Chapitre # (NUM) 3 Tableaux Numpy. Application au calcul matriciel et traitement d'images

Résumé & Plan

Nous revoyons dans cette section certains éléments sur la manipulation de tableaux `numpy`, puis leur utilisation pour manipuler des matrices (opérations élémentaires, résolutions de systèmes linéaires, Pivot de GAUß, recherche des éléments propres), ainsi qu'en traitement d'images. Il existe d'autres exemples d'utilisation, par exemple en théorie des graphes (Chapitre (ALGO) 7) au travers de la notion de matrice d'adjacence. On termine par leur application au traitement d'images.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices 🌓 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

1. PRÉSENTATION SUCCINTE DE LA BIBLIOTHÈQE NUMPY

La librairie `numpy` est consacrée entièrement au calcul numérique en Python. Elle comprend les principales fonctions mathématiques (à l'instar du module `math`). Elle utilise essentiellement des variables de type `ndarray` (en abrégé `array`), que l'on peut voir comme des tableaux à plusieurs dimensions. Les calculs avec numpy sont particulièrement optimisés car les array sont homogènes (ils ne contiennent que des valeurs d'un même type) et de taille fixée à la création. [1] Traditionnellement on charge la librairie `numpy` avec la ligne :

```
>>> import numpy as np
```

Remarque 1 On pourrait utiliser aussi des listes de listes. L'avantage du type `array` est qu'on accède à toute une batterie de fonctions matricielles déjà définies (rang, recherche d'inverse, résolution de systèmes linéaires, *etc.*).

1.1. Généralités

DÉFINIR UN TABLEAU, TAILLE. On définit un tableau avec la fonction `np.array`. Regardons ensuite comment obtenir ses dimensions.

```
>>> A = np.array([[8, 3, 2] , [5, 1, 6]])
>>> A.shape # nb lignes / nb colonnes
(2, 3)
>>> A.dtype # le type des données contenues dans le tableau
dtype('int64')
>>> B = np.array([[8, 3, 2]])
>>> B.shape # nb lignes / nb colonnes
(1, 3)
```

1. C'est donc une différence notable avec les listes de listes

Attention Récupérer le nombre de lignes et de colonnes

Pour récupérer le format, l'inconvénient de `shape` est que la taille de ce tuple varie (pour un vecteur ligne, il n'aura qu'une seule dimension, donc l'instruction `n, p = A.shape` échouera). On conseille donc plutôt l'instruction suivante :

```
>>> A = np.array([[8, 3, 2]])
>>> n = len(A) # nombre de lignes
>>> n
1
>>> p = len(A[0]) # nombre de colonnes
>>> p
3
>>> A = np.array([[8, 3, 2] , [5, 1, 6]])
>>> n = len(A)
>>> n
2
>>> p = len(A[0])
>>> p
3
```

qui fonctionne donc aussi bien sur un vecteur ligne qu'une matrice plus classique.

Attention aux indices

Pour `n` lignes et `p` colonnes, la numérotation Python s'effectue entre `0` et `n-1` pour les lignes, et `0` et `p-1` pour les colonnes. Il y a donc un décalage avec l'indice des Mathématiques, source d'erreurs au début.

```
>>> A[1][2] # c'est bon
6
>>> A[1][3] # là ça ne va plus
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3
```

On ne peut pas non plus modifier un coefficient en le remplaçant par une valeur d'un autre type. Par exemple,

```
>>> A[1][1] = [-1, -1]
TypeError: int() argument must be a string, a bytes-like \
↪ object or a real number, not 'list'

The above exception was the direct cause of the following \
↪ exception:

Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Ainsi, les seules modifications autorisées sont celles de type initial que l'on obtient avec la méthode `dtype` :

```
>>> A.dtype # Donc ici, uniquement par un entier.
dtype('int64')
```

Attention Toutes les lignes ont même nombre d'éléments

Par exemple, la définition suivante échoue :

```
>>> M = np.array([[1, 2], [2]])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: setting an array element with a sequence. The \
↪ requested array has an inhomogeneous shape after 1 \
↪ dimensions. The detected shape was (2,) + inhomogeneous part.
```

On ne peut donc pas convertir n'importe quelle liste de listes en tableau : il faut que chaque ligne ait même nombre d'éléments.

Pour créer une matrice, on peut définir une liste de listes puis la convertir en tableau avec `np.array`. Mais on utilise généralement des fonctions permettant de créer facilement les matrices usuelles.

CONSTRUCTEURS DE TABLEAUX

Operations	Commande	Commentaire
Création	`A = np.array(...)`	On indique une liste de listes en argument
Matrice nulle	`A = np.zeros((n, p))`	Un tuple est demandé en argument, donc deux parenthèses
Matrice ATTILA *((avec que des 1))*	`A = np.ones((n, p))`	Un tuple est demandé en argument, donc deux parenthèses
Matrice identité	`A = np.identity(n)` ou `A = np.eye(n)`†	
Matrice diagonale	`A = np.diag(L)`	La liste `L` contient la diagonale
Subdivision de pas `h` de $[a, b]$	`np.arange(a, b, h)`	Analogue de `list(range(a, b, h))`

CONSTRUCTEURS DE TABLEAUX

Subdivision à `n` points de $[a,b]$	`np.linspace(a, b, n)`	On s'en est servi pour tracer des suites, fonctions, *etc.*
Coefficients	`A[i, j]` ou `A[i][j]`	Terme `i,j` du tableau
Ligne `i`	`A[i]`	
Colonne `j`	`A[:, j]`	« `:` » signifie en *slicing* « on prend tout »

† `eye` comme identity en anglais.

Attention **Copies**

Comme pour les listes, attention aux copies. En cas de copie « en dur » souhaitée, on utilisera la syntaxe `N = np.copy(M)` qui réalise une copie indépendante de `M`, dans `N`.

Voici quelques exemples.

Exemple 1

```
>>> np.identity(5)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

Exercice 1 | Solution Créer les matrices suivantes en Python à l'aide des fonctions précédentes (*i.e.* sans boucle).

$$A = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

On tapera directement les résultats dans la console.

Méthode **Créer une matrice**

Plusieurs options s'offrent à vous.

- Si la matrice est de petite taille, on écrit directement les coefficients.
- Si la matrice est de grande taille (typiquement dépendant d'un certain entier `n`), on peut :
  - ◇ soit utiliser des commandes existantes si la matrice est proche d'une matrice usuelle. Par exemple, `np.eye`, `np.zeros`, `np.ones`, *etc.*
  - ◇ Soit partir d'une matrice nulle initialisée à la bonne taille (avec `np.zeros`), puis la compléter des bons coefficients à l'aide d'une boucle `for`.

OPÉRATIONS. On peut effectuer un grand nombre d'opérations directement sur les array. On peut tout d'abord y appliquer des fonctions coefficient par coefficient. Par exemple,

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A**2
array([[ 1,  4],
       [ 9, 16]])
```

Ainsi, `A**2` va élever au carré chaque coefficient de `A`. Ce n'est donc pas $A \times A$. En utilisant les fonctions de `numpy`, on peut appliquer une fonction coefficient par coefficient.

```
>>> np.log(A)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

La plupart des fonctions mathématiques sont définies par `numpy`. La particularité de ces fonctions est qu'elles peuvent s'appliquer à un réel (comme avec le module `math`) mais aussi sur un tableau (voir l'exemple précédent avec la fonction ln).

OPÉRATIONS

Operations	Commande
Somme de deux matrices compatibles	`A + B`
Produits de deux matrices compatibles	`A @ B` **OU** `np.dot(A,B)`
Transposée	`np.transpose(A)`
Somme de tous les éléments	`np.sum(A)`

Opérations	
Produits de tous les éléments	`np.prod(A)`

Remarque 2 Pensez au symbole `@` pour des expressions matricielles compliquées (bien plus pratique que `np.dot()`).

PARCOURIR UN TABLEAU. Puisqu'on s'y repère comme dans une liste de listes, les parcours se font de la même manière. On imbrique donc deux boucles : l'une dont la variable parcourt les indices des lignes de la matrice et l'autre dont la variable parcourt les indices de ses colonnes. Supposons que `n, p` désignent le nombre de lignes et colonnes.

En indice

```
n, p = len(A), len(A[0])
for i in range(n):
    for j in range(p):
        ...
```

En valeur

```
for L in M:
    for x in L:
        ...
        # L est ici une ligne \
        ↪ de la matrice
```

Exercice 2 | **Matrices à créer** Solution On considère les matrices $A, B \in \mathfrak{M}_n(\mathbb{R})$ définies par :

1. $\forall (i,j) \in \{1,\dots n\}^2, \quad A_{ij} = ij,$
2. $\forall (i,j) \in \{1,\dots n\}^2, \quad B_{ij} = i^2 - j^2$ si $\quad i \leq j, \quad B_{ij} = 0$ sinon.

