TP4 : Implémentation de l'algorithme du pivot de Gauss

▶ Dans votre dossier `Info_2a`, créer le dossier `TP_4`.

I. Pivot de Gauss : rappel théorique

Le but de ce TP est d'implémenter en **Python** l'algorithme du pivot de Gauss.
Pour rappel, cet algorithme a été utilisé dans deux chapitres.

× pour calculer les solutions d'un système linéaire.

× pour calculer l'inverse d'une matrice (ce qui revient à résoudre un système ...)

Nous implémentons ici l'algorithme du pivot de Gauss dans le but de calculer l'inverse d'une matrice. Nous optons ici pour la présentation matricielle de l'algorithme, bien plus adaptée à **Python** que la présentation utilisant des systèmes linéaires.

Données d'entrée : une matrice $M \in \mathscr{M}_n(\mathbb{R})$. $(\star)$

Principe de l'algorithme : on peut le décrire en 3 étapes.

- Première étape : mise sous forme triangulaire.
  Partant du couple (M, I_n), on applique à ces deux matrices une suite d'opérations sur les lignes permettant de transformer M en une matrice triangulaire supérieure T.

$$(M, I_n) \xrightarrow{\text{Étape 1}} (T, P)$$

- Deuxième étape : mise sous forme diagonale.
  Partant du couple (T, N), on applique à ces deux matrices une suite d'opérations sur les lignes permettant de transformer T en une matrice diagonale Δ.

$$(T, P) \xrightarrow{\text{Étape 2}} (\Delta, R)$$

- Troisième étape : mise sous forme identité.
  Partant du couple (Δ, N), il s'agit de multiplier chaque ligne L de Δ (et donc de N) par l'inverse du cœfficient diagonal présent sur L.

$$(\Delta, R) \xrightarrow{\text{Étape 3}} (I_n, U)$$

Sortie : la matrice $U \in \mathscr{M}_n(\mathbb{R})$, inverse de la matrice M en entrée.

$(\star)$ Dans un premier temps, le programme ne sera appliqué qu'à des matrices inversibles. Il sera ensuite modifié pour repérer, au cours de l'exécution, les matrices qui ne sont pas inversibles.

II. Implémentation de l'algorithme du pivot de Gauss

II.0. Manipulation de matrices en Python : quelques rappels

Dans la suite, on considère la matrice $S = \begin{pmatrix} 1 & 3 & 2 & 5 \\ 7 & 4 & -1 & 0 \\ 3 & 8 & -1 & 3 \\ 1 & -5 & 4 & 6 \end{pmatrix}$

- Dans la console **Python**, stocker cette matrice dans une variable `S`.
- Par quel appel accède-t-on au cœfficient $(3, 2)$ de la matrice `S` ?

```
S[2,1]
```

- On souhaite remplacer le cœfficient $(3, 2)$ de la matrice `S` par le réel -2.
  Quel appel doit-on utiliser ?

```
S[2,1] = -2
```

- Par quel appel accède-t-on à la $3^{\text{ème}}$ colonne de la matrice `S` ?

```
S[:,2]
```

- Par quel appel accède-t-on à la $2^{\text{ème}}$ ligne de la matrice `S` ?

```
S[1,:]
```

- On souhaite effectuer, sur la matrice `S`, l'opération élémentaire suivante : $L_2 \leftarrow L_2 - 7\,L_1$.
  Quel appel doit-on réaliser ?

```
S[1,:] = S[1,:] - 7 * S[0,:]
```

- Par quel appel peut-on récupérer la taille $(= 4 \times 4)$ de la matrice `S` ?
  (on notera que le résultat apparaît sous forme d'un tuple de taille 1×2*)*

```
S.shape
```

- Par quel appel peut-on récupérer le nombre de lignes de $(= 4)$ de la matrice `S` ?

```
S.shape[0]
```

- Par quel appel peut-on récupérer le nombre de colonnes de $(= 4)$ de la matrice `S` ?

```
S.shape[1]
```

- Par quel appel peut-on obtenir la matrice identité d'ordre 4 ?
  Stocker cette matrice dans la variable `I`.

```
I = np.eye(4)
```

II.1. Première étape de l'algorithme

La première étape de l'algorithme consiste à transformer, par opérations élémentaires sur les lignes, la matrice M en une matrice sous forme triangulaire supérieure.

