Les tableaux 2D
Les tableaux multidimensionnels

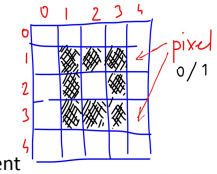
Les tableaux 2D et quelques traitements classiques
Généralisation aux tableaux multidimensionnels

Les tableaux 2D

et quelques traitements classiques

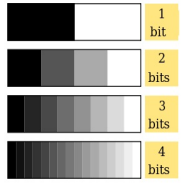
Tableaux 2D et des images

Représenter une image



Une image = quadrillage de pixels
- pixel blanc ou noir : un bit 0 ou 1, respectivement

- niveaux de gris :
  un entier
  niveaux/nuances : variation de l'entier
  entre 0 et 255 (format PGM : foncé -> clair)



- couleurs :
  . RGB (rouge/vert/bleu)
  . G+B = cyan (bleu mers du sud) , R+B = magenta (violet), R+G = jaune
  . triplet d'entiers RGB pour les nuances

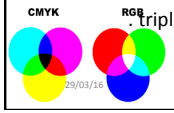
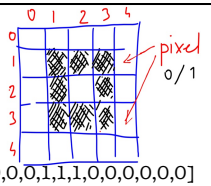


Image = tableau 2D de pixels



Avec un tableau 1D :

000000111001010000111000000

t : tableau[25] d'entiers = [0,0,0,0,0,0,1,1,1,0,0,1,0,1,0,0,0,1,1,1,0,0,0,0,0,0]

Avec un tableau 2D :

N/B	niveaux de gris
00000	0 0 0 0 0
01110	0 128 128 128 0
01010	0 128 0 128 0
01110	0 128 128 128 0
00000	0 0 0 0 0

t : tableau[5,5] d'entiers = [[0,0,0,0,0] , [0,1,1,1,0], [0,1,0,1,0], [0,1,1,1,0] , [0,0,0,0,0]]

- **2 indices :** ligne / colonne
- **t[1, 2] == 1** et **t[4,4] == 0**
- **t[3, 10] :** tableau rectangulaire de 3 lignes et 10 colonnes

L'ami du tableau 2D : la double-boucle imbriquée

Un premier exemple

```
declare
    img : tableau[5,8] d'entiers      // une image rectangulaire
                                      // de 5 lignes, 8 colonnes
    i  : entier                       // indice de parcours des lignes
    j  : entier                       // indice de parcours des colonnes

debut
    pour i de 0 à 4 faire       // pour chaque ligne : il y en a 5
        pour j de 0 à 7 faire           // pour chaque colonne : il y en a 8
            img[i , j] = 128                  // un pixel de gris
        finpour                         // j'ai fini toutes les colonnes donc une
ligne
    finpour                     // j'ai fini toutes les lignes donc l'image ....
fin                             //... comporte 40 pixels gris
```


Des algorithmes sur les tableaux 2D

D'abord bien comprendre la double boucle imbriquée

Bien comprendre la double-boucle imbriquée

Etape 1 : indices indépendants

t : tableau[2, 5] d'entiers = *initialisé comme celui-ci* →

1	-3	5	-7	9
-2	4	-6	8	-10

```
declare
    t : tableau[2, 5] d'entiers = ...
    i  , j  : entier
debut
    pour i de 0 à 1 faire
        pour j de 0 à 4 faire

            afficher( t[i , j])

        finpour
        afficher (sdl) // saut de ligne

    finpour
fin
```

L'instruction d'affichage n'est pas modifiée !!

```
declare
    t : tableau[2, 5] d'entiers = ... i  ,
j  : entier
debut
    pour j de 0 à 4 faire
        pour i de 0 à 1 faire
            afficher( t[i , j])
        finpour
        afficher (sdl) // saut de ligne

    finpour
fin
```

Parcours "en ligne d'abord" : indice i, puis indice j

```
1 -3 5 -7 9
-2 4 -6 8 -10
```