Créer ces deux matrices en Python dans deux fonctions d'en-têtes `creer_matrice_A(n)`, `creer_matrice_B(n)`.

Exercice 3 | **Somme** Solution Écrire une fonction d'en-tête `somme(M)` qui retourne la somme des éléments de `M`. *On s'interdira bien entendu d'utiliser `np.sum`.*

1.2. Différences entre tableaux numpy et listes

Même si les objets `ndarray` et `list` (listes de listes) semblent être très proches, il y a néanmoins quelques différences à bien garder en tête.

- La méthode `append` n'existe pas sur les tableaux, même unidimensionnels. Ainsi, un tableau a une certaine taille lors de sa création et conservera sa taille tant qu'il existe. Ce qui n'empêche pas de construire une liste de liste avec `append`, puis de convertir le tout en tableau avec `np.array()`.
- Une liste peut contenir des objets de natures différentes, alors que tous les éléments d'un tableau sont de même type. Type là encore défini lors de sa création et fixé jusqu'à la fin. Ainsi, la conversion en array d'une liste de listes ne respectant pas cela échouera.

2. APPLICATION AU CALCUL MATRICIEL

Nous avons vu pour l'instant comment créer un tableau, le parcourir, modifier ses éléments *etc.* et qu'un tableau permettait de coder une matrice en Mathématiques. L'objectif de cette section est de traiter des problèmes du cours de calcul matriciel à l'aide du module `numpy`.

Cadre

Dans toute cette section, l'ensemble $\mathbb{K}$ désignera $\mathbb{R}$ ou $\mathbb{C}$, et n, p désignent deux entiers supérieurs ou égaux à 1.

2.1. Calcul de puissances

Méthode Calcul des puissances d'une matrice avec Python

Soit `M` un tableau carré correspondant à une matrice M carrée. Il n'y a pas de fonction toute faite dans numpy pour calculer M^k. Rappelons également que `M**k` élève les coefficients de M à la puissance k mais n'effectue pas le produit matriciel. On procède comme suit :

- on initialise un tableau `P` à l'identité, de même format que `A`.
- On effectue k fois l'affectation `P = P @ M`.
- On renvoie `P`.

Exercice 4 | **Puissances et suites** Solution On considère trois suites $(x_n), (y_n), (z_n)$ vérifiant :

$$\forall n \in \mathbb{N}, \quad \begin{cases} x_{n+1} &= -x_n - 3y_n + 3z_n \\ y_{n+1} &= 3x_n - 7y_n + 3z_n \\ z_{n+1} &= 6x_n - 6y_n + 2z_n \end{cases} \qquad x_0 = y_0 = 1, \quad z_0 = 2.$$

On note par ailleurs : $\forall n \in \mathbb{N}, \quad X_n = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix}$.

1. Donner une matrice A telle que : $\forall n \in \mathbb{N}, \quad X_{n+1} = AX_n$. Créer cette matrice dans Python sous forme de tableau numpy. On peut montrer par récurrence que : $\forall n \in \mathbb{N}, \quad X_n = A^n X_0$.

2. Créer une fonction d'en-tête `puissance_mat(A, k)` qui retourne le tableau correspondant à A^k pour un entier k. En déduire une fonction `val_xyz(n)` qui retourne x_n, y_n, z_n étant donné un entier `n`. Conjecturer leur nature en exécutant pour plusieurs valeurs jusqu'à $n = 50$.
3. Proposer une version récursive `puissance_mat_rec` de la fonction `puissance_mat`.

Exercice 5 | Indice de nilpotence Solution En cas d'existence, on dit qu'une matrice $A \in \mathfrak{M}_{n,n}(\mathbb{K})$ est *nilpotente* s'il existe $p \in \mathbb{N}$ tel que $A^p = 0_{n,n}$. On appelle *indice de nilpotence* le plus petit entier p vérifiant cette propriété.

1. Soit $M = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix}$. Montrer que M est nilpotente, préciser son indice.

2. Écrire une fonction d'en-tête `indice_nilpo(M)` prenant en argument une matrice carrée `M` et renvoyant l'indice en question. *On pourra constater que la commande `np.all(P == np.zeros(P.shape))` teste si une matrice `P` est nulle.*

```
def indice_nilpo(M):
    ind = _
    P = np.copy(M)
    while ____________________________________ :
        P = _____
        ind += 1
    return ind
```

2.2. Propriétés

Exercice 6 | Triangulaire ou pas? Solution On rappelle qu'une matrice carrée $M \in \mathfrak{M}_{n,n}(\mathbb{K})$ est triangulaire supérieure si tous ses coefficients strictement en-dessous de la diagonale sont nuls, *i.e.* :

$$\forall (i,j) \in [\![1\,,\,n]\!]^2, \quad j < i \implies M_{i,j} = 0.$$

1. Écrire une fonction d'en-tête `est_triangulaire_sup(M)` qui retournent `True` si la matrice `M` est triangulaire supérieure, `False` sinon. Même chose pour tester si une matrice est triangulaire inférieure, en utilisant la fonction `est_triangulaire_sup`.
2. En déduire une fonction `est_diagonale(M)` qui retournent `True` si la matrice `M` est diagonale, `False` sinon.

Exercice 7 | Matrice des entiers consécutifs Solution

1. Créer fonction d'en-tête `creer_mat_entiers(n)` qui retourne une matrice de format $n \times n$ contenant tous les entiers entre 1 et n^2 (de gauche à droite et haut en bas). Par exemple, `creer_mat_entiers(3)` retournera `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.
2. Que vaut $\sum_{k=1}^{n^2} k$? Le retrouver à l'aide de la question précédente et d'un exercice précédent. *On exécutera les fonctions sur plusieurs valeurs de n*

Exercice 8 | Min / Max Solution

1. Écrire une fonction d'en-tête `min_max(L)` qui retourne le maximum et le minimum d'une liste `L`.
2. On souhaite trouver ici $\mathscr{M}$ défini par :

$$\mathscr{M} = \max \mathrm{E} - \min \mathrm{E}, \quad \mathrm{E} = \left\{ \sin^2(3k) \,\middle|\, k \in \mathbb{N}, 3k \leqslant 100 \right\}.$$

   2.1) Créer un tableau `A` qui contient tous les multiples de 3 entre 0 et 100. *On répondra en une seule commande du module numpy.*
   2.2) Créer un tableau ligne `B` qui contient les éléments de E. *On répondra en une seule commande du module numpy.*
   2.3) En déduire la valeur de $\mathscr{M}$.

Exercice 9 | Matrices stochastiques Solution On dit qu'une matrice $M \in \mathfrak{M}_{n,n}(\mathbb{R})$ est *stochastique* si la somme sur chaque ligne vaut 1 et chaque coefficient est entre 0 et 1, *i.e.* :

$$\forall i \in \{1,n\}, \quad \sum_{j=1}^{n} M_{i,j} = 1, \quad \forall (i,j) \in [\![1\,,\,n]\!]^2, \quad M_{i,j} \in [0,1].$$

Une matrice stochastique est dite *bistochastique* si en plus la somme sur chaque colonne vaut 1. *Ce type de matrice apparait souvent en probabilités, ce qui légitime leur étude.*

1. Parmi les matrices suivantes, dire si elles sont stochastiques et/ou bistochastiques. $A = \frac{1}{2}J_2$, $B = I_2$, $C = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{pmatrix}$, $D = \begin{pmatrix} -1 & 2 \\ 2 & -1 \end{pmatrix}$.

2. Écrire une fonction d'en-tête `est_stochastique(M)`, qui étant donnée une matrice carrée retourne `True` si elle est stochastique, et `False` dans le cas contraire.
3. Écrire une fonction d'en-tête `est_stochastique_prod(M, N)`, qui étant données deux matrices carrées (de formats compatibles) retourne `True` si MN est stochastique, et `False` dans le cas contraire. Tester sur plusieurs couples de matrices stochastiques. Que conjecturer ?
4. En utilisant la fonction `stochastique`, écrire une fonction d'en-tête `est_bistochastique(M)` qui retourne `True` si `M` est bistochastique.

2.3. Sous-module `np.linalg`

Le sous-module `np.linalg` de `numpy` possède des fonctions dédiées au calcul numérique en Algèbre linéaire (résolution de systèmes linéaires, calculs d'éléments propres, *etc.*). Nous allons en voir quelques unes.

Remarque 3 L'utilisation de `np.linalg` n'est pas un attendu du programme. En cas d'utilisation nécessaire, le sujet vous rappellera les commandes utiles.

OPÉRATIONS PLUS ÉVOLUÉES

Opérations	Commande
Inverse de A (en cas d'existence)	`np.linalg.inv(A)`
Rang	`np.linalg.matrix_rank(A)`
Résolution du système AX = B	`np.linalg.solve(A, B)`

Exercice 10 | Matrice de HILBERT Solution Pour tout $n \in \mathbb{N}^*$, on appelle matrice de HILBERT d'ordre n la matrice H_n de terme général $a_{i,j} = \frac{1}{i+j-1}$ pour tout $(i,j) \in [\![1, n]\!]^2$.

