Réflexe :
Importer la librairie nécessaire au calcul scientifique.

Entrée[1]:
```
1 import numpy as np
```

TP Python 14 : Calcul matriciel

Correction

Partie I - Calcul matriciel : le module `linalg`

Dans le TP précédent, nous avons vu comment définir des matrices de toutes tailles en tant qu'*array*. Nous savons obtenir certaines informations de base sur les matrices, et savons effectuer un produit matriciel.

Il est désormais temps d'**utiliser** ces matrices...

Le module `linalg` (pour *linear algebra*) de la bibliothèque `numpy` contient un grand nombre d'algorithmes classiques d'algèbre linéaires. En ECG, on l'importera sous l'alias `al`, pour "*algèbre linéaire*". Cet acronyme étant français, il ne faut donc pas être perturbée si vous en rencontrez un différent en consultant des sites anglophones!

Réflexe :
Importer le module `linalg` de la bibliothèque `numpy` avec l'alias `al`.

Entrée[2]:
```
1 import numpy.linalg as al
```

Les algorithmes (ce sont des *fonctions* Python) prêts-à-l'emploi du module `linalg` que nous allons utiliser sont les suivants :

Fonctions	Usage
al.inv(M)	Pour inverser une matrice M.
al.matrix_rank(M)	Pour obtenir le rang d'une matrice M.
al.det(M)	Pour obtenir le déterminant d'une matrice M (en ECG, seulement pour les matrices 2×2.)
al.matrix_power(M,n)	Pour mettre une matrice **carrée** M à la puissance $n \in \mathbb{N}$.

al.solve(A,B)	Détermine l'exacte solution X d'un système matriciel $AX = B$ où A est une matrice **carrée** et B un **vecteur**.

Partie II - Exercices

Exercice 1 - Déterminant

Soit $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathcal{M}_2(\mathbb{R})$.

1. En vous rappelant de la formule du déterminant de M, implémenter une fonction `determinant` qui prend en argument une matrice carrée M d'ordre 2 et qui renvoie son déterminant.

   On rappelle que le coefficient en i-ème ligne et j-ème colonne de la matrice est accessible avec la commande `M[i-1,j-1]` .
2. *Tester votre fonction pour* $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$ *et* $C = \begin{pmatrix} 3 & -1 \\ 0 & 11 \end{pmatrix}$.
3. *Vérifier la robustesse de votre fonction en calculant* $\det(A)$, $\det(B)$ *et* $\det(C)$ *à l'aide de la fonction* `al.det` *. Que remarquez-vous?*

Entrée[16]:

```
def determinant(M):
    y = M[0,0] * M[1,1]-M[1,0]*M[0,1]
    return y
A = np.eye(2)
B = np.array([[1,2],[1,2]])
C = np.array([[3,-1],[0,11]])

print(determinant(A),al.det(A))
print(determinant(B),al.det(B))
print(determinant(C),al.det(C))
```

```
1.0 1.0
0 0.0
33 33.000000000000014
```

Exercice 2 - Inverse de matrice $2x2$

1. En vous rappelant de la formule permettant d'obtenir l'inverse d'une matrice $M \in \mathcal{M}_2(\mathbb{R})$, implémenter une fonction `inverse2x2` qui prend en argument une matrice carrée M d'ordre 2 et qui :
   - affiche un message d'erreur si M n'est pas inversible puis **s'arrête**.
   - renvoie son inverse M^{-1} sinon.
2. Tester votre fonction pour $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$ et $C = \begin{pmatrix} 3 & -1 \\ 0 & 11 \end{pmatrix}$.
3. Vérifier la robustesse de votre fonction à l'aide de la fonction `al.inv` . Que se passe-t-il pour la matrice B?