Seul le parcours ligne/colonne du tableau est permuté

Parcours "en colonne d'abord" :

```
1 -2
-3 4
5 -6
-7 8
9 -10
```

Bien comprendre la double-boucle imbriquée

Etape 2 : indices liés

t : tableau[4, 5] d'entiers = *initialisé comme celui-ci* →

1	-3	5	-7	9
-2	4	-6	8	-10
11	13	15	17	19
12	14	16	18	20

```
declare
      t : tableau[4, 5] d'entiers = ...
      i , j  : entier
debut
      pour i de 0 à 3 faire
            pour j de i à 4 faire
                  afficher( t[i , j])

            finpour
            afficher (sdl) // saut de ligne

      finpour

fin
```

La boucle intérieure dépend de la boucle extérieure

Les valeurs prises par j dépendent de celles de i. Elles changent donc chaque fois que i change.

```
1 -3  5 -7  9
4 -6  8 -10
15 17 19
18 20
```

Les indices varient comme suit :

i = 0
j= 0,1,2,3,4
i =1
j= 1,2,3,4
i =2
j= 2,3,4
i =3
j= 3,4

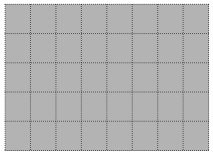
Des algorithmes sur les images

Exemples en interaction : tableaux 2D et images

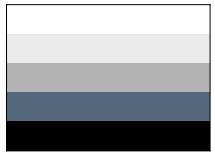
Tout gris c'est parfois triste ...

```
declare
      img : tableau[5,8] d'entiers      // une image rectangulaire
                                        // de 5 lignes, 8 colonnes
      i  : entier                       // indice de parcours des lignes
      j  : entier                       // indice de parcours des colonnes

debut
      pour i de 0 à 4 faire         // pour chaque ligne : il y en a 5
            pour j de 0 à 7 faire         // pour chaque colonne : il y en a 8
                  img[i , j] = 128              // un pixel de gris
            finpour                 // j'ai fini toutes les colonnes donc une ligne
      finpour                       // j'ai fini toutes les lignes donc l'image ....
fin                                 //... comporte 40 pixels gris
```

L'algo ci-contre donne :

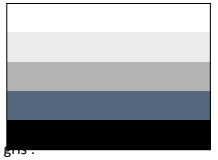


Modifions l'algo de création de l'image grise pour obtenir ça →



Tout gris c'est parfois triste ...

Modifions l'algo de création de l'image grise pour obtenir ça →

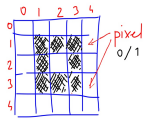


255
192
128
64
0

valeurs des gris

1. Principe de l'algorithme de remplissage en 5 niveaux de gris :
- remplir chaque ligne du même niveau de gris
- 2 choix symétriques :

Attention : les images sont souvent indicées "à l'inverse" des matrices
. du haut vers le bas : ligne 0 → ligne 4
. du bas vers le haut : ligne 4 → ligne 0

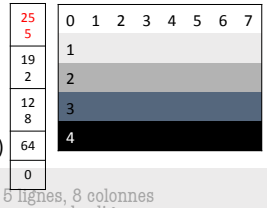


2. Petite difficulté arithmétique : comment partager la plage de gris 0..255 en 5 gris répartis de façon équilibrée ?

Essayons avec les 8 valeurs 0 1 2 3 4 5 6 7
. on prend les 2 extrêmes : 0, 7 (noir, blanc) 0 1 2 3 4 5 6 7
. il reste 5 valeurs réparties de 1 à 6 et on veut 3 autres valeurs de séparation
. choix équidistant impossible → choix arbitraire : par exemple 2, 4, 6 qui sont équidistants depuis 0
. ce qui donne 0 2 4 6 7, soit 0+2, 0+2+2, 0+2+2+2, ... soit : 0 + i x 2 avec i = 0,1,2,3
. et on rajoute 7 à la fin.

Plus généralement, difficile de partager un nombre pair de valeurs en un nombre impair (>1) de tas ...

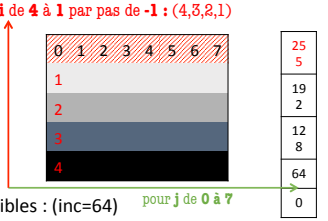
Tout gris c'est parfois triste ...