1. Écrire une fonction `hilbert(n)` qui retourne la matrice H_n.
2. Écrire, sur feuille, les matrices H_2, H_3 et justifier qu'elles sont inversibles. On admet qu'elle l'est pour tout $n \in \mathbb{N}^\star$.
3. Étudier le rang de H_n avec Python. Commenter.

Exercice 11 | Résolution d'un système linéaire Solution On considère le système suivant :

$$\begin{cases} 2x + 2y - 3z = 2 \\ -2x - y - 3z = -5 \\ 6x + 4y + 4z = 16. \end{cases}$$

Résoudre alors le système avec Python, puis vérifier que le résultat renvoyé est bien solution à l'aide de Python.

2.4. Méthode du miroir & Pivot de GAUß

Cette partie optionnelle n'est à traiter que si tout le reste a été terminé.

L'objectif est ici de programmer la méthode du Pivot de GAUß et du miroir pour l'inversion matricielle. Ce dernier thème a fait l'objet de la partie Informatique du sujet de modélisation 2019. Nous commençons par un exercice de fonctions préliminaires.

Exercice 12 | Codage des opérations élémentaires Solution On souhaite coder ici les opérations élémentaires sur les lignes du cours de Mathématiques.

1. Programmer une fonction d'en-tête `transvection(M, i, j, lamba)` qui modifie le tableau `M` directement, en appliquant $L_i \leftarrow L_i + \lambda L_j$.
2. Même question pour `permut(M, i, j)` correspondant à $L_i \leftrightarrow L_j$.
3. Même question pour `dilatation((M, i, lamba)` correspondant à $L_i \leftarrow \lambda L_i$.

Exercice 13 | Codage d'un choix de pivot Solution Soit la colonne numéro j dans la matrice `M`. On cherche le numéro $i^\star$ d'une ligne où est situé le plus grand coefficient (en valeur absolue) de cette colonne parmi les lignes j à $n-1$. Autrement dit,

$$|M[i^\star, j]| = \max\{|M[i,j]| \text{ pour } i \text{ tel que } j \leq i \leq n-1\}.$$

1. Pourquoi imposer la condition $j \leq i \leq n-1$ dans l'ensemble précédent ? À quoi cela correspond-il dans la méthode du miroir ou du pivot de GAUß ?

2. Écrire une fonction d'en-tête `rang_pivot(M, j)` prenant pour argument `M` et j, et qui retourne cette valeur de $i^\star$. Lorsqu'il y a plusieurs choix possibles pour cet indice, dire si votre algorithme retourne le plus petit, le plus grand ou autre.

MÉTHODE DU MIROIR. Nous allons implémenter la méthode du miroir pour l'inversion de matrices dans ce paragraphe.

Exercice 14 | Solution Compléter la fonction suivante pour qu'elle retourne la version échelonnée de `M`, une matrice carrée, et qui retourne une matrice où les mêmes opérations ont été réalisées sur la matrice identité.

```
def echelonnement(M):
    """
    modifie M pour avoir sa version échelonnée
    """
    n = len(M)
    N = np.eye(n, n)
    for j in range(n):
        i_star = rang_pivot(M, j)
        # On place le pivot au bon endroit
        permut(___, ___, ___)
        permut(___, ___, ___)
        # Élimination en-dessous du pivot
        for k in range(j+1, n):
            lamba = - M[k, j]/M[j, j]
            transvection(___, ___, ___, lamba)
            transvection(___, ___, ___, lamba)
    return N
```

On souhaite à présent obtenir la version échelonnée réduit de `M`.

Exercice 15 | **Échelonnement réduit** Solution Compléter la fonction ci-dessous afin que la fonction calcule l'échelonnée réduit de `echelonnement(M)`, et qui retourne une matrice où les mêmes opérations ont été réalisées sur la matrice identité.

```
def echelonnement_reduit(M):
    """
    modifie M pour avoir sa version échelonnée
    """
    n = len(M)
    echelonnement(M)

    #On fait apparaitre des pivots égaux à 1
    for i in range(n):
        ___________
        ___________

    # On fait apparaître des zéros au-dessus des pivots, en \
    ↪ commençant par la dernière colonne
    for j in range(n-1, 0, -1):
        for k in range(j-1, -1, -1):
            lamba = ___________
            transvection(___________)
            transvection(___________)
```

RÉSOLUTION DE SYSTÈMES LINÉAIRES. Toutes les briques de bases sont là pour pouvoir résoudre des systèmes linéaires à l'aide d'un algorithme d'échelonnement, comme nous l'avons fait *supra*. Soient b une matrice colonne, et A une matrice dont le nombre de colonnes est égal à la taille de b.

On commence par programmer une fonction `remontee(T, b)` qui résout le système `T X = b`, où `T` est une matrice triangulaire supérieure. Plus précisément, si le système est de CRAMER, `remontee` retourne l'unique solution, sinon elle retourne `False`.

■■ Fonctions de remontée et opération

```
import numpy as np
T = np.array([[1,7],[3,-4]])
b = np.array([1,1])

def remontee(T, b):
    """
    (T,b)->X solution de TX=b
    """
    n, p = len(T), len(T[0])
    if n == p and 0 not in [T[i,i] for i in range(n)]:
            # Systeme de Cramer
            X = np.zeros(p)
            i = p-1
            X[i] = b[i]/T[i,i]
            while i > 0:
                i = i-1
```

```
                X[i] = (1/T[i,i])*(b[i]-np.dot(X[i:],T[i][i:]))
            return X
    return False
```

On en déduit alors une fonction de résolution du système.

```
def resol_systeme(T, b):
    """
    T,b->solution au système TX=b, s'il est de cramer
    retourne false sinon
    """
    n, p = len(T), len(T[0])
    if n == p and 0 not in [T[i,i] for i in range(n)]:
        echelonnement(T)
        return remontee(T, b)
    return False
```

On peut ensuite résoudre le système correspondant à $MX = b$.

3. APPLICATION AU TRAITEMENT D'IMAGES

Les images peuvent être codées sous forme de tableaux `numpy`, nous précisons quelques points sur le sujet dans cette section.

3.1. Format & Codage d'une image

Il existe de nombreux formats d'image. Le plus fréquent est le format « .jpg » utilisé par exemples par les appareils photo numériques mais il a le défaut de comprimer les images au détriment de la richesse de l'information. On trouve ensuite le format « .bmp » essentiellement dans des contextes scientifiques. Il occupe plus de mémoire mais ne comprime en rien l'image. Nous utiliserons nous un format entre les deux, le png.

Remarque 4 (Matplotlib nécessite du png)

- matplotlib, qui sera utilisé dans la suite, ne permet de travailler qu'avec des images au format « .png ». Ça n'a rien de gênant si on considère qu'il est facile d'utiliser un logiciel de traitement d'image pour exporter au format « .png » des images de format « .bmp » ou « .jpg ». Pour autant on évitera si on le peut ces exportations qui se font au détriment d'une perte d'information et donc de la qualité de l'image.
- En cas de besoin de travail avec d'autres formats (TIPE par exemple), on utilisera le module pillow qui, une fois installé, permet l'acquisition et le traitement par matplotlib.image de plus de formats.

Dans tous les cas, une image sera comprise en informatique comme une grande grille. Chaque élément de la grille étant appelé « pixel », c'est-à-dire une portion d'image considérée de couleur constante (identique en tout point dudit pixel). Si l'image est composée de $n \times p$ pixels, chacun d'entre eux sera codé de façon différente selon que l'image est en niveaux de gris ou pas.

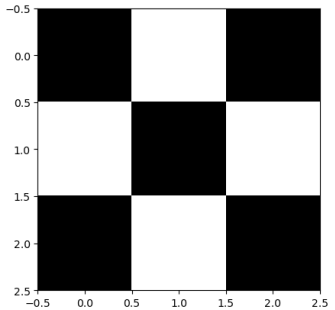
- Si c'est le cas, l'intensité lumineuse sera codée sur 8 bits et l'image aura deux dimensions.
- Sinon, dans le cas des images en couleur, c'est l'intensité lumineuse des trois couleurs fondamentales Rouge, Vert et Bleu qui est transcrite dans le domaine de sensibilité de l'oeil humain *via* un triplet de valeurs également codées sur 8 bits. On parle alors de mode « RGB » (pour Red, Green, Blue).
- Parfois, certaines images couleur possèdent en plus une composante de transparence. Les pixels auront donc chacun une 4 ème coordonnée. On peut ensuite facilement n'extraire que les trois premières et travailler ensuite avec cela. Un exemple sera fait plus tard.

Exemple 2 (Petits tableaux) Pour une image en noir et blanc, il y a deux états possibles et un bit suffira (0 pour un pixel noir, 1 pour le blanc). Par exemple, on précise ci-dessous un tableau ainsi que l'image associée.

