TP 12

Algorithme de Gauss-Jordan

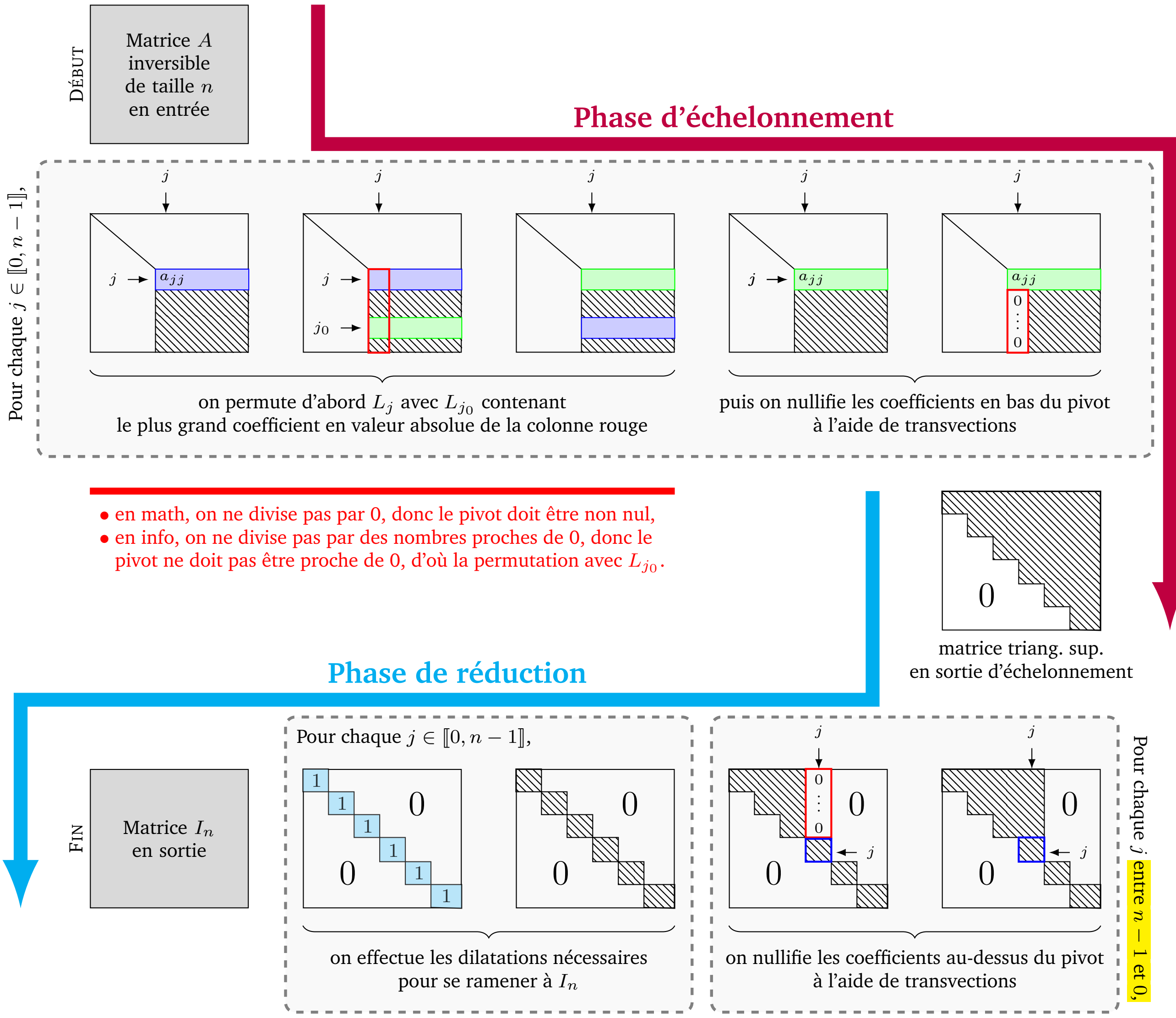
Vous compléterez le fichier réponse *TP12-Fichier réponse.py*

Explication Ce *TP 12 - Algorithme de Gauss-Jordan* est un prolongement naturel des chapitres suivants :

- *Chapitre 8 - Calculs matriciels*
- *Chapitre 9 - Systèmes linéaires*

I - Algorithme de Gauss-Jordan : Math vs Info !