Pour ce faire :

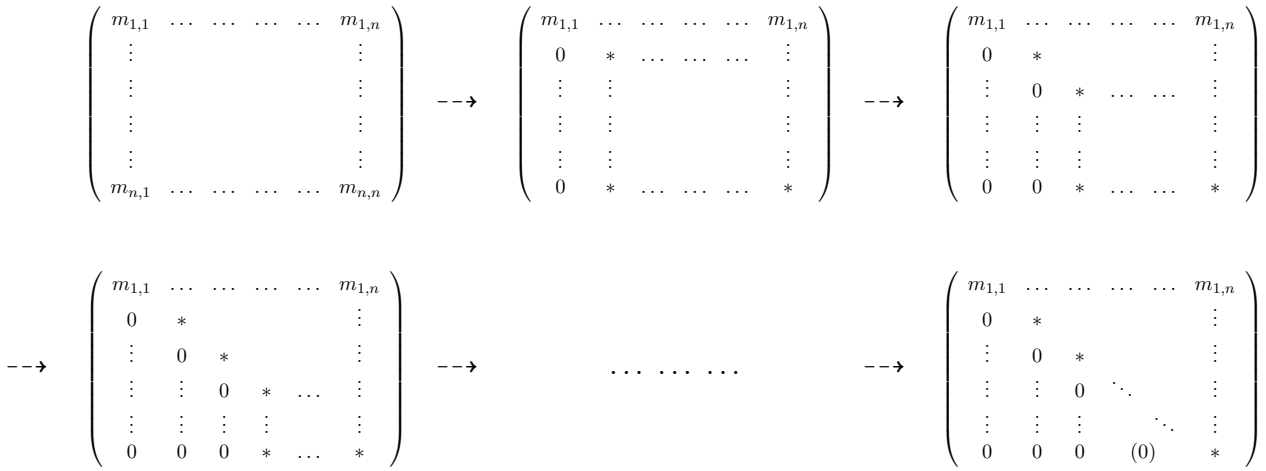
1) on commence par placer des 0 sous la diagonale dans la première colonne,

2) on place ensuite des 0 sous la diagonale dans la deuxième colonne,

...

n-1) on place enfin des 0 sous la diagonale dans la $(n-1)^{\text{ème}}$ colonne.

On rappelle que ces opérations doivent être effectuées en parallèle sur deux matrices (M et I_n initialement). Graphiquement, l'idée est donc la suivante (on représente uniquement les modifications successives sur la matrice M et pas sur I_n pour conserver de la lisibilité).



La matrice obtenue à la fin de cette étape est sous forme triangulaire supérieure.

Dans la suite, on considère la matrice $A = \begin{pmatrix} 1 & 3 & 2 & 5 \\ -1 & -1 & 1 & 2 \\ 2 & 4 & -1 & 3 \\ 3 & 5 & 4 & 6 \end{pmatrix}$ que l'on utilsera pour effectuer les tests sur nos programmes.

- Stocker cette matrice dans une variable `A`.
- Programmer la fonction `zeroColSousDiag` qui :
  - × prend en paramètre un numéro de colonne `j`, une matrice `M`, une matrice `N`,
  - × renvoie en sortie le couple de matrices `[G,D]` obtenu comme suit :
    - ***a)*** `G` contient initialement la matrice `M` (on n'oubliera pas qu'en **Python** il faut **copier** la matrice `M` dans `G` sinon toute modification sur `G` entrainera la même modification sur `M`),
    - ***b)*** `D` contient initialement la matrice `N`,
    - ***c)*** `G` est modifiée afin d'obtenir, par une succession d'opérations élémentaires, une matrice dont les cœfficients sous-diagonaux de la $\texttt{j}^{\text{ème}}$ colonne sont tous nuls,
      (on devra utiliser une structure itérative)
    - ***d)*** `D` est modifiée par application successive des mêmes opérations élémentaires que sur `G`.

```python
def zeroColSousDiag(j, M, N) :
    G = np.copy(M)
    D = np.copy(N)
    n = M.shape[0]
    for i in range(j+1, n) :
        D[i,:] = G[j,j] * D[i,:] - G[i,j] * D[j,:]
        G[i,:] = G[j,j] * G[i,:] - G[i,j] * G[j,:]
    return G, D
```

