

---

## TP 10 : Résolution de systèmes et pivot de Gauss

---

**Devoir à la maison.** Le TP est à terminer à la maison. La section 3 est à rendre sur feuille pour le 17/03.

**Fichier annexe.** Télécharger sur le site web le fichier `annexe_tp10_gauss.py`.

**Rappels.** Si  $M$  est une liste de  $n$  listes contenant elles-mêmes  $m$  éléments, on la considère comme une matrice  $n \times m$ .  $n$  est accessible par `len(M)`,  $m$  est accessible par `len(M[0])`. Pour  $0 \leq i < n$  et  $0 \leq j < m$ , l'élément à la  $i$ -ème ligne et la  $j$ -ème colonne est accessible par `M[i][j]`. Un vecteur est représenté comme une matrice  $n \times 1$ . Dans ce TP, on ne considérera que des matrices carrées, et des vecteurs.

### 1 Le pivot de Gauss

**L'algorithme du pivot de Gauss.** Les matrices sont donc représentées comme des listes de listes. Par exemple dans le système suivant :

$$AX = Y \quad \text{avec} \quad A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 2 \\ 2 & -1 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \quad \text{et} \quad Y = \begin{pmatrix} 2 \\ 9 \\ 7 \end{pmatrix}$$

la matrice  $A$  et le vecteur  $Y$  sont donnés comme suit :

```
A=[[1, 1, 1], [1, -1, 2], [2, -1, 1]]
Y=[[2], [9], [7]]
```

---

#### Algorithme 1 : Algorithme du pivot

---

**Entrées :** Une matrice  $A$  de taille  $n \times n$ , un vecteur  $Y$  de taille  $n \times 1$

**Sortie :** Un vecteur  $X$  solution de  $AX = Y$

Faire une copie des matrices  $A$  et  $Y$ ;

**pour chaque**  $i$  allant de 0 à  $n - 1$  **faire**

$j \leftarrow \text{indice\_pivot}(A, i)$ ;

**si**  $j \neq i$  **alors**

$\text{echange\_ligne}(A, i, j)$ ;

$\text{echange\_ligne}(Y, i, j)$

**pour chaque**  $j$  allant de  $i + 1$  à  $n - 1$  **faire**

$\mu \leftarrow -A[j][i]/A[i][i]$ ;

$\text{transvection}(A, j, i, \mu)$ ;

$\text{transvection}(A, j, i, \mu)$

$X \leftarrow$  un vecteur nul de taille  $n \times 1$ ;

**pour chaque**  $i$  allant de  $n - 1$  à 0 **par pas de**  $-1$  **faire**

$X[i][0] \leftarrow Y[i][0] - \frac{1}{A[i][i]} \sum_{j=i+1}^{n-1} A[i][j] \times Y[j][0]$

**retourner**  $X$

---

L'algorithme du pivot (algorithme 1) nécessite l'écriture des quatre fonctions de base suivantes :

- `echange_lignes(A, i, j)`, qui réalise l'échange de lignes  $L_i \leftrightarrow L_j$  sur la matrice  $A$ .
- `transvection(A, i, j, mu)`, qui réalise l'opération de transvection  $L_i \leftarrow L_i + \mu L_j$  sur les lignes de la matrice  $A$ .
- `indice_pivot(A, i)`, qui cherche et renvoie l'indice de la ligne  $j$  tel que le coefficient  $a_{j,i}$  soit maximal en valeur absolue parmi les éléments  $a_{k,i}$  avec  $i \leq k \leq n - 1$  où  $n$  est le nombre de lignes de la matrice  $A$ .
- `copie_matrice(A)`, qui renvoie une copie de la matrice  $A$  passée en entrée.

Il faut savoir recoder ces fonctions, mais elles vous sont données pour ce TP, ainsi que le code du pivot (dans l'annexe).

**Question 1.** Vérifier que tout fonctionne en résolvant le système  $AX = Y$  suivant (donné dans l'annexe) :

```
>>> A=[[1, 1, 1], [1, -1, 2], [2, -1, 1]]
>>> Y=[[2], [9], [7]]
>>> X=gauss(A,Y)
>>> print(X)
[[1.0], [-2.0], [3.0]]
```

Remarque : il est pratique d'utiliser la fonction (donnée dans l'annexe) `affiche(A)` qui affiche la matrice  $A$ . Pour ce faire, on imprime simplement à l'écran la matrice convertie en tableau numpy (via `np.array(A)`), où numpy a été importé sous le nom `np`.