Un choix (parmi d'autres) :
. on part d'en bas en ajoutant 64 à chaque fois
. on traite à part la première ligne (celle d'en haut !)

```
declare
    img : tableau[5,8] d'entiers          // une image 5 lignes, 8 colonnes
    i  : entier                           // indice de parcours des lignes
    j  : entier                           // indice de parcours des colonnes
    inc : constante entier = 64           // incrément de gris : 64 ici
debut
    pour i de 4 à 1 par pas de -1 faire   // pour chaque ligne en partant du bas sauf la "dernière"
        pour j de 0 à 7 faire             // ne dépend pas de j
            img[i, j] = (4 – i)* inc      // numéro de ligne en partant du bas  x incrément de gris
        finpour                           // j'ai fini toutes les colonnes donc une ligne
    finpour                               // j'ai fini de griser les 4 lignes en partant du bas

    pour j de 0 à 7 faire
        img[0, j] = 255                   // je fini avec la première ligne (i=0) de blanc
    finpour
fin
```


Tout gris c'est parfois triste ...




```
pour i de 4 à 1 par pas de -1 faire
    pour j de 0 à 7 faire
        img[i, j] = (4 – i)* inc
    finpour
finpour
```

D'autres écritures du corps de boucle sont possibles : (inc=64)
. (4 – i)*inc vaut successivement (4-4)*64, (4-3)*64, (4-2)*64, (4-1)*64 : ok
. 256 – i*inc vaut successivement 256-(4*64), 256-(3*64), 256-(2*64), 256-(1*64): ok

Autre sol. avec une addition sur inc
. img[i,j] = img[i+1, j]+inc
donne les mêmes itérations
après avoir initialisé img[4, j] = 0

```
// initialisation ligne 4
pour j de 0 à 7 faire
    img[4, j] = 0
finpour
```

```
// griser les lignes 3,2,1
pour i de 3 à 1 par pas de -1 faire
    pour j de 0 à 7 faire
        img[i, j] = img[i+1, j] + inc
    finpour
finpour
```

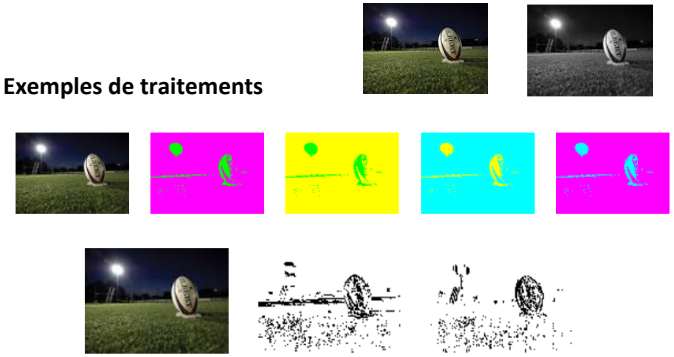
D'autres boucles (et corps) sont bien sûr aussi possible :

```
pour i de 1 à 4 faire  // haut -> bas
    pour j de 0 à 7 faire
        img[i, j] = 256 – i* inc
    finpour
finpour
```


Quelques exercices sur des images

- Générer le négatif (*reverse vidéo*) d'une image NB ou par niveau de gris
- Générer une image NB à partir d'une image niveau de gris
- Générer une image miroir vertical ou horizontal d'une image NB
- Augmenter le contraste d'une image à niveau de gris
  Principe : fixer un seuil et remplacer les pixels plus clairs que le seuil par des pixels blancs, et inversement les plus sombres que le seuil par des pixels noirs.
- Augmenter la luminance d'une image à niveau de gris
  Principe : ajouter ou retrancher une constante de la valeur des pixels.
- Générer les contours significatifs d'une image
  Principe : on remplace par un pixel noir chaque pixel dont la variation des valeurs de ses voisins varient au delà d'un certain seuil, sinon on le remplace par un pixel blanc
- Réduire par 2 la taille d'une image à niveau de gris
  Principe : chaque carré de 2x2 pixels est remplacé par 1 pixel de valeur la moyenne des pixels du carré