- Tester la fonction `zeroColSousDiag` sur le couple `(A,I)` pour la colonne 1.
  On stockera le résultat dans le couple `[G,D]`. Noter l'appel correspondant.

```
[G,D] = zeroColSousDiag(0, A, I)
```

- On souhaite maintenant tester `zeroColSousDiag` sur le couple `(G,D)` pour la colonne 2.
  Noter l'appel correspondant.

```
zeroColSousDiag(1, G, D)
```

- Programmer la fonction `transfTriangSup` qui :
  - × prend en paramètre une matrice `M`, une matrice `N`,
  - × renvoie en sortie le couple de matrices `[G,D]` obtenu comme suit :
    - ***a)*** `G` contient initialement la matrice `M`,
    - ***b)*** `D` contient initialement la matrice `N`,
    - ***c)*** `G` est modifiée afin d'obtenir une matrice triangulaire supérieure. On devra faire appel à la fonction `zeroColSousDiag` pour les colonnes de la matrice `M` initiale.
      (on devra utiliser une structure itérative)
    - ***d)*** `D` est modifiée par application successive des mêmes opérations élémentaires que sur `G`.

```python
from zeroColSousDiag import zeroColSousDiag
def transfTriangSup(M,N) :
    G = np.copy(M)
    D = np.copy(N)
    n = M.shape[0]
    for j in range(n) :
        [G,D] = zeroColSousDiag(j, G, D)
    return G, D
```

- Tester votre fonction `transfTriangSup` sur le couple `(A,I)`.
  Noter ci-dessous le résultat obtenu.

In [1] :
```
transfTriangSup(A, I)
```

Out [1] :
$$\begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 2 & 3 & 7 \\ 0 & 0 & -4 & 0 \\ 0 & 0 & 0 & -40 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ -2 & 2 & 2 & 0 \\ 24 & -32 & -16 & -8 \end{pmatrix}$$

- Comparer ce résultat avec le résultat obtenu lorsque vous effectuez à la main cette première étape de l'algorithme.

II.2. Deuxième étape de l'algorithme

La première étape de l'algorithme a consisté à transformer le couple (A, I_n) de matrices en un couple (T, P) où la matrice T est une matrice triangulaire supérieure. La deuxième étape de l'algorithme consiste à transformer, par opérations élémentaires sur les lignes, la matrice T en une matrice sous forme triangulaire diagonale.

Pour ce faire :

1) on commence par placer des 0 au-dessus de la diagonale dans la dernière colonne,

2) on place ensuite des 0 au-dessus de la diagonale dans l'avant dernière colonne,

...

n-1) on place enfin des 0 au-dessus de la diagonale dans la 1ère colonne.

On rappelle que ces opérations doivent être effectuées en parallèle sur deux matrices (T et P initialement). Graphiquement, l'idée est donc la suivante (on représente uniquement les modifications successives sur la matrice T et pas sur P pour conserver de la lisibilité).

$$\begin{pmatrix} m_{1,1} & \dots & \dots & \dots & \dots & m_{1,n} \\ 0 & * & & & & \vdots \\ \vdots & 0 & * & & & \vdots \\ \vdots & \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & (0) & & * \end{pmatrix} \dashrightarrow \begin{pmatrix} m_{1,1} & \dots & \dots & \dots & * & 0 \\ 0 & * & & & \vdots & \vdots \\ \vdots & 0 & * & & \vdots & \vdots \\ \vdots & \vdots & 0 & \ddots & * & \vdots \\ \vdots & \vdots & \vdots & & \ddots & 0 \\ 0 & 0 & 0 & (0) & & * \end{pmatrix} \dashrightarrow \begin{pmatrix} m_{1,1} & \dots & \dots & * & 0 & 0 \\ 0 & * & & \vdots & \vdots & \vdots \\ \vdots & 0 & * & * & \vdots & \vdots \\ \vdots & \vdots & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \vdots & & \ddots & 0 \\ 0 & 0 & 0 & (0) & & * \end{pmatrix}$$