Entrée[18]:

```
def inverse2x2(M):
    if al.det(M)==0:
        print("La matrice n'est pas inversible")
        return
    else :
        a = M[0,0]
        b = M[0,1]
        c = M[1,0]
        d = M[1,1]
        N = (1/al.det(M))*np.array([[d,-b],[-c,a]])
        return N

A = np.eye(2)
B = np.array([[1,2],[1,2]])
C = np.array([[3,-1],[0,11]])

inverse2x2(B)
al.inv(B)
```

```
La matrice n'est pas inversible
```

Sortie[18]:

```
array([[nan, nan],
       [nan, nan]])
```

Exercice 3 - Inverse de matrices plus grandes

1. A l'aide de la fonction `al.rank` , écrire une fonction `test_inverse` qui prend en argument une matrice M quelconque et renvoie `True` si la matrice est inversible et `False` sinon. *On pourra penser à utiliser la fonction `np.shape` .*
2. Tester votre fonction pour $A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -3 & -2 \\ 0 & -2 & 0 \end{pmatrix}$, $B = \begin{pmatrix} 4 & -3 & 7 \\ -1 & 6 & 3 \\ 2 & 9 & 13 \end{pmatrix}$, $C = \begin{pmatrix} 1 & 2 & 4 \\ -2 & -4 & -8 \\ 3 & 6 & 12 \end{pmatrix}$ et $D = \begin{pmatrix} 1 & 1 \\ 0 & -1 \\ 2 & 3 \\ 4 & 0 \end{pmatrix}$
3. Ecrire une fonction `show_inverse` qui prend en argument une matrice quelconque M et qui :

- affiche un message d'erreur et **s'arrête** si la matrice n'est pas carrée.
- affiche un message d'erreur et **s'arrête** si la matrice est carrée non-inversible.
- affiche un message de succès et **renvoie** la matrice inverse M^{-1}.

4. Tester votre fonction avec les matrices A, B, C et D.

Entrée[16]:

```
def test_inverse(M):
    nb_lignes = np.shape(M)[0]
    nb_colonnes = np.shape(M)[1]


# cette correction est concise : il y a bien entendu d'autres manières de
# avec plusieurs tests.
    test = False

    if (nb_lignes == nb_colonnes) and (al.matrix_rank(M) == nb_lignes) :
        test = True

    return test

A = np.array([[1,2,1],[0,-3,-2],[0,-2,0]])
B = np.array([[4,-3,7],[-1,6,3],[2,9,13]])
C = np.array([[1,2,4],[-2,-4,-8],[3,6,12]])
D = np.array([[1,1],[0,-1],[2,3],[4,0]])

print("A est inversible?",test_inverse(A),'\n')
print("B est inversible?",test_inverse(B),'\n')
print("C est inversible?",test_inverse(C),'\n')
print("D est inversible?",test_inverse(D),'\n')

def show_inverse(M):

    # on teste si la matrice n'est pas carrée
    if np.shape(M)[0]!= np.shape(M)[1]:
        print("La matrice n'est pas carrée donc pas inversible.")
        return

    # si elle est carrée de taille nxn
    else :

        # soit son rang est différent de n auquel cas
        #la matrice n'est pas inversible
        if test_inverse(M) == False :
            print("La matrice est carrée mais pas inversible.")
            return

        # soit son rang vaut n et la matrice est
        # inversible
        else :
            print("La matrice est inversible!")
            return al.inv(M)

# on a déjà "return" pour chaque cas : il y a un "point final" dans chaque
# il est inutile d'en rajouter un en fin de fonction.

print(show_inverse(A),'\n')
print(show_inverse(B),'\n')
print(show_inverse(C),'\n')
print(show_inverse(D),'\n')

# l'usage de print est nécessaire pour la correction : vous pouvez ainsi v
# simultanément. Ce ne serait pas le cas si vous faisiez des appels succes
# Notez que puisque B, C et D ne sont pas inversibles, lorsqu'on demande d
# à show_inverse renvoie "None". En effet, quand une matrice n'est pas inv
# Autrement dit, on ne renvoie "Rien"...
```

```
A est inversible? True

B est inversible? False

C est inversible? False

D est inversible? False

La matrice est inversible!
[[ 1.    0.5   0.25]
 [ 0.    0.   -0.5 ]
 [ 0.   -0.5   0.75]]

La matrice est carrée mais pas inversible.
None

La matrice est carrée mais pas inversible.
None

La matrice n'est pas carrée donc pas inversible.
None
```

Exercice 4 - Puissances de matrices

On définit la matrice $A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Conjecturer la forme de A^n pour tout entier naturel n, et vérifier votre conjecture à l'aide d'une commande Python bien choisie.