$T =$

0	1	0
1	0	1
0	1	0

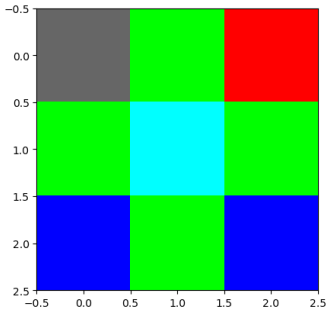
codera l'image :



$T =$

[0.4, 0.4, 0.4]	[0,1,0]	[1,0,0]
[0,1,0]	[0,1,1]	[0,1,0]
[0,0,1]	[0,1,0]	[0,0,1]

codera l'image :



- Si $r = g = b = 0$, le pixel est noir. Si r, g et b ont leur valeur maximale (donc 1 ou 255, la valeur par défaut étant 255), le pixel est blanc.
- Supposons chaque coordonnée soit un entier entre 0 et 1, alors : `[1,0,0]` est

du rouge pur, `[0,1,1]` est le cyan (complémentaire du rouge), etc. Les couleurs ayant des proportions identiques de rouge, vert et bleu `[x,x,x]` sont des gris de plus en plus clair lorsque `x` augmente.

Exemple 3 **(Le soleil et l'oiseau)** Autre exemple un peu plus sophistiqué, on considère les trois matrices ci-après (ici l'intervalle des valeurs n'est plus $[0,1]$ mais $[0,255]$, on bien sûr se ramener à $[0,1]$ en divisant chaque coordonnée par 255) :

$$R = \begin{pmatrix} 153 & 153 & 153 & 153 & 153 & 153 & 153 & 255 & 255 & 255 \\ 153 & 153 & 153 & 0 & 153 & 153 & 153 & 255 & 255 & 255 \\ 0 & 153 & 0 & 153 & 153 & 153 & 153 & 153 & 255 & 255 \\ 153 & 0 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 \\ 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 \\ 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 \\ 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 \\ 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 & 153 \\ 181 & 34 & 181 & 34 & 181 & 34 & 181 & 34 & 181 & 34 \\ 181 & 34 & 181 & 34 & 181 & 34 & 181 & 34 & 181 & 34 \end{pmatrix}, \quad G = \begin{pmatrix} 217 & 217 & 217 & 217 & 217 & 217 & 217 & 201 & 201 & 201 \\ 217 & 217 & 217 & 0 & 217 & 217 & 217 & 201 & 201 & 201 \\ 0 & 217 & 0 & 217 & 217 & 217 & 217 & 217 & 201 & 201 \\ 217 & 0 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 \\ 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 \\ 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 \\ 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 \\ 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 & 217 \\ 230 & 177 & 230 & 177 & 230 & 177 & 230 & 177 & 230 & 177 \\ 230 & 177 & 230 & 177 & 230 & 177 & 230 & 177 & 230 & 177 \end{pmatrix},$$

$$B = \begin{pmatrix} 234 & 234 & 234 & 234 & 234 & 234 & 234 & 14 & 14 & 14 \\ 234 & 234 & 234 & 0 & 234 & 234 & 234 & 14 & 14 & 14 \\ 0 & 234 & 0 & 234 & 234 & 234 & 234 & 234 & 14 & 14 \\ 234 & 0 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 \\ 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 \\ 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 \\ 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 \\ 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 & 234 \\ 29 & 76 & 29 & 76 & 29 & 76 & 29 & 76 & 29 & 76 \\ 29 & 76 & 29 & 76 & 29 & 76 & 29 & 76 & 29 & 76 \end{pmatrix}.$$

Alors le tableau de même taille que ces matrices, obtenu en assemblant les 3 valeurs des matrices pour former un pixel donnera l'image ci-après (par exemple, tout en haut à gauche, on aura `[153, 217, 234]`). On voit bien l'oiseau qui est matérialisé par les 0 dans chaque matrice, qui donne le pixel `[0, 0, 0]`. Le bleu ici est plutôt un cyan, *a contrario* du bleu précédent : ceci est normal puisque ce n'est pas un bleu « pur » mais un mélange de rouge (poids 153), vert (poids 217) et bleu (poids 234).

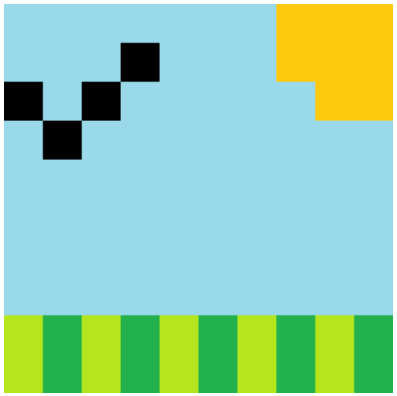


IMAGE OBTENUE PAR FUSION DES TROIS MATRICES

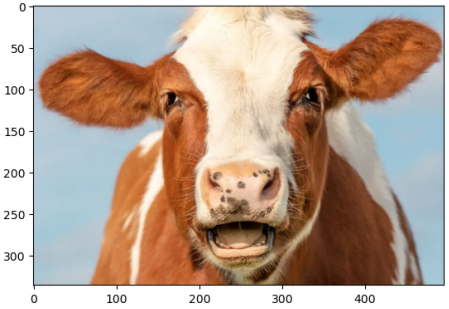
3.2. Importation & Sauvegarde d'une image

L'objectif de cette sous-section est de voir comme on peut importer une image depuis l'arborescence vers un tableau numpy, de forme précédente.

Exemple 4 **(Importation d'une image couleur sur un exemple)**

- Après avoir copié l'image `Vache.png` du cahier de prépa (dans le dossier « Données ») vers le dossier contenant votre fichier Python, on importe l'image puis on exécute en faisant « Ctrl+Shift+E » à l'aide du code suivant :

```
import numpy as np
import matplotlib.pyplot as plt
M = plt.imread('Vache.png') lecture de l'image et stockage dans un tableau numpy M
plt.imshow(M) on peut la réafficher si on veut
plt.imshow(M, cmap = "gray") ou avec cmap, dans le cas d'un niveau de gris

plt.show()
```



- Analysons le tableau en question.

```
>>> type(M) le résultat est bien un tableau numpy
<class 'numpy.ndarray'>
>>> np.shape(M) à 336 lignes x 496 colonnes
(336, 496, 3)
>>> M[0, 0] un pixel à 3 coordonnées (codage RGBA, flottants dans [0, 1])
array([0.7294118 , 0.8       , 0.84313726], dtype=float32)
>>> N = M[:,:,:3] on ne garde que les coordonnées RGB si besoin (cordonnées 0, 1, 2), mais inutile ici
>>> N[0, 0]
array([0.7294118 , 0.8       , 0.84313726], dtype=float32)
```

Bien entendu, si les pixels ont déjà 3 coordonnées, on travaille directement avec `M` plutôt que `N`.

Remarque 5 **(Convertir directement l'image png en RGB)** On peut utiliser le module `PIL` afin de modifier directement l'image de départ, afin qu'elle soit convertie en un tableau sans opacité.

RGBA vers RGB

```
import PIL.Image as pim
im_RGBA = pim.open('Vache.png')
im_RGB = im_RGBA.convert('RGB')
im_RGB.save("Vache.png")
```

Ce sont ces commandes qui ont servi à créer l'image `Vache.png`.

```
import numpy as np
import matplotlib.pyplot as plt
M_vache = plt.imread('Vache.png')
plt.imshow(M)
```

Exemple 5 (Sauvegarde d'une image sur un exemple) Pour sauvegarder un tableau sous forme d'une image, on utilise les commandes suivantes.

```
plt.savefig("nom_image_voulu.png") # pour une image couleur
plt.savefig("nom_image_voulu.png", cmap = "gray") # pour une \
↪ image noir et blanc
```

Cadre

Dans toute la suite, le tableau `M_vache` fera référence au tableau `M` importé précédemment.

3.3. Quelques transformations

Maintenant que nous savons gérer des images en Python (importation et exportation), nous allons étudier quelques algorithmes permettant de convertir une image en une autre.