$$\dashrightarrow \quad \dots \dots \dots \quad \dashrightarrow \begin{pmatrix} m_{1,1} & 0 & \dots & \dots & 0 & 0 \\ 0 & * & \ddots & (0) & \vdots & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \vdots & & \ddots & 0 \\ 0 & 0 & 0 & (0) & & * \end{pmatrix}$$

La matrice obtenue à la fin de cette étape est sous forme diagonale.

On testera les programmes qui suivent sur la matrice A introduite précédemment : $A = \begin{pmatrix} 1 & 3 & 2 & 5 \\ -1 & -1 & 1 & 2 \\ 2 & 4 & -1 & 3 \\ 3 & 5 & 4 & 6 \end{pmatrix}$.

On l'utilsera de nouveau pour effectuer les tests sur nos programmes.

- Programmer la fonction `zeroColSurDiag` qui :
  - × prend en paramètre un numéro de colonne `j`, une matrice `T`, une matrice `P`,
  - × renvoie en sortie le couple de matrices `[G,D]` obtenu comme suit :
    - ***a)*** `G` contient initialement la matrice `T`,
    - ***b)*** `D` contient initialement la matrice `P`,
    - ***c)*** `G` est modifiée afin d'obtenir, par une succession d'opérations élémentaires, une matrice dont les cœfficients sur-diagonaux de la `j`ème colonne sont tous nuls,
      (on devra utiliser une structure itérative)
    - ***d)*** `D` est modifiée par application successive des mêmes opérations élémentaires que sur `G`.

```
def zeroColSurDiag(j, T, P) :
    G = np.copy(T)
    D = np.copy(P)
    for i in range(j) :
        D[i,:] = G[j,j] * D[i,:] - G[i,j] * D[j,:]
        G[i,:] = G[j,j] * G[i,:] - G[i,j] * G[j,:]
    return G, D
```

- On note `(T,P)` le couple obtenu par application de la fonction **`transfTriangSup`** sur le couple `(A,I)`. Tester votre fonction **`zeroColSurDiag`** sur `(T,P)` pour la colonne 4. On stockera le résultat dans le couple `[G,D]`. Noter l'appel correspondant.

```
[T,P] = transfTriangSup(A,I)
[G,D] = zeroColSurDiag(3, T, P)
```

- On souhaite maintenant tester **`zeroColSurDiag`** sur le couple `(G,D)` pour la colonne 3.
On stockera de nouveau le résultat dans le couple `[G,D]`. Noter l'appel correspondant.

```
[G,D] = zeroColSurDiag(2, G, D)
```

- Programmer la fonction **`transfDiag`** qui :
  - × prend en paramètre une matrice `T` (supposée triangulaire supérieure) et une matrice `P`,
  - × renvoie en sortie le couple de matrices `[G,D]` obtenu comme suit :
    - ***a)*** `G` contient initialement la matrice `T`,
    - ***b)*** `D` contient initialement la matrice `P`,
    - ***c)*** `G` est modifiée afin d'obtenir une matrice triangulaire supérieure. On devra faire appel à la fonction **`zeroColSurDiag`** pour les colonnes de la matrice `T` initiale.
      (on devra utiliser une structure itérative)
    - ***d)*** `D` est modifiée par application successive des mêmes opérations élémentaires que sur `G`.

```
from zeroColSurDiag import zeroColSurDiag
def transfDiag(T, P) :
    G = np.copy(T)
    D = np.copy(P)
    n = T.shape[0]
    for j in range(n) :
        [G, D] = zeroColSurDiag(j, G, D)
    return G, D
```

- Tester votre fonction `transfDiag` sur le couple `(T,P)` défini précédemment.
  Stocker le résultat obtenu dans le couple `[Delta,R]`. Noter ci-dessous le résultat obtenu.