**Question 2.** Exécuter le code suivant (donné dans l'annexe) :

```
from random import randint
n=5
A=[[0]*n for _ in range(n)]
for i in range(n):
    for j in range(n):
        A[i][j]=randint(0,50)
Y=[[sum(A[i])] for i in range(n)]
X=gauss(A,Y)
print(X)
```

La solution est sensée être un vecteur colonne constitué de 1. Vérifier que la solution obtenue n'est qu'approchée.

**Question 3.** *Utilisation de Numpy.* En transformant nos matrices en tableaux Numpy (comme ceux vus dans le TP sur les images), on peut utiliser les fonctions Numpy pour résoudre des systèmes. La syntaxe est la suivante (c'est l'exemple du cours) :

```
import numpy as np
A=np.array([[1, 1, 1], [1, -1, 2], [2, -1, 1]])
Y=np.array([[2], [9], [7]])
X=np.linalg.solve(A,Y)
```

En fait, la conversion en tableaux Numpy se fait automatiquement à l'utilisation de `np.linalg.solve`. Reprendre la question précédente avec Numpy. On utilisera `X.tolist()` pour transformer les entrées d'un tableau Numpy  $X$  en liste (Numpy a tendance à écourter l'affichage, on a l'impression qu'il calcule exactement mais non !)

*Remarque :* Numpy est souple d'utilisation : si on lui donne le second membre  $Y$  comme une liste (ou un tableau Numpy à une dimension), il renvoie un tableau Numpy à 1 dimension. Si on lui donne une matrice colonne (de dimension 2, comme ci-dessus), il renvoie la solution sous cette forme également.

On va maintenant travailler avec la famille des matrices  $H_n$  qui sont connues pour être *très mal conditionnées*, c'est-à-dire que les résultats fournis par l'algorithme du pivot sont très vite très éloignés de la réalité, et on verra que Numpy ne fait pas mieux que nous.

**Question 4.** Écrire une fonction `matrice_hilbert(n)` prenant en entrée un entier naturel strictement positif, et retournant la matrice  $H_n$  de taille  $n \times n$ , dont le coefficient en case  $(i, j)$  est  $\frac{1}{i+j+1}$  (lignes et colonnes sont indexées à partir de 0).

**Question 5.** On considère ici  $n = 5$ . Résoudre le système  $H_5 X = Y$ , avec pour  $Y$  les deux vecteurs ci-dessous (donnés dans l'annexe) :

$$\begin{pmatrix} -7.7 \\ -6 \\ -2.1 \\ -0.5 \\ 0 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} -7.7 \\ -6 \\ -2.1 \\ -0.4 \\ 0 \end{pmatrix}$$

On voit donc qu'une petite variation sur  $Y$  peut induire une grande variation sur  $X$  : cela est du au mauvais conditionnement de  $H_5$ . La notion de conditionnement est trop dure à définir pour nous en première année, mais elle ressemble à celle vue en cours pour l'étude des suites de la forme  $u_{n+1} = f(u_n)$  : plus le conditionnement est élevé, plus l'écart entre la solution de  $AX = Y$  et celle de  $AX = Y'$  (avec  $Y'$  « proche » de  $Y$ ) est important.

**Module fractions et résolution sur  $\mathbb{Q}$ .** En travaillant sur les rationnels, on n'introduit pas d'erreurs d'approximation, et on résout nos systèmes de manière *exacte*. L'algorithme du pivot fonctionne aussi sur les rationnels, s'il est correctement écrit. Pour travailler sur  $\mathbb{Q}$ , il faut importer la fonction `Fraction` du module `fractions`. Pour construire un rationnel, on donne son numérateur et son dénominateur de la façon suivante :

```
from fractions import Fraction
r=Fraction(p,q)
```

Les opérations  $+$ ,  $-$ ,  $*$ ,  $/$ ,... s'appliquent ensuite sur des rationnels comme sur des flottants. Si l'un des argument est un entier et l'autre un rationnel, le résultat sera rationnel.