Il est important de bien aborder les exercices dans cet ordre

3.3.1. Sur les couleurs

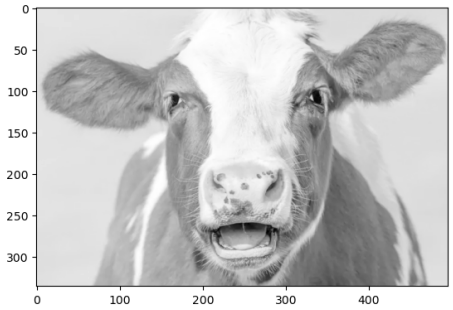
Exercice 16 | Convertir une image couleur en niveau de gris Solution Il est possible de convertir une image couleur RGB en image en niveau de gris en moyennant chaque pixel, c'est-à-dire en créant un pixel de valeur :

$$\text{Gris} = \frac{1}{3}\text{Rouge} + \frac{1}{3}\text{Vert} + \frac{1}{3}\text{Bleu}.$$

En utilisant `np.mean`, qui retourne la moyenne d'un tableau, compléter la fonction `niveaux_gris` prenant en argument un tableau `M` correspondant à une image couleur, et renvoyant un nouveau tableau satisfaisant cette règle. Utilisez cette fonction sur `M_vache` et affichez l'image correspondante, sauvegardez-la sous le nom `Vache_NG.png`.

```
def niveaux_gris(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_ng = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            pixel = ________
            M_ng[i, j] = np.mean( ______ )
    return  ____
```

Voici ce que vous devriez obtenir :



Exercice 17 | Convertir une image en niveaux de gris en noir et blanc par seuillage Solution

1. L'idée est ici de mettre à 1 tous les pixels qui sont supérieurs à une valeur fixée $m \in [0,1]$, et 0 pour les autres. Écrire une fonction d'en-tête `seuillage(M, m)` effectuant cette opération, elle retournera donc un nouveau tableau.
2. L'utiliser sur la vache en niveaux de gris, faire varier le paramètre en utilisant notamment la moyenne de `M` (avec `np.mean`) pour seuille `m`.

Exercice 18 | Saturer une couleur Solution

1. Dans `M_vache`, on peut affecter la valeur 1 au pixel bleu. Écrire une fonction d'en-tête `sature_bleu(M)` qui effectue cette opération. On dit que l'on a « saturé le bleu ». *On commencera par créer une copie du tableau `M` en écrivant : `M_b = np.copy(M)`, puis on le parcourera, et on modifiera la bonne composante.*

2. Faire afficher l'image associée à `sature_bleu(M_vache)`.
3. Comment saturer le rouge ? Le vert ?

Exercice 19 | **Inversion** Solution

1. Soit `P = np.array([[1, 2, 3]])`. Observez l'effet de la commande `1 - P` dans la console.
2. Écrire une fonction `inversion` qui prend en argument un tableau `M`, et qui retourne un autre tableau où chaque valeur x d'un pixel est remplacée par $1-x$, sur chaque composante en cas d'image couleur.
3. L'appliquer à `M_vache`, puis à sa version noir et blanc.

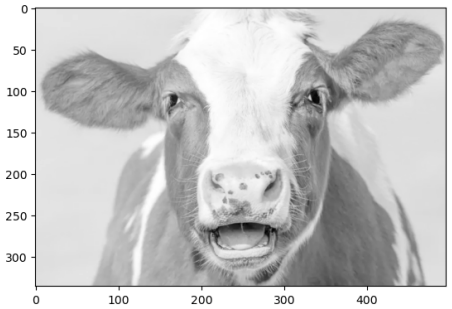
3.3.2. Sur le contraste

On peut essayer d'améliorer le contraste d'images, tout se fait en appliquant des fonctions bien choisies aux pixels au but d'exagérer les différences entre les niveaux. On détaille uniquement le contraste au travers d'un exemple.

Exemple 6 **(Contraste en appliquant une fonction)** On travaille ici avec la version en noir et blanc, donc `M_ng_vache`.

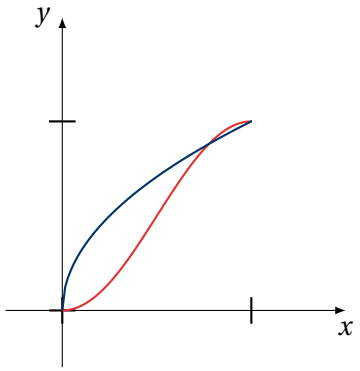
- Essayons dans un premier temps d'appliquer la fonction racine à chaque pixel.)

```
def contraste_1(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_b = np.copy(M)
    for i in range(n):
        for j in range(p):
            pixel = M_b[i, j]
            M_b[i, j] = np.sqrt(pixel)
    return M_b

M_ng_vache_contr1 = contraste_1(M_ng_vache)
plt.imshow(M_ng_vache_contr1, cmap = 'gray')
```



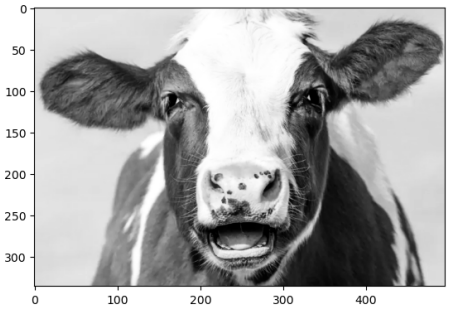
En comparant avec l'image originale, cette fonction ne semble pas vraiment augmenter le contraste.

- On considère $f : x \longmapsto \frac{1}{2} + \frac{1}{2}\sin(\pi(x-1/2))$ (courbe en rouge), que l'on peut par exemple tracer avec la racine (courbe en bleu) sur $[0,1]$.



La racine carrée croît très vite au voisinage de 0 (pixels noirs) donc elle a tendance à éclaircir l'ensemble, c'est ce que nous avons observé). La seconde fonction est beaucoup mieux car elle est proche de zéro au voisinage de zéro (maintient le noir), et proche de 1 au voisinage de 1 (maintient le blanc) et croit très vite entre (donc les pixels entre 0 et 1 se rapprochent sensiblement de 0 ou de 1).

-
```
def contraste_2(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_b = np.copy(M)
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            M_b[i, j] = 1/2+1/2*np.sin(np.pi*(pixel-1/2))
    return M_b
M_ng_vache_contr2 = contraste_2(M_ng_vache)
plt.imshow(M_ng_vache_contr2, cmap = 'gray')
```



Remarque 6 Il existe d'autres méthodes pour contraster une image. Par exemple, la méthode de convolution par un masque, qui ne sera pas vue dans ce TP.

3.4. Croissance de blob

Cette partie optionnelle n'est à traiter que si tout le reste a été terminé.

Exercice 20 | Le blob Solution Le *Physarum polycephalum*, plus communément appelé « blob », est une espèce de myxomycète faisant partie de la famille des Physaraceae et du règne des Amoebozoaires (comme l'amibe). Ce curieux organisme est composé d'une seule et unique cellule géante. Bien que dépourvu de cerveau ou de système nerveux, cet organisme vivant est tout de même capable d'apprendre. Ce n'est ni un animal, ni une plante, ni un champignon. Il vit dans les sous-bois depuis plus d'un milliard d'années. À l'état naturel, il se nourrit de bactéries et de moisissures (champignons). En laboratoire, les scientifiques leur donnent des flocons d'avoine, mais ils se nourrissent en fait des bactéries présentes sur l'avoine. Il existe plus de 1 000 espèces différentes de blob.

Vous trouverez dans le répertoire « Données » de cahier de prépa des images de la forme `Blob_xxx.png`, montrant l'évolution (schématique) d'un blob où xxx désigne la durée en heures depuis le début de l'expérience. L'objectif de cet exercice est de mesurer la proportion de surface colonisée par le blob en un temps donné, puis de la tracer en fonction des temps relevés. Il faudra bien faire attention au fait suivant : la surface totale colonisable est donc celle en **gris clair**, alors que celle en gris foncé n'est pas accessible.

1. Importer et lire l'image `Blob_11.png`, ne garder qu'un tableau avec les trois premières coordonnées de chaque pixel. Passer la souris sur le jaune foncé, le jaune clair, le gris clair, et le gris foncé, la valeur du pixel s'affiche en haut de la fenêtre.

IMAGE (SCHÉMATIQUE) DU BLOB AU TEMPS $t = 11$ HEURES

Notez ces valeurs dans des tableaux numpy que vous appellerez `J, J_c, G, G_c`. On créera en plus le pixel noir `N`, et le pixel blanc `B`.

2. Que fait la fonction suivante pour deux tableau numpy X, Y ? S'ils sont de longueur 2 avec `X = [x_1, x_2]`, `Y = [y_1, y_2]`, à quoi correspond `distance(X, Y)` ? Que signifie sur les coordonnées de `X` et `Y` que `distance(X, Y)` est très petit ?

```
def distance(X, Y):
    return np.sqrt(np.sum((X-Y)**2))
```

3. Recopiez et complétez la fonction suivante pour que, étant donné un tableau numpy `M` et un réel positif `prec`, retourne un autre tableau de même taille où :
   - un pixel de `M` noté `pixel` tel que `distance(pixel, J) < prec` ou `distance(pixel, J_c) < prec` est mis à la valeur `J`,
   - un pixel de `M` noté `pixel` tel que `distance(pixel, G_c) < prec` ou `distance(pixel, B) < prec` est mis à la valeur `G_c`,
   - et enfin un pixel de `M` noté `pixel` tel que `distance(pixel, G) < prec` est mis à la valeur `N`.