Exemples de traitements



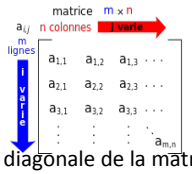
Des algorithmes sur les tableaux 2D

Suite d'exemples en interaction : les matrices

Matrices

Vocabulaire

- matrice = tableau 2D de nombres (entiers, flottants), de booléens
- matrice carrée = autant de lignes que de colonnes



$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{pmatrix} 2 & 4 & 6 \\ 4 & 0 & 10 \\ 6 & 10 & 12 \end{pmatrix}$$

- la diagonale de la matrice A : A[i,i] → c'est un vecteur
- une matrice (carrée) diagonale : A[i, j] = 0 pour tous les i ≠ j
- la matrice (diagonale) identité I ou `Id(n)` = des 1 sur la diagonale, des 0 ailleurs
- une matrice (carrée) symétrique : A[i,j] = A[j,i]
- ...

Matrices et premiers algorithmes

Vérifier si une matrice donnée est ou n'est pas d'une certaine forme :

- diagonale ? symétrique ? égale à l'identité ? l'inverse d'une autre matrice donnée ?...

fonction à réponse booléenne

Générer une matrice ou une forme matricielle particulière à partir d'une (ou plusieurs) matrice(s) donnée(s) :

- matrice transposée
- matrice symétrique à partir de la partie triangulaire supérieure
- arithmétique matricielle : A+B, AxB, Axu, ...

fonction/procédure à réponse numérique/matricielle

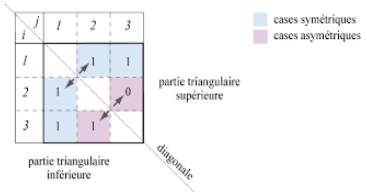
Vérifier si une matrice donnée est symétrique ou non ?

Un premier problème : vérifier

Vérifier si une matrice donnée est symétrique ou non ?

Entrée : une matrice (carrée) A
Sortie : une réponse booléenne

Principe
- parcourir chaque valeur A[i,j]
- la comparer à A[j,i]
- si égalité, on continue
- sinon la réponse est connue

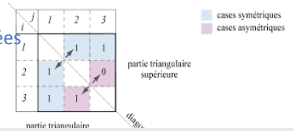


Premières solutions à base de boucles pour imbriquées
- choix 1 : parcourir toute la matrice
- choix 2 : parcourir une moitié de la matrice : triangulaire supérieure

Erreurs possibles
- l'entrée n'est pas une matrice carrée

Vérifier si une matrice donnée est symétrique ou non ?

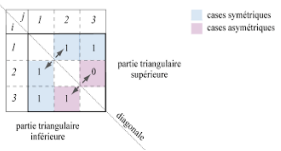
Premières solutions à base de boucles pour imbriquées
- choix 1 : parcourir toute la matrice




```
declare
    n : constante entier = ...          // paramètre taille de l'une matrice carrée
    A : tableau[n, n] d'entiers = ...   // une matrice carrée 4x4 initialisée ...
    rep : booléen = VRAI          // la réponse : vrai ou faux
    i, j : entier                       // les indices de parcours ligne-colonne
debut
    pour i de 0 à n-1 faire             // pour chaque ligne
        pour j de 0 à n-1 faire         // pour chaque colonne
            si A[i , j] != A[j, i] alors    // test de non-symétrie
                rep = FAUX
            finsi
        finpour                         // j'ai fini toutes les colonnes
    finpour                             // j'ai fini toutes les lignes
fin
```


Vérifier si une matrice donnée est symétrique ou non ?

Premières solutions à base de boucles pour imbriquées
- choix 1 : parcourir toute la matrice




```
declare
    n : constante entier = ...
    A: tableau[n, n] d'entiers      = ...
    rep : booléen = VRAI
    i, j : entier
debut
    pour i de 0 à n-1 faire
        pour j de 0 à n-1 faire
            si A[i , j] != A[j, i] alors
                rep = FAUX
            finsi
        finpour
    finpour
fin
```

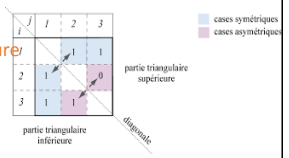
Cet algorithme n'oublie rien
- double boucle imbriquée → n^2 tests internes

mais n'est pas optimal
- n tests inutiles :
  la diagonale → i=j : A[i, i] vs. A[i, i]
- n/2 test inutiles :
  A[i, j] vs. A[j, i] puis A[j, i] vs. A[i, j]

Amélioration
ne parcourir que la partie triangulaire supérieure

Vérifier si une matrice donnée est symétrique ou non ?