In [2] :
```
[Delta, R] = transfDiag(T, P)
```

Out [2] :
$$\begin{pmatrix} 320 & 0 & 0 & 0 \\ 0 & 320 & 0 & 0 \\ 0 & 0 & 160 & 0 \\ 0 & 0 & 0 & -40 \end{pmatrix} \qquad \begin{pmatrix} -816 & 528 & 304 & 352 \\ 592 & -496 & -208 & -224 \\ 80 & -80 & -80 & 0 \\ 24 & -32 & -16 & -8 \end{pmatrix}$$

- Comparer ce résultat avec le résultat obtenu lorsque vous effectuez à la main cette deuxième étape de l'algorithme.

II.3. Troisième étape de l'algorithme

La deuxième étape de l'algorithme consiste à transformer, par opérations élémentaires sur les lignes, le couple (T, P) en le couple (Δ, R) où Δ est une matrice sous forme triangulaire diagonale. La troisième étape de l'algorithme consiste à transformer, par opérations élémentaires sur les lignes, la matrice diagonale Δ en la matrice I_n. On rappelle que ces opérations doivent être effectuées en parallèle sur deux matrices (Δ et R initialement).

Pour ce faire, il suffit de diviser chaque ligne par son terme diagonal.
(on rappelle que si la matrice initiale `A` *est inversible, tous les termes diagonaux de la matrice diagonale* Δ *sont non nuls)*

- Par quel appel **Python** peut-on effectuer l'opération élémentaire $L_1 \leftarrow \frac{1}{2} L_1$?

```
Delta[0,:] = (1/2) * Delta[0,:]
```

- Programmer la fonction `transfId` qui :
  - × prend en paramètre une matrice `Delta` (supposée diagonale) et une matrice `R`,
  - × renvoie en sortie le couple de matrices `[G,D]` obtenu comme suit :
    - ***a)*** `G` contient initialement la matrice `Delta`,
    - ***b)*** `D` contient initialement la matrice `R`,
    - ***c)*** `G` est modifiée afin d'obtenir la matrice identité.
    - ***d)*** `D` est modifiée par application successive des mêmes opérations élémentaires que sur `G`.

```
def transfId(Delta, R) :
    G = np.copy(Delta)
    D = np.copy(R)
    n = Delta.shape[0]
    for i in range(n) :
        D[i,:] = (1 / G[i,i]) * D[i,:]
        G[i,:] = (1 / G[i,i]) * G[i,:]
    return G, D
```

- Tester votre fonction `transfId` sur le couple `(Delta,R)` défini précédemment.
Stocker le résultat obtenu dans le couple `[L,U]`. Noter ci-dessous le résultat obtenu.

In [3] :
```
[L, U] = transfId(Delta, R)
```

Out [3] :
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -2.55 & 1.65 & 0.95 & 1.1 \\ 1.85 & -1.55 & -0.65 & -0.7 \\ 0.5 & -0.5 & -0.5 & 0 \\ -0.6 & 0.8 & 0.4 & 0.2 \end{pmatrix}$$

- Que doit contenir la matrice `L` ?

La matrice `L` doit contenir la matrice identité (de même taille que la matrice `A`).

- Vérifier que la matrice `U` est bien l'inverse de la matrice `A` initiale.
Noter ci-dessous la commande utilisée pour ce test.

```
np.dot(A, U)
```

III. Calculer l'inverse d'une matrice par pivot de Gauss

Toutes les étapes de l'algorithme du pivot de Gauss étant codées, il ne reste plus qu'à les effectuer de manière successive pour obtenir l'algorithme permettant le calcul de l'inverse d'une matrice.

- Programmer la fonction `calcInv` qui :
  - × prend en paramètre une matrice `M` (supposée inversible),
  - × et renvoie en sortie la matrice `Inv`, inverse de la matrice `M`.

```
from transfTriangSup import transfTriangSup
from transfDiag import transfDiag
from transfId import transfId
def calcInv(M) :
    G = np.copy(M)
    n = M.shape[0]
    D = np.eye(n)
    [G,D] = transfTriangSup(G, D)
    [G,D] = transfDiag(G, D)
    [G,D] = transfId(G, D)
    Inv = D
    return Inv
```

- On rappelle que l'on a introduit précédemment la matrice $A = \begin{pmatrix} 1 & 3 & 2 & 5 \\ -1 & -1 & 1 & 2 \\ 2 & 4 & -1 & 3 \\ 3 & 5 & 4 & 6 \end{pmatrix}$.