   Affichez alors l'image obtenue par cette fonction, pour plusieurs précisions, commentez. Dans la suite, vous choisirez alors une précision qui vous semble optimale.
4. Modifiez la fonction précédente pour qu'elle retourne en plus la proportion de surface colonisée par le blob et la surface colonisable.

4. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé

```
>>> A = 2*np.ones((3, 3))
>>> B = np.ones((4, 4)) - np.eye(4)
```

Solution (exercice 2) Énoncé Les matrices ont des grands formats, donc on part d'une matrice nulle que l'on complète correctement. Attention au décalage entre les indices Maths / Python.

```
def creer_matrice_A(n):
    A = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            A[i, j] = (i+1)*(j+1)
    return A

def creer_matrice_B(n):
    B = np.zeros((n, n))
    for i in range(n):
        for j in range(i, n):
            # j >= i uniquement
            B[i, j] = (i+1)**2-(j+1)**2
    return B
>>> creer_matrice_A(4)
array([[ 1.,  2.,  3.,  4.],
       [ 2.,  4.,  6.,  8.],
       [ 3.,  6.,  9., 12.],
       [ 4.,  8., 12., 16.]])
>>> creer_matrice_B(4)
array([[  0.,  -3.,  -8., -15.],
       [  0.,   0.,  -5., -12.],
       [  0.,   0.,   0.,  -7.],
       [  0.,   0.,   0.,   0.]])
```

Solution (exercice 3) Énoncé

```
def somme(M):
    S = 0
    n, p = len(M), len(M[0])
    for i in range(n):
        for j in range(p):
            S += M[i, j]
    return S
```

```
def somme(M):
    S = 0
    for L in M:

# L est une ligne de M
        for x in L:
            S += x
    return S
```

```
>>> A = np.array([[1, 2], [3, 4]])
>>> somme(A)
10
```

Solution (exercice 4) Énoncé

1. La matrice $A = \begin{pmatrix} -1 & -3 & 3 \\ 3 & -7 & 3 \\ 6 & -6 & 2 \end{pmatrix}$ convient. On la code en python dans la question suivante.
2.

```
A = np.array([[-1, -3, 3], [3, -7, 3], [6, -6, 2]])

def puissance_mat(A, k):
    n, p = len(A), len(A[0])
    P = np.eye(n)
    for _ in range(k):
        P = P @ A
    return P
>>> puissance_mat(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
```

```
def val_xyz(n):
    P = puissance_mat(A, n)
    X_0 = np.array([[1], [1], [2]])
    X_n = P @ X_0
    return X_n[0], X_n[1], X_n[2]
>>> val_xyz(1)
(array([2.]), array([2.]), array([4.]))
>>> val_xyz(2)
(array([4.]), array([4.]), array([8.]))
```

```
>>> val_xyz(3)
(array([8.]), array([8.]), array([16.]))
>>> val_xyz(7)
(array([128.]), array([128.]), array([256.]))
>>> val_xyz(10)
(array([1024.]), array([1024.]), array([2048.]))
>>> val_xyz(50)
(array([1.12589991e+15]), array([1.12589991e+15]), \
↪ array([2.25179981e+15]))
```

Les suites semblent diverger vers $+\infty$.

3. La version récursive est basée sur la relation suivante : $\forall k \in \mathbb{N}, \quad A^{k+1} = A \times A^k$.

```
def puissance_mat_rec(A, k):
    n, p = len(A), len(A[0])
    if k == 0:
        return np.eye(n)
    else:
        return A @ puissance_mat_rec(A, k-1)
>>> puissance_mat(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
>>> puissance_mat_rec(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
```

Solution (exercice 5) Énoncé Un calcul simple montre que $M^3 = 0_{3,3}$ alors que $M^2 \neq 0_{3,3}$, elle est donc nilpotente d'ordre 3.

```
def indice_nilpo(M):
    ind = 1
    P = np.copy(M) # ou ici P = M n'est pas gênant
    while not np.all(P == np.zeros(P.shape)):
        P = P @ M
        ind += 1
    return ind
>>> M = np.array([[0, 1, 2], [0, 0, 3], [0, 0, 0]])
>>> indice_nilpo(M)
3
```

Solution (exercice 6) Énoncé L'idée est de parcourir la matrice et de renvoyer `False` dès qu'on trouve un coefficient non nul strictement en-dessous de la diagonale.

```
def est_triangulaire_sup(M):
    n = M.shape[0] # matrice carrée donnée en entrée, m = n
    for i in range(n):
        for j in range(i):
            if M[i, j] != 0:
                return False
    return True

def est_triangulaire_inf(M):
    return est_triangulaire_sup(np.transpose(M)) and \
    ↪ est_triangulaire_sup(M)

def est_diagonale(M):
    return est_triangulaire_sup(M) and \
    ↪ est_triangulaire_sup(np.transpose(M))
>>> A = np.array([[1, 1], [0, 1]])
>>> est_triangulaire_sup(A)
True
>>> est_triangulaire_inf(A)
False
>>> est_diagonale(A)
False
>>> B = np.eye(3)
>>> B
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> est_diagonale(B)
True
>>> est_triangulaire_sup(B)
True
>>> est_triangulaire_inf(B)
True
```

Solution (exercice 7) Énoncé

```
def creer_mat_entiers(n):
    M = np.zeros((n, n))
```

```
    m = 1
    for i in range(n):
        for j in range(n):
            M[i][j] = m
            m += 1
    return M
>>> creer_mat_entiers(3)
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

On peut aussi utiliser une expression explicite aussi :

```
def creer_mat_entiers(n):
    M = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            M[i][j] = (i*n)+(j+1)
    return M
>>> creer_mat_entiers(3)
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

D'après le cours de Maths, $\sum_{k=1}^{n^2} k = \frac{n^2(n^2+1)}{2}$. Donc par exemple 10 pour $n=2$, et 45 pour $n=3$.

```
>>> somme(creer_mat_entiers(2))
10.0
>>> somme(creer_mat_entiers(3))
45.0
```

Solution (exercice 8) Énoncé

```
def min_max(L):
    mini, maxi = L[0], L[0]
    for x in L[1:]:
        if x > maxi:
            maxi = x
        if x < mini:
            mini = x
    return mini, maxi
```

La fonction précédente fonctionne aussi très bien pour un tableau numpy qui ne comporte qu'une ligne.

```
>>> A = np.arange(0, 100, 3)
>>> B = (np.sin(A))**2
>>> B
array([0.00000000e+00, 1.99148567e-02, 7.80730206e-02, \
↪ 1.69841646e-01,
       2.87910496e-01, 4.22874275e-01, 5.63981845e-01, \
       ↪ 6.99992657e-01,
       8.20072170e-01, 9.14654916e-01, 9.76206490e-01, \
       ↪ 9.99823728e-01,
       9.83625294e-01, 9.28901547e-01, 8.40011748e-01, \
       ↪ 7.24036808e-01,
       5.90215225e-01, 4.49207148e-01, 3.12245201e-01, \
       ↪ 1.90239694e-01,
       9.29095147e-02, 2.80079304e-02, 7.04963780e-04, \
       ↪ 1.31755535e-02,
       6.44262995e-02, 1.50374597e-01, 2.64173853e-01, \
       ↪ 3.96758885e-01,
       5.37568045e-01, 6.75384557e-01, 7.99230035e-01, \
       ↪ 8.99239019e-01,
       9.67444853e-01, 9.98414297e-01])
>>> mini, maxi = min_max(B)
>>> maxi - mini
0.9998237279831751
```

Solution (exercice 9) Énoncé

1. On a $A = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}$, donc A est bistochastique, tout comme B.
2. L'idée est la suivante : on parcourt la somme des coefficients de chaque ligne et on retourne **False** dès que l'une des sommes est différente de 1, ou qu'un des coefficients n'est pas dans $[0,1]$. Sinon, on retourne **True** et la matrice sera bien stochastique.

```
def est_stochastique(M):
    n, p = len(M), len(M[0])
    for i in range(n):
        somme = 0
        for j in range(p):
            somme += M[i, j]
            if not 0 <= M[i, j] <= 1:
```

```
                return False
        if somme != 1:
            return False
    return True
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> est_stochastique(M)
True
>>> M = np.array([[0, 1/2], [0.1, 0.9]])
>>> est_stochastique(M)
False
```

3.
```
def est_stochastique_prod(M, N):
    P = M @ N
    return est_stochastique(P)
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> N = np.array([[0, 1], [1, 0]])
>>> est_stochastique_prod(M, N) # c'est gagné
True
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> N = np.array([[1, 0], [0.1, 0.9]])
>>> est_stochastique_prod(M, N) # encore gagné
True
```

On conjecture raisonnablement que le produit de deux matrices stochastiques est encore stochastique. Pour savoir si une matrice stochastique est bistochastique, il suffit de regarder si la transposée est stochastique.

```
def est_bistochastique(M):
    return est_stochastique(M) and \
    ↪ est_stochastique(np.transpose(M))
>>> M = np.array([[0, 1], [1, 0]])
>>> est_bistochastique(M)
True
>>> M = np.array([[0, 1/2], [0.1, 0.9]])
>>> est_bistochastique(M)
False
```

