermatrice
SUPERMRT SM = matSupermat(matrice, 3, 3, 3, 3);
if (SM == NULL) {
    printf("Échec de la conversion matrice -> supermatrice\n");
} else {
    printf("Supermatrice SM (à partir d'un tableau) :\n");
    afficherSupermat(SM);
}
```

```c
// Libération mémoire
reQdreSupermat(SM);
reQdreSupermat(A);
reQdreSupermat(B);
```

Libère proprement toute la mémoire allouée dynamiquement pour éviter les fuites mémoire.