  Tester la fonction `calcInv` sur la matrice `A`. On stockera le résultat dans une variable `V`.
  Noter le résultat obtenu.

```
In [4] : V = calcInv(A)
```

Out [4] : $\begin{pmatrix} -2.55 & 1.65 & 0.95 & 1.1 \\ 1.85 & -1.55 & -0.65 & -0.7 \\ 0.5 & -0.5 & -0.5 & 0 \\ -0.6 & 0.8 & 0.4 & 0.2 \end{pmatrix}$

- Quel appel doit-on réaliser pour vérifier que `V` est l'inverse de `A` ?
  Commenter le résultat obtenu par cet appel.

`np.dot(V, A)`
Le résultat fourni par **Python** n'est pas clairement l'identité (mais une matrice qui en est extrêmement proche). En effet, le calcul sur les objets de type `float` fourni souvent seulement des valeurs approchées.

- Stocker la matrice $\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$ dans une variable `J`.

```
J = np.matrix([[1 for _ in range(4)] for _ in range(4)])
```

▶ Tester la fonction `calcInv` sur la matrice `J`. Commenter le résultat obtenu.

> On obtient un message d'erreur indiquant une division par 0.

▶ On considère la matrice $P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Est-elle inversible ? La stocker dans une variable `P`.

> On remarque :
> $$\det(P) \;=\; 0 \times 0 - 1 \times 1 \;=\; -1 \neq 0$$
> La matrice P est donc inversible.
> `P = np.matrix([[0,1], [1,0]])`

▶ Tester la fonction `calcInv` sur la matrice `P`. Commenter le résultat obtenu.

> Le même message d'erreur indiquant une division par 0 apparaît. Cependant, il semble que **Python** ait tout de même pu effectuer la première étape de l'algorithme.

IV. Déterminer si une matrice est inversible par pivot de Gauss

IV.1. Trouver le pivot

Une phase importante de l'algorithme a été oubliée !

- Lors de la mise sous forme triangulaire supérieure, nous avons choisi comme pivot à la $j^{\text{ème}}$ phase le cœfficient diagonal en position (j, j).
- Cependant cet élément peut être nul et ce même si la matrice considérée est bien inversible. *(cf matrice P précédente)*
- En fait, le pivot de cet $j^{\text{ème}}$ phase doit être choisi parmi les éléments sous-diagonaux non nuls. Le plus simple, algorithmiquement parlant, est de choisir (s'il existe) le premier élément sous-diagonal non nul comme pivot.

▶ Programmer la fonction `trouvePivot` qui :

× prend en paramètre un numéro de colonne `j`, une matrice `M`,

× renvoie en sortie l'entier `p` calculant la ligne du premier cœfficient sous-diagonal non nul de la `j`$^{\text{ème}}$ colonne de `M`,

× s'il n'y a pas de tel cœfficient, `p` devra contenir $n+1$ où n est le nombre de lignes de la matrice.

```
1  def trouvePivot(j, M) :
2      n = M.shape[0]
3      p = j
4      while M[p,j] == 0 and p < (n-1) :
5          p = p + 1
6      if M[p,j] == 0 and p == (n-1) :
7          p = n
8      return p
```

▶ Tester la fonction `trouvePivot` sur la matrice `P` précédente (en colonne 1).

IV.2. Échanger deux lignes

▶ On considère le programme suivant.

```
1 from random import*
2 x = random()
3 y = random()
4 print(‘‘La valeur de x est de : ‘‘ + str(x))
5 print(‘‘et la valeur de y est de : ‘‘ + str(y))
```

Copier ce programme dans un nouvel onglet. Que réalise-t-il ?