**Question 6.** Écrire une fonction `matrice_hilbert_Q(n)` créant aussi une matrice de Hilbert, mais dont les éléments sont sous forme de fractions.

**Question 7.** Résoudre, pour quelques  $n$  entre 2 et 20, le système  $H_n X = Y$ , où  $H_n$  est la matrice de Hilbert d'ordre  $n$  et  $Y$  le vecteur ayant un 1 dans sa dernière composante, et des 0 ailleurs. On affichera à l'écran le vecteur  $X$  obtenu. On fera trois résolutions : sur les rationnels avec notre algorithme du pivot<sup>1</sup> (pour l'affichage, on pourra utiliser la fonction `affiche_float(M)` fournie, qui affiche à l'écran un vecteur / une matrice de rationnels sous forme de flottants), sur les flottants avec notre algorithme, et sur les flottants avec Numpy. Vérifier qu'à partir de  $n = 12$  environ, les deux résolutions flottantes s'éloignent drastiquement de la vraie solution (calculée de manière exacte avec l'algorithme de Gauss sur  $\mathbb{Q}$ ).

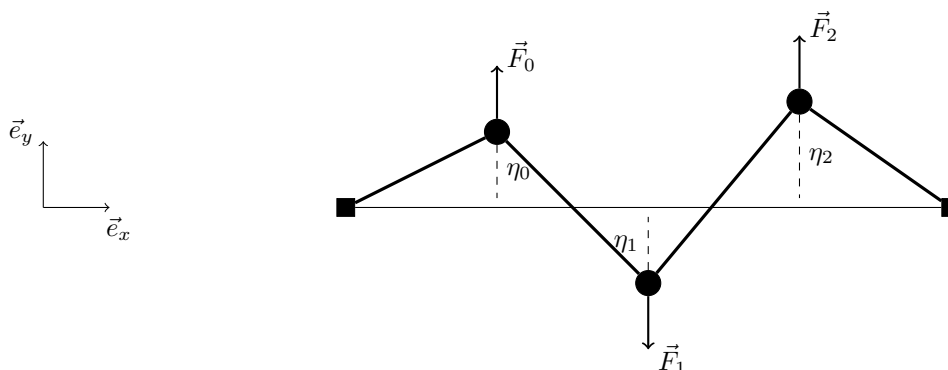
**Question 8.** Pour mesurer le temps d'exécution d'un processus, on peut utiliser la fonction `process_time` du module `time` (déjà importée dans l'annexe). Cette fonction (sans argument) mesure le temps processeur depuis le début de l'exécution. Pour mesurer le temps d'exécution d'un script, il suffit de faire un appel à `process_time` avant et après, la soustraction des deux temps obtenus donne le résultat, comme suit

```
t1 = process_time ()
[instructions dont on veut mesurer le temps]
t2 = process_time ()
# t2 - t1 donne le temps d'exécution.
```

Comparer le temps d'exécution pour des matrices de taille  $\simeq 100$  du pivot avec des rationnels et du pivot sur flottants sur la matrice  $H_n$ . Le temps d'exécution de l'algorithme avec des rationnels n'est plus en  $O(n^3)$ , pourquoi ?

## 2 Une application physique : système de masses sur un fil tendu

On considère un fil horizontal, attachés aux extrémités, sur lequel sont régulièrement réparties  $N$  masses, numérotées de 0 à  $N - 1$ . On applique sur chaque masse une force perpendiculaire au fil, également dans le plan horizontal, comme le montre le schéma suivant ( $N = 3$  ici).



On cherche à étudier les petits déplacements  $\eta_i$  des masses (largement exagérés sur la figure ci-dessus) dans la direction perpendiculaire à l'axe du fil. Pour de petits déplacements, la longueur du fil entre deux masses reste

1. Il fonctionne sans changement sur les rationnels !

constante égale à  $\ell$ , et la tension du fil est également constante. Au premier ordre, l'équation du mouvement de la masse numéro  $i$  dans la direction  $\vec{e}_y$  s'écrit :

$$m_i \ddot{\eta}_i = f_i - \frac{T}{\ell}(\eta_i - \eta_{i-1}) + \frac{T}{\ell}(\eta_{i+1} - \eta_i)$$

où  $\ell$  est la longueur du fil entre deux masses (ou une masse et l'extrémité du fil),  $T$  est la tension du fil et  $\vec{F}_i = f_i \vec{e}_y$ , avec la convention  $\eta_{-1} = \eta_N = 0$ .