Entrée[17]:

```
1 A = np.array([[1,0,1],[0,1,0],[0,0,1]])
2
3 print(al.matrix_power(A,3))
4 print(al.matrix_power(A,17))
```

```
[[1 0 3]
 [0 1 0]
 [0 0 1]]
[[ 1  0 17]
 [ 0  1  0]
 [ 0  0  1]]
```

Exercice 5 - Somme des premières puissances

1. Écrire une fonction prenant en argument une matrice carrée M et un entier p et renvoyant la somme $I_n + M + \ldots + M^p$.
2. Testez votre fonction avec la matrice A de l'exercice 4.

Entrée[27]:

```
def SumP(M,p):
    n = np.shape(M)[0]
    S = np.eye(n,n)

    for k in range(1,p+1):
        S = S + al.matrix_power(M,k)

    return S

# Vous pouviez bien entendu initialiser S avec al.matrix_power(M,0).

# Test
A = np.array([[1,0,1],[0,1,0],[0,0,1]])
SumP(A,3)
```

Sortie[27]:

```
array([[4., 0., 6.],
       [0., 4., 0.],
       [0., 0., 4.]])
```

Exercice 6 - Résolution de systèmes (1)

On considère le système suivant :

$$(\mathcal{S}) : \begin{cases} x + 2y + z &= 5 \\ -3y - 2z &= 2 \\ -2y &= 11 \end{cases}.$$

1. Au brouillon, écrire le système $(\mathcal{S})$ sous forme matricielle : $(\mathcal{S}) \sim AX = B$ où $A \in \mathcal{M}_4(\mathbb{R})$.
2. Écrire un programme permettant de déterminer la ou les solutions de ce système.

Entrée[39]:

```
A = np.array([[1,2,1],[0,-3,-2],[0,-2,0]])
B = np.array([5,2,11])
X = al.solve(A,B)
X
```

Sortie[39]:

```
array([ 8.75, -5.5 ,  7.25])
```

Exercice 7 - Résolution de systèmes (2)

Soit $(u_n)_n$, $(v_n)_n$ et $(w_n)_n$ les suites réelles définies par $u_0 = 0$, $v_0 = 1$ et $w_0 = 2$ et pour tout entier naturel n :

$$\begin{cases} u_{n+1} &= u_n &+ v_n &+ w_n \\ v_{n+1} &= &v_n &+ w_n \\ w_{n+1} &= & &w_n \end{cases}. \qquad \text{On note aussi } C_n = \begin{pmatrix} u_n \\ v_n \\ w_n \end{pmatrix}.$$

1. Définir la matrice M telle que, pour tout entier naturel n, $C_{n+1} = M \times C_n$.
2. Écrire un programme renvoyant le vecteur C_n pour un entier n de votre choix.
3. À l'aide des questions précédentes, écrire une fonction `sol_suite(n)` qui affiche les termes u_n, v_n et w_n au rang n, et qui ne renvoie rien.

Entrée[35]:

```
# Q1 : il suffit de retranscrire ce système sous forme matricielle
M = np.array([[1,1,1],[0,1,1],[0,0,1]])

# Q2 : C_n = M**n x C_0 , voir cours.
# On définit donc d'abord C_0

C0 = np.array([0,1,2])

# Remarquez qu'il n'y a pas besoin de faire np.transpose(C0) pour que le
# produit soit compatible : cela n'est propre qu'à Python

C3 = np.dot(al.matrix_power(M,3),C0)


# Q3
def sol_suite(n):
    C0 = np.array([0,1,2])
    M = np.array([[1,1,1],[0,1,1],[0,0,1]])

    Cn = np.dot(al.matrix_power(M,n),C0)
    un = Cn[0]
    vn = Cn[1]
    wn = Cn[2]
    print("Pour n = ",n,", u_n vaut",un,", v_n vaut",vn,", et w_n vaut",wn

    return

sol_suite(3)
```

```
Pour n =  3 , u_n vaut 15 , v_n vaut 7 , et w_n vaut 2
```