Premières solutions à base de boucles pour imbriquées
- choix 2 : ne parcourir que la partie triangulaire supérieure
  A[i, j] avec i < j




```
declare
    n : constante entier = ...
    A : tableau[n, n] d'entiers     = ...
    rep : booléen = VRAI
    i, j : entier
debut
    pour i de 0 à n-1 faire
        pour j de i+1 à n-1 faire
            si A[i , j] != A[j, i] alors
                rep = FAUX
            finsi
        finpour
    finpour
fin
```

Analyse de l'amélioration
- **oui** : parcours de la partie triangulaire supérieure

- **non optimal encore** :
si A[1,2] ≠ A[2,1] et taille n grande ...
beaucoup de tests/d'itérations inutiles

Exercice : écrire un algorithme qui réduit le nombre d'itérations/comparaisons

Vérifier si une matrice donnée est symétrique ou non ?

L'écrire sous forme d'une fonction

Entrée : une matrice (carrée) A, sa taille n
Sortie : une réponse booléenne

~~fonction VerifSym(M: tableau d'entiers, n: entier) retourne booléen~~
~~fonction VerifSym(M: tableau[] d'entiers, n: entier) retourne booléen~~

fonction VerifSym(M: tableau[n,n] d'entiers, n: entier) retourne booléen

fonction VerifSym(n: entier, M: tableau[n,n] d'entiers) retourne booléen

Erreurs possibles
Si le paramètre effectif n'est pas une matrice carrée :
- contrôler la taille si elle n'est pas un paramètre formel de la fonction,
- puis arrêter le traitement si besoin.
. ce n'est pas le cas ici.

```
declare
   n : constante entier = ...
   A: tableau[n, n] d'entiers = ...
   rep : booléen = VRAI
   i, j : entier
debut
   pour i de 0 à n-1 faire

      pour j de i+1 à n-1 faire

         si A[i , j] != A[j , i] alors
            rep = FAUX
         finsi
      finpour
   finpour
fin
```


Sous forme de fonction

En-tête

```
fonction VerifSym(n: entier, M: tableau[n,n] d'entiers) retourne booléen
```

Corps

```
fonction VerifSym(n: entier, M: tableau[n,n] d'entiers) retourne booléen
  rep : booléen = VRAI
  i, j : entier
debut
  pour i de 0 à n-1 faire
    pour j de i+1 à n-1 faire
      si M[i , j] != M [j , i] alors
        rep = FAUX
      finsi
    finpour
  finpour
  retourne rep
fin
```


Transposer une matrice donnée

Un deuxième problème : générer

Transposer une matrice donnée

Entrée : une matrice A
Sortie : une matrice ^tA

$$A = \begin{pmatrix} 1 & 1 \\ 2 & 3 \\ 1 & -1 \end{pmatrix} \quad , \quad {}^tA = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & -1 \end{pmatrix} .$$

On suppose A carrée

Principe ... trop naïf pour être juste

- parcourir chaque valeur A[i,j] → double boucle pour imbriquée
- échanger A[i, j] et A[j, i]

Exemple

- à dérouler sur la matrice A
- et ...

Proposer un algorithme correct.

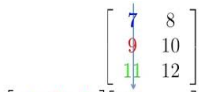
```
declare
    A: tableau[3, 3] d'entiers = [ [1, 1, 1], [0, 2, 3], [2, 1, -1]]
    i, j, tmp : entier
debut
    pour i de 0 à 2 faire
        pour j de 0 à 2 faire
            tmp = A[i, j]                    // échange classique
            A[i, j] = A[j, i]
            A[j, i] = tmp
        finpour
    finpour
fin
```

Calculer un produit matrice x matrice

Un troisième problème : calculer

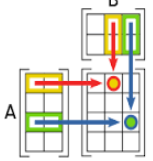
Calculer un produit matrice x matrice

Entrée : 2 matrices A[m, n] et B[n,p]
Sortie : une matrice C[m, p]