Explication Détaillons l'algorithme **informatique** de Gauss-Jordan step by step :



Rappelons que pour déterminer l'inverse de la matrice A, il suffit d'effectuer en parallèle les opérations élémentaires (permutations, transvections et dilatations) subies par A sur la matrice I_n.

Cliquez ici pour accéder au corrigé

II - Codage de l'algorithme de Gauss-Jordan

✌ **Explication** ✌ **(Opérations élémentaires avec les matrices Numpy)**
Soit $A \in \mathscr{M}_n(\mathbb{R})$, $(\lambda, \alpha) \in \mathbb{R} \times \mathbb{R}^*$ et $(i,j) \in [\![0, n-1]\!]^2$ tel que $i \neq j$. Compléter :

Opérations élémentaires	Code Python (sans boucle)
$L_j \leftarrow L_i + \lambda L_j$	
$L_j \leftarrow \alpha L_j$	
$L_j \leftrightarrow L_i$	

1 ***Exercice***

🕒🕒 **(Algorithme de Gauss-Jordan)**

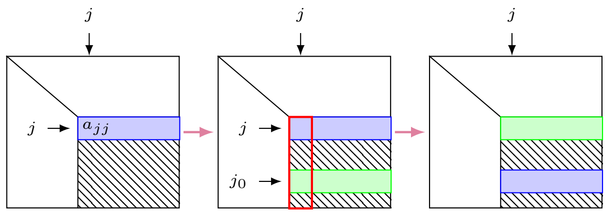
1) *Phase d'échelonnement de l'algorithme de Gauss-Jordan*

a) Écrire une fonction

`pivot_max(A,B,j)`,

qui prend en paramètres deux matrices `A` et `B`, ainsi qu'un entier `j`, et modifie (sans rien retourner) :

- la matrice `A` en permutant sa ligne d'indice `j` avec la ligne contenant le pivot maximal en valeur absolue comme illustré ci-contre,
- la matrice `B` en lui faisant subir la permutation précédemment effectuée sur la matrice `A`.



La matrice A en entrée est partiellement échelonnée jusqu'à sa ligne d'indice j. D'abord on cherche la ligne d'indice j_0 contenant le plus grand coefficient en valeur absolue de la colonne rouge, puis on permute L_j et L_{j_0}.

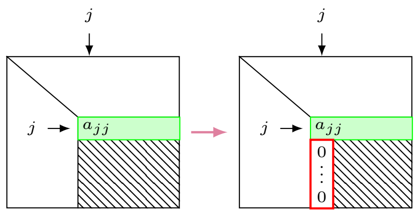
FIGURE 12.1 – `pivot_max(A,B,j)`

b) Écrire une fonction

`trans_bas(A,B,j)`,

qui prend en paramètres deux matrices `A` et `B`, ainsi qu'un entier `j`, et modifie (sans rien retourner) :

- la matrice `A` en effectuant les transvections nécessaires pour nullifier les coefficients en bas du pivot comme illustré ci-contre,
- la matrice `B` en lui faisant subir les transvections précédemment effectuées sur la matrice `A`.



On nullifie les coefficients en bas du pivot à l'aide de transvections

FIGURE 12.2 – `trans_bas(A,B,j)`

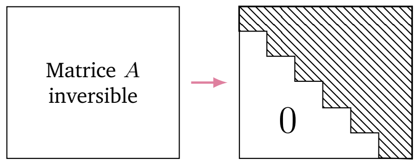
☞ Cliquez ici pour accéder au corrigé

c) À l'aide des deux fonctions précédentes, écrire une fonction

`echelonnement(A,B),`

qui prend en paramètres deux matrices `A` et `B`, et modifie (sans rien retourner) :

- la matrice `A` en l'échelonnant (trigonalisant même) suivant l'algorithme de Gauss-Jordan,
- la matrice `B` en lui faisant subir les opérations élémentaires précédemment effectuées sur la matrice `A`.