En convenant que  $\frac{T}{\ell} = 1 \text{ N.m}^{-1}$  pour simplifier, le système linéaire vérifié par  $(\eta_0, \dots, \eta_{N-1})$  lorsque le système est à l'équilibre et que les forces  $\vec{F}_i$  sont de plus supposées constantes est donc :

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \cdots & \cdots & -1 & 2 & -1 & 0 \\ 0 & \cdots & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & \cdots & 0 & 0 & -1 & 2 \end{pmatrix} \times \begin{pmatrix} \eta_0 \\ \eta_1 \\ \eta_2 \\ \vdots \\ \vdots \\ \eta_{n-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{n-1} \end{pmatrix}$$

Ce genre de système *tridiagonal*<sup>2</sup> intervient fréquemment en mécanique.

**Question 9.** Écrire une fonction `tridiag(n)` prenant en entrée un entier  $n \geq 3$ , et renvoyant la matrice précédente (sous forme de liste de listes).

**Question 10.** Reprendre l'étude de la section 1 avec les matrices obtenues pour ce système, et le vecteur  $Y$  dont les composantes sont toutes égales à 1. Vérifier que l'algorithme du pivot se passe plutôt bien pour ces matrices là.

### 3 Inversion de matrice

Une matrice carrée  $A$  de taille  $n \times n$  est dite inversible s'il existe  $B$  telle que  $AB = I_n$ , où  $I_n$  est la matrice identité (comportant des 1 sur la diagonale et des zéros ailleurs). On note  $A^{-1}$  cette matrice  $B$ . Il n'est pas très compliqué de modifier l'algorithme du pivot pour calculer  $A^{-1}$  (si elle existe). C'est le but des questions qui suivent, ici on ne travaillera qu'avec des listes Python.

**Question 11.** Écrire une fonction `dilatation(A,i,mu)` réalisant l'opération  $L_i \leftarrow \mu L_i$  sur la matrice  $A$  (c'est-à-dire la multiplication de tous les coefficients de la  $i$ -ème ligne de  $A$  par  $\mu$ ).

Le principe du calcul de l'inverse de  $A$  est le suivant :

- on part de la matrice  $A$  et de la matrice  $T = I_n$  ;
- on effectue des opérations sur les lignes comme dans l'algorithme du pivot de Gauss, simultanément sur la matrice  $A$  et la matrice  $T$ . Pour le pivot de Gauss, on se contentait de transformer  $A$  en matrice triangulaire supérieure, ici il faut transformer  $A$  en la matrice identité ;
- une fois  $A$  transformée en l'identité, la matrice  $T$  est alors égale à  $A^{-1}$  !

**Question 12.** Écrire une fonction `creer_id(n)` créant la matrice identité de taille  $n$  (constituée de 0 partout, sauf sur la diagonale où il y a des 1).

**Question 13.** Coder effectivement une fonction `inverse(A0)`, semblable à la fonction `gauss`, mais calculant l'inverse de  $A_0$  (on commencera par faire une copie de la matrice passée en paramètre).

**Question 14.** Quelle est sa complexité ?

**Question 15.** Comparer avec numpy et la fonction `np.linalg.inv`, qui calcule l'inverse de la matrice passée en argument. Vérifier que pour résoudre  $AX = Y$ , il suffit alors de calculer  $X = A^{-1}Y$ . On pourra utiliser le produit de matrices de Numpy : `np.dot(M,N)` réalise le produit  $MN$ .

**Question 16.** Quel est l'intérêt de calculer l'inverse d'une matrice  $A$  si l'on a à résoudre *plusieurs* systèmes de la forme  $AX = Y$  ?

2. c'est-à-dire que les seuls coefficients non nuls sont sur la diagonale principale et les deux diagonales situées en dessous et au dessus.