Principe sur un produit de matrices carrées de taille n : C=A*B
- calculer la matrice produit C = calculer ses n^2 coefficients: $1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58$
  → C[i, j] pour i=0,n-1 et j=0,n-1
  → **double boucle pour imbriquée sur i (ligne de C) et j (colonne de C)**

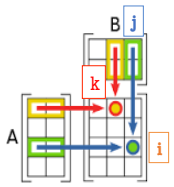
- pour chaque couple (i,j), calculer la valeur du coefficient C[i,j] :
  → C[i, j] = Σ A[i, k] * B[k, j]
  = A[i, 0]*B[0, j] + A[i, 1]*B[1, j] + A[i, 2]*B[2, j] + ...
  + ... + A[i, k]*B[k, j] + ... + A[i, n-1]*B[n-1, j]
  → **boucle pour sur k (une ligne de A, une colonne de B)**



Conclusion
- **triple boucle pour imbriquée**, sur i,j (parcourir C), k (parcourir A et B)

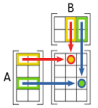
Calculer un produit matrice x matrice

```
declare
      A: tableau[4, 2] d'entiers = [ [1, 1], [2, 3], [1, -1], [0,-2] ]
      B: tableau[2, 3] d'entiers = [ [1, 2, 3], [-2, 0,-4] ]
      C: tableau[4, 3] d'entiers
      i, j, k : entier // itérateurs

debut
      // C est mise à zéro
      pour i de 0 à 3 faire                  // exemple d'initialisation
            pour j de 0 à 2 faire   // Bien voir pourquoi
                  C[i, j] = 0             // la mise à zéro est nécessaire !
            finpour
      finpour

      // calcul de C=A*B
      pour i de 0 à 3 faire                  // ligne i
            pour j de 0 à 2 faire   // colonne j
                  pour k de 0 à 1    // parcours k
                        C[i, j] = C[i,j] + A[i,k]*B[k,j]   // vu ?
                  finpour
            finpour
      finpour
fin
```



Question
Autre solution pour initialiser C[i, j] avant l'accumulation de la boucle sur k ?

Calculer un produit matrice x matrice



Exercice facile pour se détendre
- expliciter la suites des indices i,j,k en déroulant le produit matrice x matrice ci-contre.

Remarques
- l'ordre du parcours de C est arbitraire
→ les boucles "i" et "j" peuvent être permutées

- l'ordre de l'accumulation dans C[i,j] est aussi arbitraire (au moins dans IR)
→ la boucle sur "k" peut être aussi permutée avec celles sur "i" ou "j"

6 ordres possibles pour ce produit
i-j-k, i-k-j, j-i-k, k-i-j, k-j-i, j-k-i

```
declare
      A: tableau[4, 2]  d'entiers = ....
      B: tableau[2, 3]  d'entiers = ....
      C : tableau[4, 3] d'entiers =
      i, j, k : entier
debut
      // C est mise à zéro
      pour i de 0 à 3 faire
            pour j de 0 à 2 faire
                  C[i, j] = 0
            finpour
      finpour

      // calcul de C=A*B
      pour i de 0 à 3 faire
            pour j de 0 à 2 faire
                  pour k de 0 à 1
                        C[i, j] = C[i,j] + A[i,k]*B[k,j]
                  finpour
            finpour
      finpour
fin
```

Calculer un produit matrice x matrice

L'écrire sous forme d'une fonction

Entrées : une matrice A de taille m x n et une matrice B de taille n x p (de flottants)
Sortie : une matrice C de taille m x p (de flottants)

Problèmes de convention
- une fonction peut retourner un tableau : oui/non ?
  → ce qui implique que l'affectation A = B est autorisée pour A et B,
  Déjà : il faut 2 tableaux de même type d'éléments et de même taille ...

- si oui, faut-il préciser la taille du tableau résultat : oui/non ?
  fonction Produit(....) retourne tableau de flottants
  fonction Produit(....) retourne tableau[m,p] de flottants

- sinon, une procédure convient.