Ce programme :

× stocke dans une variable `x` une simulation d'une loi uniforme sur $[0, 1]$

× stocke dans une variable `y` une simulation d'une loi uniforme sur $[0, 1]$ (indépendante de la première),

× affiche la valeur contenue dans la variable `x` et dans la variable `y`

▶ À la fin de ce programme, ajouter une série d'instructions permettant de mettre à jour le contenu des variables `x` et `y` de sorte que :

× `x` contienne la valeur initiale de `y`,

× `y` contienne la valeur initiale de `x`.

```
6 aux = y
7 y = x
8 x = aux
```

▶ Noter ci-dessous les commandes permettant d'échanger les contenus de la $1^{\text{ère}}$ et $3^{\text{ème}}$ ligne de la matrice `U`.

```
6 aux = U[2,:]
7 U[2,:] = U[0,:]
8 U[0,:] = aux
```

▶ Programmer la fonction `echangeLignes` qui :

× prend en paramètre deux matrices `M` et `N` ainsi que deux entiers `i1` et `i2`,

× renvoie en sortie le couple `[G,D]` obtenu en échangeant les contenus de la `i1`$^{\text{ème}}$ et `i2`$^{\text{ème}}$ ligne des matrices `M` et `N`.

```
def echangeLignes(M, N, i1, i2) :
    G = np.copy(M)
    D = np.copy(N)
    auxG = np.copy(M)
    auxD = np.copy(N)
    G[i1,:] = auxG[i2,:]
    G[i2,:] = auxG[i1,:]
    D[i1,:] = auxD[i2,:]
    D[i2,:] = auxD[i1,:]
    return G, D
```

- Tester la fonction `echangeLignes` sur le couple de matrice `(U,J)`.
  Noter ci-dessous l'appel permettant de stocker le résultat dans un couple de matrice `[A,B]`.

```
[A, B] = echangeLignes(U, J, 0, 1)
```

IV.3. Modification des programmes précédents

- Modifier la fonction `transfTriangSup` comme suit :
  - × avant la structure itérative, ajouter une variable `j` contenant initialement `0`,
  - × avant la structure itérative, ajouter une variable `p` contenant initialement le résultat de `trouvePivot` appliqué au couple `(M,N)`,
  - × dans la structure itérative, et avant d'effectuer l'appel de `zeroColSousDiag`, procéder à un échange de ligne adéquat sur les matrices `G` et `D`,
  - × les variables `p` et `j` doivent être mises à jour dans la structure itérative,
  - × modifier la structure itérative de telle sorte que l'itération s'arrête si `p` vaut $n+1$ ou si `j` vaut $n+1$ (où n est le nombre de colonnes de la matrice `G`),
  - × ajouter une variable `bool` en sortie de boucle qui contient `T` si la matrice `M` est inversible et `F` sinon. Cette variable `bool` devra apparaître comme un paramètre de sortie de `transfTriangSup`.

```
def transfTriangSup(M, N) :
    G = np.copy(M)
    D = np.copy(N)
    n = M.shape[0]
    boolInv = True
    j = 0
    p = trouvePivot(j,G)
    while j <= (n-2) and p < n :
        [G, D] = echangeLignes(G, D, j, p)
        [G, D] = zeroColSousDiag(j, G, D)
        j = j + 1
        p = trouvePivot(j, G)
    if p == n :
        boolInv = False
    return G, D , boolInv
```

- Modifier la fonction `calcInv` afin :
  - × qu'elle affiche le message `La matrice n'est pas inversible` si la matrice considérée n'est pas inversible,
  - × qu'elle calcule l'inverse de la matrice considérée si elle est inversible.

```
from transfTriangSup import transfTriangSup
from transfDiag import transfDiag
from transfId import transfId

def calcInv(M) :
    G = np.copy(M)
    n = M.shape[0]
    D = np.eye(n)
    [G, D, boolInv] = transfTriangSup(G, D)
    if boolInv == True :
        [G, D] = transfDiag(G, D)
        [G, D] = transfId(G, D)
        Inv = D
    else :
        Inv = ‘‘La matrice n’est pas inversible‘‘
    return Inv
```