On échelonne/trigonalise la matrice A en suivant l'algorithme informatique de Gauss-Jordan.

FIGURE 12.3 – `echelonnement(A,B)`

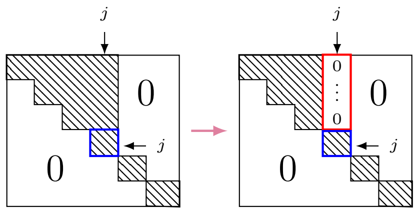
2) *Phase de réduction de l'algorithme de Gauss-Jordan*

a) Écrire une fonction

`trans_haut(A,B,j),`

qui prend en paramètres deux matrices `A` et `B`, ainsi qu'un entier `j`, et modifie (sans rien retourner) :

- la matrice `A` en effectuant les transvections nécessaires pour nullifier les coefficients au-dessus du pivot comme illustré ci-contre,
- la matrice `B` en lui faisant subir les transvections précédemment effectuées sur la matrice `A`.



La matrice A en entrée est partiellement échelonnée réduite. On nullifie les coefficients au-dessus du pivot à l'aide de transvections.

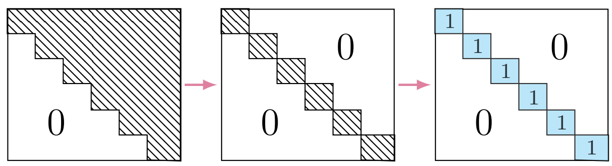
FIGURE 12.4 – `trans_haut(A,B,j)`

b) À l'aide de la fonction précédente, écrire une fonction

`reduction(A,B),`

qui prend en paramètres deux matrices `A` et `B`, et modifie (sans rien retourner) :

- la matrice `A` en la réduisant suivant l'algorithme de Gauss-Jordan,
- la matrice `B` en lui faisant subir les opérations élémentaires précédemment effectuées sur la matrice `A`.



La matrice A en entrée est triangulaire supérieure. On la réduit en suivant l'algorithme de Gauss-Jordan.

FIGURE 12.5 – `reduction(A,B)`

3) *Conclusion - Algorithme de Gauss-Jordan*

a) À l'aide des fonctions précédemment créées, écrire une fonction

`inverse(A),`

qui prend en paramètre une matrice `A` inversible, et renvoie sa matrice inverse sans modifier la matrice d'origine.

☞ Cliquez ici pour accéder au corrigé

b) Tester la fonction `inverse(A)` avec la matrice associée au système linéaire ($\bigstar$) suivant d'inconnue $(x, y, z, t) \in \mathbb{R}^4$, puis en déduire sa résolution :

$$\begin{cases} 2x + 4y - 4z + t = 0 \\ 3x + 6y + z - 2t = -7 \\ -x + y + 2z + 3t = 4 \\ x + y - 4z + t = 0 \end{cases}$$

On pourra comparer le résultat à celui obtenu par la fonction native `np.linalg.inv(A)` permettant de calculer l'inverse d'une matrice A inversible.

4) *Introduction à la notion de complexité temporelle : vous versus Numpy !*

a) Écrire une fonction

`mat_aleat(n),`

qui prend en paramètre un entier `n`, et renvoie une matrice carrée de taille `n` dont les coefficients sont des réels aléatoirement choisis dans l'intervalle $[0, 1[$.

```
import random

#la fonction random du module random renvoie un flottant aléatoire de [0,1[
>>> random.random()
0.09876709033668662
```

b) Tracer sur un même graphe :

- le logarithme du temps nécessaire à la fonction `inverse(A)` pour calculer l'inverse d'une matrice générée aléatoirement en fonction du logarithme de sa taille n pour n variant de 100 à 1000, en incrémentant de 100 à chaque fois.
- la même chose mais avec la fonction native `np.linalg.inv(A)` pour ces dix mêmes matrices.

Quelles conclusions peut-on en tirer ?

```
import time

debut=time.time()       #temps initial
###  Ici les instructions dont on veut mesurer le temps d'exécution  ###
fin=time.time()         #temps final
duree=fin-debut         #durée d'exécution des instructions
```

☞ *Cliquez ici pour accéder au corrigé*