Calculer un produit matrice x matrice

L'écrire sous forme d'une procédure

Entrées : une matrice A de taille m x n et une matrice B de taille n x p
Sortie : une matrice C de taille m x p

```
procedure Produit(m: in entier, n: in entier, p: in entier
                  A: in tableau[m,n] de flottants,
                  B: in tableau[n,p] de flottants,
                  C: inout tableau[m,p] de flottants)
```

Erreurs possibles
Si les paramètres effectifs ne vérifient pas (m, n) x (n, p) → (m, p)
- contrôler les tailles des paramètres effectifs,
- puis, arrêter le traitement si besoin

Procédures et tableaux 2D

Appelant

```
declare
   M: tableau[4,2] de flottants
   N: tableau[2,3] de flottants
   P: tableau[4,3] de flottants

avec  // en-tête des sous-programmes externes utilisés
   procedure Lire(m: in entier, n: in entier, A : inout tableau[m,n] de flottants)
   procedure Afficher(m: in entier, n: in entier, A : in tableau[m,n] de flottants)
   procedure Produit(      m: in entier, n: in entier, p: in entier,
                           A: in tableau[m,n] de flottants, B : in tableau[n,p] de
flottants,
                           C: inout tableau[m,p] de flottants)

debut
   Lire(4, 2, M)
   Lire(2, 3, N)                    // seules M et N sont initialisées. OK ?
   Produit(4, 2, 3, M, N, P)
   Afficher(4, 3, P)
fin
```


Procédures et tableaux 2D

Sous programmes externes : lecture et affichage de matrices

```
procedure Lire(m: in entier, n: in entier, A : inout tableau[m,n] de flottants)
   i, j : entier
debut
   pour i de 0 à m-1 faire
         pour j de 0 à n-1 faire
               Lire(A[i, j])
         finpour
   finpour
fin procedure


procedure Afficher(m: in entier, n: in entier, A : in tableau[m,n] de flottants)
   i, j : entier
debut
   pour i de 0 à m-1 faire
         pour j de 0 à n-1 faire
               Afficher(A[i, j])
         finpour
         Afficher(sdl)    // saut de ligne quand il faut
   finpour
fin procedure
```

Procédures et tableaux 2D

Procédure de produit matrice x matrice

```
procedure Produit(m: in entier, n: in entier, p: in entier,
                  A: in tableau[m,n] de flottants, B : in tableau[n,p] de flottants,
                  C: inout tableau[m,p] de flottants)
  i, j, k : entier
debut

  pour i de 0 à m-1 faire
      pour j de 0 à p-1 faire
          C[i, j] = 0.0              // autre façon d'initialiser C[i,j] avant
l'accumulation
          pour k de 0 à n-1 faire
              C[i, j] = C[i, j] + A[i, k]*B[k, j]
          finpour
      finpour
  finpour
fin procedure
```


Un problème de synthèse

Vérifier qu'une matrice donnée est l'inverse ou non d'une autre matrice aussi donnée

Vérifier que deux matrices données sont l'inverse l'une de l'autre

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Le peu de math nécessaire

- Déf.: La matrice A est l'inverse de la matrice B ssi A*B = B*A = I : matrice identité
- Conséquence : le produit A*B doit commuter
  donc A et B sont carrées de même taille, celle de I

Mise en place de l'algo

- Entrées : deux matrices A et B carrées de taille n
- Sortie : un booléen

→ donc peut se présenter sous forme d'une fonction

```
fonction VerifInverse ( n: in entier,
                        A: in tableau[n,n] de flottants,
                        B: in tableau[n,n] de flottants)  retourne booléen
```

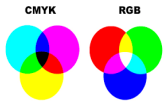
- Principe du traitement
  . effectuer le produit A*B (ou B*A)
  . comparer la matrice résultat avec I[n,n]

Quelques remarques avant de conclure

Généralisation aux tableaux 3D et plus

Aspects pratiques

Retour sur les images en couleurs



Une image = quadrillage de pixels couleur
img : tableau[1024,768] de ... pixels couleurs

1 pixel couleur = triplet d'entiers RGB
exemple : [255, 0, 0] = [255,255,0] = [127,127,127]=
pixel : tableau[3] d'entiers