Solution (exercice 10) Énoncé

1.
```
import numpy as np
def hilbert(n):
    A = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            A[i,j] = 1/((i+1)+(j+1)-1)
    return A
```

2. $H_2 = \begin{pmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{pmatrix}$, $\det(H_2) = 1/3 - 1/4 \neq 0$ donc elle est bien inversible. Pour H_3, on échelonne :

3.
$$\begin{pmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/2 \end{pmatrix} \underset{L_3 \leftarrow L_3 - 1/3 L_1}{\overset{L_2 \leftarrow L_2 - 1/3 L_1}{\sim}} \begin{pmatrix} 1 & 1/2 & 1/3 \\ 0 & 1/6 & 5/36 \\ 0 & 1/12 & 5/12 \end{pmatrix} \underset{\sim}{L_3 \leftarrow L_3 - 1/2 L_2} \begin{pmatrix} 1 & 1/2 & 1/3 \\ 0 & 1/6 & 5/36 \\ 0 & 0 & 25/72 \end{pmatrix}.$$

Donc : $\mathrm{Rg}(H_3) = 3$ et H_3 est donc inversible. *La preuve de l'inversibilité dans le cas général nécessite des théorèmes de 2ème année ; nous l'admettons.*

4. On tape simplement : `np.linalg.matrix_rank(hilbert(10))` qui retourne 10.

5. En testant pour les valeurs de $n \leqslant 10$, on remarque que la matrice de HILBERT semble de rang n. En particulier elle est donc inversible.
Cependant dès que $n > 10$, on voit que les rangs ne sont plus égaux à n. On pourrait ainsi en déduire que `hilbert(n)` n'est plus inversible pour ces n, sauf que mathématiquement nous sommes capables de démontrer le contraire. En fait, la responsable dans l'histoire est la méthode de calcul du rang utilisée par Python (gardez donc un regard critique des résultats), qui fait intervenir des divisions et des flottants.

Solution (exercice 11) Énoncé

```
>>> import numpy as np
>>> A = np.array([[2,2,-3], [-2,-1,-3], [6,4,4]]) # la \
↪ matrice du système
>>> B = np.array([[2], [-5], [16]]) # le second membre
>>> X = np.linalg.solve(A, B)
>>> X
array([[-14.],
       [ 21.],
       [  4.]])
>>> A@X - B # vérification. OK si résultat proche du vecteur \
↪ nul.
array([[-8.88178420e-16],
       [ 2.66453526e-15],
```

```
       [-3.55271368e-15]])
```

Solution (exercice 12) Énoncé

```
def permut(M, i, j):
    """
    M->tableau où Li Lj sont permutées
    modifie M
    """
    p = len(M[0])
    for k in range(p):
        M[i,k], M[j,k] = M[j,k], M[i,k]

def dilatation(M, i, lamba):
    """
    M->tableau où Li<- lamba Li
    modifie M
    """
    p = len(M[0])
    for j in range(p):
        M[i, j] = lamba*M[i, j]

def transvection(M, i, j, lamba):
    """
    M->tableau où Li<- Li+lamba Lj
    """
    p = len(M[0])
    for k in range(p):
        M[i, k] += lamba * M[j, k]
```

Faisons quelques tests.

```
M = np.array([[1, 2], [2, 3]], dtype = 'float') # il faut \
↪ imposer un type flottant, sinon les coefficients seront \
↪ toujours considérés comme des entiers et seront arrondis à \
↪ chaque opération
permut(M, 0, 1)
```

La matrice vaut alors [[2. 3.] [1. 2.]].

```
M = np.array([[1, 2], [2, 3]], dtype = 'float')
dilatation(M, 0, 10)
```

La matrice vaut alors [[10. 20.] [2. 3.]].

Solution (exercice 13) Énoncé

1. Lorsque l'algorithme se trouve en la j-ème colonne, on souhaite éliminer uniquement les coefficients sous le pivot, on ne touche pas à ceux du dessus ! On recherche donc le plus grand coefficient en valeur absolue [a] que l'on considère comme un pivot.
2.
```
def rang_pivot(M, j):
    n = np.shape(M)[0]
    maxi = np.abs(M[j, j])
    ind_maxi = j
    for i in range(j+1, n):
        if np.abs(M[i, j]) > maxi:
            maxi = np.abs(M[i, j])
            ind_maxi = i
    return ind_maxi
M = np.array([[-2, 2], [-1, 3]])
```

Avec la matrice de test précédente, nous obtenons pour `rang_pivot(M, 0)` : 0. Avec l'algorithme précédent, on retourne le plus petit $i^\star$ possible.

Solution (exercice 14) Énoncé

```
def echelonnement(M):
    """
    modifie M pour avoir sa version échelonnée
    """
    n = len(M)
    N = np.eye(n, n)
    for j in range(n):
        i_star = rang_pivot(M, j)
        # On place le pivot au bon endroit
        permut(M, i_star, j)
        permut(N, i_star, j)
        # Élimination en-dessous du pivot
        for k in range(j+1, n):
            lamba = - M[k, j]/M[j, j]
            transvection(M, k, j, lamba)
            transvection(N, k, j, lamba)
    return N
M = np.array([[1, 2], [2, 3]], dtype = 'float')
N = echelonnement(M)
```

a. choix qui permet d'avoir un algorithme plus efficace, mais n'importe quel coefficient non nul ferait l'affaire

Pour cette matrice, la fonction a transformé `M` en [[2. 3.] [0. 0.5]]. La matrice `N` vaut [[0. 1.] [1. -0.5]].

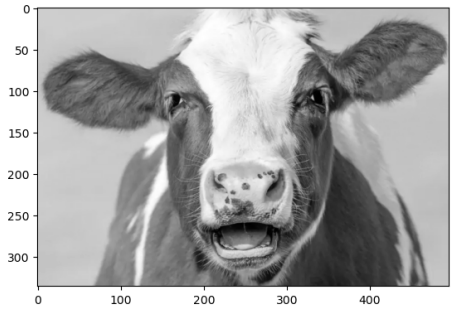
Solution (exercice 15) Énoncé

```
def echelonnement_reduit(M):
    """
    modifie M pour avoir sa version échelonnée
    """
    n = len(M)
    N = echelonnement(M)

    #On fait apparaitre des pivots égaux à 1
    for i in range(n):
        lamba = 1/M[i, i]
        dilatation(M, i, lamba)
        dilatation(N, i, lamba)

    # On fait apparaître des zéros au-dessus des pivots, en \
    ↪ commençant par la dernière colonne
    for j in range(n-1, 0, -1):
        for k in range(j-1, -1, -1):
            lamba = -M[k,j]
            transvection(M, k, j, lamba)
            transvection(N, k, j, lamba)
    return N
M = np.array([[1, 2], [2, 3]], dtype = 'float')
N = echelonnement_reduit(M)
```

Pour cette matrice, la fonction a transformé `M` en [[1. 0.] [0. 1.]]. La matrice `N` vaut [[-3. 2.] [2. -1.]]. On peut ensuite vérifier que l'inverse convient, en effet `N@np.array([[1, 2], [2, 3]])` retourne : [[1. 0.] [0. 1.]].

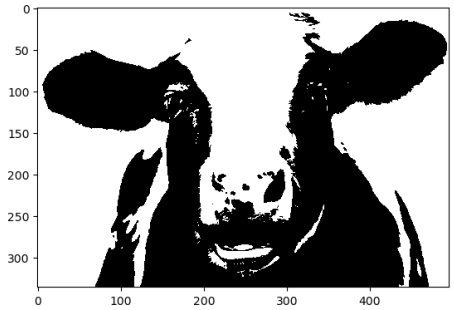
Solution (exercice 16) Énoncé

```
M_vache = plt.imread('Vache.png')

def niveaux_gris(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_nb = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            M_nb[i, j] = np.mean(pixel)
    return M_nb

M_ng_vache = niveaux_gris(M_vache)
plt.imshow(M_ng_vache, cmap = 'gray')
```



Solution (exercice 17) Énoncé

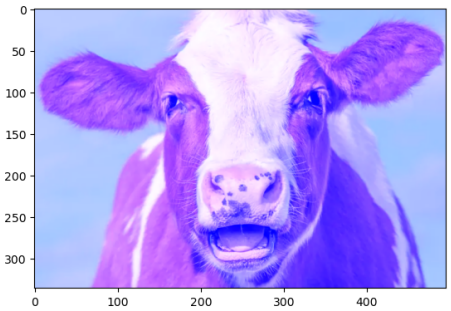
```
def seuillage(M, m):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_nb = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            if pixel > m:
                M_nb[i, j] = 1
    return M_nb

M_nb_vache = seuillage(M_ng_vache, np.mean(M_ng_vache))
plt.imshow(M_nb_vache, cmap = 'gray')
```