Une solution : le tableau 3D
imgCouleur : tableau [1024, 768, 3] d'entiers
imgCouleur[i, j, 0] → composante R du pixel [i,j]
imgCouleur[i, j, 2] → composante B du pixel [i,j]
- 3 indices : initialisation/lecture/affichage avec triple boucle pour imbriquées

Remarques sur les tableaux multidimensionnels

En pratique, les tableaux nD sont représentés comme des vecteurs
- déroulage/dépliage selon les lignes OU les colonnes

1	2	3
4	5	6

- ligne vs. colonnes : dépend du langage de programmation

C : par ligne

1	2	3	4	5	6

Fortran : par colonne

1	4	2	5	3	6

Les indices multiples sont une commodité de notation → s'en servir !!!!

Ce qui reste à présenter sur les tableaux

Vu : tableau statique
la taille du tableau est fixé une fois pour toute lors de sa déclaration
(ce qui ne veut pas dire qu'il soit initialisé)
on peut être amené à réserver plus de place que nécessaire

Pas vu : tableau dynamique ou redimensionnable
la taille du tableau n'est pas précisée lors de sa déclaration
mais il existe une fonction spéciale pour fixer la taille de la mémoire nécessaire au stockage du contenu du tableau
→ allocation dynamique de mémoire
(et une autre fonction pour libérer cette "place mémoire" si on n'en n'a plus besoin)

En pratique, ces tableaux ne sont pas stockés dans la même zone mémoire

Ce qui reste à présenter sur les tableaux

Vu : tableau
un tableau regroupe des objets de même type
ses éléments sont accédés par les indices

Pas vu : enregistrement
structure de données qui regroupe des valeurs de type différents
chaque valeur est appelé un champ
un champ est accédé par son nom et une notation pointée (souvent)

Enregistrements

```
declare
    enregistrement Etudiant
        NumEtud: entier
        Nom: tableau[55] de caractères
        Prenom: tableau[55] de caractères
    finenregistrement

    Moi, MonVoisin: Etudiant

debut
    Moi.NumEtudiant = 432165
    MonVoisin.Nom = "Machin"
fin
```

```
declare
    enregistrement pixelCouleur
        R: entier = 0     // valeur par défaut
        G: entier = 0
        B: entier = 0
    finenregistrement

    p: pixelCouleur = [255, 255, 0]
    // on peut convenir d'autres écritures
    // d'affectations

    ImgCouleur: tableau[512, 384] de pixelCouleur

debut
    si ImgCouleur.R ==  p.G alors
        ...
    finsi
fin
```


Ce qui reste à présenter sur les tableaux

Vu : tableau

les indices sont des entiers consécutifs entre 0 et taille-1

Pas vu : tableau associatif ou indice énumératif

un type énumératif est un ensemble de valeurs discrètes

```
Couleurs : [ Rouge, Orange, Vert ]   // autres syntaxes possibles
feu: Couleurs = Rouge
```

Attention : Rouge ≠ "Rouge"

un tableau associatif est indicé par les valeurs d'un type énumératif

```
declare
    pixelCouleur : tableau[Couleurs] d'entiers
    p : pixelCouleur
debut
    p[Rouge] = 128
fin
```


Synthèse sur les tableaux

en une diapo

Les tableaux 1D, 2D et plus ... en une diapo

Un tableau regroupe des objets de même type

Ses éléments sont accédés par des indices

Autant d'indices que de dimensions : [i] → 1D [i, j] → 2D [i,j,k] → 3D ...

Les indices varient entre 0 et taille-1 (en général)

La taille est fixée une fois pour toute à la déclaration (en général)

Déclarer n'est pas initialiser

Initialiser, lire, afficher : la classique boucle pour imbriquée 1 fois, 2 fois, 3 fois ...

Exemples de traitements classiques avec des images et des matrices

Les sous-programmes avec des paramètres tableaux doivent aussi préciser les tailles correspondantes

Commencer à compter le nombre d'itérations des boucles imbriquées, et leur effet sur le nombre d'opérations ou d'instructions alors exécutées

Commencer à compter la place mémoire nécessaire au stockage et aux traitements