Solution (exercice 18) Énoncé

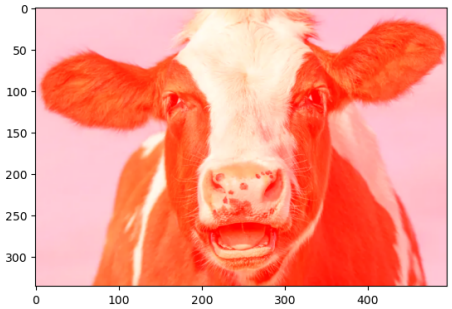
1.
```
def sature_bleu(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_b = np.copy(M)
    for i in range(n):
        for j in range(p):
            M_b[i, j][2] = 1
    return M_b
```

2.
```
M_b_vache = sature_bleu(M_vache)
plt.imshow(M_b_vache)
```



3. Pour saturer les autres couleurs, on vient modifier les autres pixels, par exemple le rouge :
```
def sature_rouge(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_b = np.copy(M)
    for i in range(n):
        for j in range(p):
            M_b[i, j][0] = 1
    return M_b
```

4.
```
M_r_vache = sature_rouge(M_vache)
plt.imshow(M_r_vache)
```

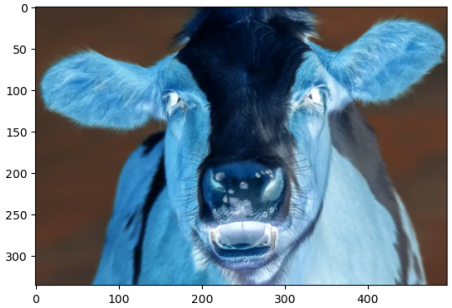


Solution (exercice 19) Énoncé

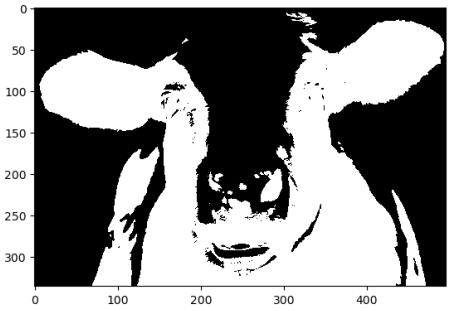
1. Cela applique sur chaque coordonnée la fonction $x \longmapsto 1-x$.
```
>>> P = np.array([[1, 2, 3]])
>>> 1-P
array([[ 0, -1, -2]])
```

2.
```
def inversion(M):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_inv = np.copy(M)
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            M_inv[i, j] = 1 - pixel
    return M_inv
```

3.
```
M_inv_vache = inversion(M_vache)
plt.imshow(M_inv_vache)
```

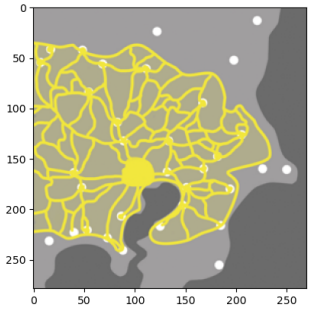


```
M_invnb_vache = inversion(M_nb_vache)
plt.imshow(M_invnb_vache, cmap = 'gray')
```



Solution (exercice 20) Énoncé En affichant l'image comme d'habitude, on trouve les valeurs des pixels ci-après.

```
blob_RGBA = plt.imread('Blob_11.png')
blob_11 = blob_RGBA[:,:,:3]
plt.imshow(blob_11)
```



```
J = np.array([0.949, 0.906, 0.243])
J_c = np.array([0.686,0.675,0.553])
G_c = np.array([0.627, 0.62, 0.624])
G = np.array([0.42, 0.42, 0.42])
B = np.array([1, 1, 1])
N = np.array([0, 0, 0])
```

La fonction distance retourne la racine de la somme des carrés des différences des coefficients. Plus concrètement, si les vecteurs sont des lignes de taille 2, alors il s'agit de la distance euclidienne entre les deux vecteurs. Avec les notations de l'énoncé :

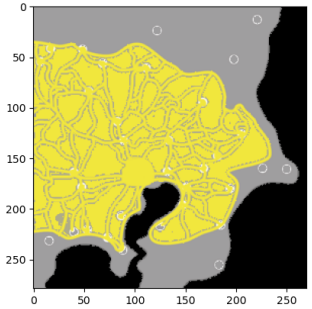
$$\text{distance}(X,Y) = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2}.$$

Si `distance(X,Y)` est très petite, alors les coefficients des deux vecteurs (ou des deux matrices) seront très proches. Dans la suite, les vecteurs en question seront des pixels ! La distance va alors nous servir à détecter les couleurs très proches.

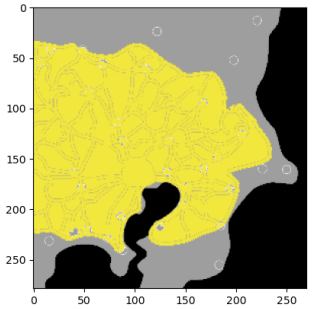
```
def distance(X, Y):
    return np.sqrt(np.sum((X-Y)**2))

def garde_jaune(M, prec):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_jaune = np.zeros(M.shape)
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            if distance(pixel, J) < prec or distance(pixel, \
            ↪ J_c) < prec:
                M_jaune[i,j] = J
            elif distance(pixel, G_c) < prec or \
            ↪ distance(pixel, B) < prec:
                M_jaune[i,j] = G_c
            elif distance(pixel, G) < prec:
                M_jaune[i,j] = N
            else:
                M_jaune[i,j] = pixel
    return M_jaune
```

Observons à présent l'image obtenue par exemple au temps 11, et ce pour plusieurs précisions.

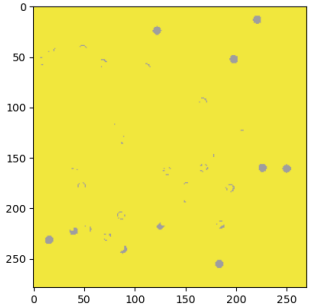
```
M_jaune_0 = garde_jaune(blob_11, 0.01)
plt.imshow(M_jaune_0)
```



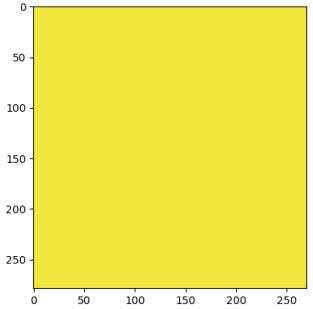
```
M_jaune_1 = garde_jaune(blob_11, 0.1)
plt.imshow(M_jaune_1)
```



```
M_jaune_2 = garde_jaune(blob_11, 0.5)
plt.imshow(M_jaune_2)
```

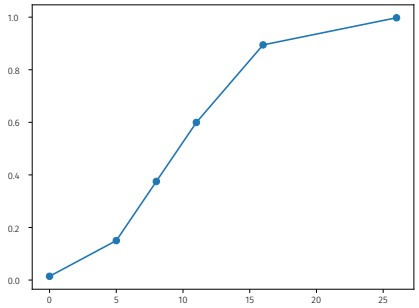


```
M_jaune_3 = garde_jaune(blob_11, 1)
plt.imshow(M_jaune_3)
```



Une précision trop élevée donne bien sûr une image jaune, puisqu'alors le premier test **if** est toujours satisfait. On choisit alors `prec = 0.1` pour la suite. On modifie la fonction comme suit.

```
def garde_jaune(M, prec):
    n = len(M) # nb de lignes
    p = len(M[0]) # nb de colonnes
    M_jaune = np.zeros(M.shape)
    nb_blob = 0
    nb_vide = 0
    for i in range(n):
        for j in range(p):
            pixel = M[i, j]
            if distance(pixel, J) < prec or distance(pixel, \
            ↪ J_c) < prec:
                M_jaune[i,j] = J
                nb_blob += 1
            elif distance(pixel, G_c) < prec or \
            ↪ distance(pixel, B) < prec:
                M_jaune[i,j] = G_c
                nb_vide += 1
            elif distance(pixel, G) < prec:
                M_jaune[i,j] = N
            else:
                M_jaune[i,j] = pixel
    return M_jaune, nb_blob/(nb_blob+nb_vide), \
    ↪ nb_vide/(nb_blob+nb_vide)

def trace():
    temps = [0, 5, 8, 11, 16, 26]
    P_blob = []
    for t in temps:
        blob_RGBA = plt.imread("Blob_%s.png" % t)
        blob = blob_RGBA[:,:,:3]
        P_t = garde_jaune(blob, 0.1)[1]
        P_blob.append(P_t)
    plt.plot(temps, P_blob, marker = 'o')

trace()
```



Il nous faudrait plus de points pour être plus convaincant, mais la courbe ressemble fortement à celle d'un modèle logistique (dynamique avec capacité de milieu).