ANNÉE UNIVERSITAIRE : 2025 – 2026
UNITÉ DE RECHERCHE : Larema
ÉQUIPE : Analyse & Probabilités
SUJET : Programmation sous Python

SOLUTIONS DES EXERCICES DU TP3 [1]

Mohamed Alfaki ABOUBACRINE ASSADEK

SOLUTION DE L'EXERCICE 2 DU TP3

```python
# Un developement de la methode du pivot de Gauss basee sur les operations avec les lignes. La fonction principale est pivotDeGauss qui prend comme argument une matrice. Il y a aussi une fonction qui calcule le determinant d'une matrice (caree).

import numpy as np

# Retourne la matrice inversible tmpP = I_n+X, ou X a une seule composante non nulle, X[i, j]= x. Remrquez que le produit tmpP*A est la matrice A avec A[i,:], i.e. la ligne i, remplacee par A[i,:] + x*A[j,:]. Des matrices de type tmpP seront utilisees pour modifier A en appliquant le pivot de Gauss avec la ligne j comme ligne du pivot (le pivot sera un element de cette ligne). L'element x dependra de la matrice A qu'on voudra echelonner.
def tmpP(n, i, j, x):
    P = np.eye(n)
    P[i, j] = x
    return P


# A est la matrice qu'on veut echelonner ; posPivot est la position du pivot, i.e. une liste [i, j] ; le troisieme argument ligne (un entier) designe la ligne de A qui sera modifiee par un produit a gauche du type U*A.
def lElement_x(A, posPivot, ligne):
    lPivot, cPivot = posPivot
    pivot = A[lPivot, cPivot]
    return -A[ligne, cPivot]/pivot


# A est une matrice et posPivot la position d'un pivot.  La fonction retourne la matrice obtenue en utililsant la ligne du pivot. Le coefficient x avec lequel une ligne i sera modifiee en appliquant l'algorithme du pivot est renvoye par la fonction lElement_x. Le pivot n'est pas change !
def utilisationDUnPivot(A, posPivot):
    lPivot, cPivot = posPivot
    n = A.shape[0]
    for i in range(n):
        if i != lPivot:
            U = tmpP(n, i, lPivot, lElement_x(A, posPivot, i))
            A = U.dot(A)
    return A


# Retourne un element maximal et sa position dans A_2darray.  L'argument A_2darray est un tableau ayant deux dimensions (i.e. une matrice). C'est cette position qui deviendra le prochain pivot dans le developement de l'algorithme de Gauss.
def maxEtPosition(A_2darray):
    indices = np.where(A_2darray == A_2darray.max())
    # print(indices, "\n")  # il faut comprendre la sortie de np.where !
    position = [indices[0][0], indices[1][0]]
    return A_2darray[position[0], position[1]], position
```

1. Voir TP3 pour les énoncés

```python


# Remplace par des zeros les elements correspondant aux lignes et colonnes des positions de positions_list.  Enleve les signes des autres termes.
def zeroSurLignesEtColonnes(A, positions_list):
    B = abs(A)
    for pos in positions_list:
        B[pos[0], :] = 0
        B[:, pos[1]] = 0
    return B


#Le pivot de Gauss en utilisant les lignes.  La fonction suit les operations qu'un humain ferait. Le deuxieme argument controle l'egalite avec 0. La fonction renvoie la matrice "echelonnee" (sans changer la position des pivots lors de la construction) ainsi que la liste des positions des pivots utilises.
def pivotDeGauss(A, eps=.1**14):
    posPivots_list = []
    tmp_array = abs(A)
    a, pPivot = maxEtPosition(tmp_array)
    while a>eps:  # a est toujours >= 0
        posPivots_list.append(pPivot)
        A = utilisationDUnPivot(A, pPivot)
        tmp_array = zeroSurLignesEtColonnes(A, posPivots_list)
        a, pPivot = maxEtPosition(tmp_array)
    return A, posPivots_list


# Calucl du determinant d'une matrice par la methode du pivot de Gaus : produit au signe pres des elements diagonaux de la matrice echelonnee.
def determinant(A):
    E, posPivots_list = pivotDeGauss(A)
    if np.shape(A)[0] != np.shape(A)[1]:
        return False
    if len(posPivots_list)==np.shape(A)[0]:
        d = 1.
        for p in posPivots_list:
            d = d*(-1)**(p[0]+p[1])*E[p[0], p[1]]
    else:
        d = 0.
    return d


# Transforme les coefficients dont la valeur absolue est <eps en 0 et montre la matrice dans le terminal.
def montrerMatrice(A, eps=.1**8):
    m, n = np.shape(A)
    C = np.zeros((m, n))
    for i in range(m):
        for j in range(n):
            if abs(A[i, j])>eps:
               C[i, j] =  A[i, j]
    print(C)



# tests numeriques
A = np.array([[1, 2, 3, -4], [5, -7, 9, 11], [0, 0, 0, 2], [0, -1, 0, 3]])
B = np.array([[1, -8, 3], [-1, 4, 8], [0, 0, 0]])
C = np.array([[2, -1, 0, 1, 0, 0], [-1, 2, -1, 0, 1, 0], [0, -1, 2, 0, 0, 1]])
M = B
E, listePositionsPivots = pivotDeGauss(M)
print(f"\npour la matrice \n{M} \nune forme echelonee est")
montrerMatrice(E)
print(f"\nde plus, son determinant est {determinant(M)}")
print(np.linalg.det(M), " est le dereminant calcule par linalg")
```

SOLUTION DE L'EXERCICE 3 DU TP3

```python
import numpy as np

# L est une liste ou un vecteur et la sortie est la matrice circulante dont la premiere ligne est donnee par L en utilisant le produit matriciel.
def matriceCirculante(L):
    n = len(L)
    J = np.eye(n, n, k=1)
    J[n - 1, 0] = 1
    Jk = np.eye(n, n)
    C = 0
    for k in range(n):
        C = C + L[k]*Jk
        Jk = Jk.dot(J)
    return C


# L est une liste et la sortie est la matrice circulante dont la premiere ligne est donnee par L sans utiliser le produit matriciel.
def matriceCirculante_v2(L):
    n = len(L)
    R = L.copy()
    C = np.zeros((n, n))
    for k in range(n):
        C[k, :] = R
        R = [R[-1]] + R[0:-1]
    return C

# test
L = [-1, 1, 2, 4]
print(f"matrice circulante a partir de {L}, \n{matriceCirculante_v2(L)}\n")


# L est une liste ou un vecteur et la sortie est la matrice de Vandermond dont la deuxieme colonne est donnee par L.

def matriceVandermond(L):
    n = len(L)
    if type(L)=='list':
        R = np.array(L)
    else:
        R = np.array(list(L))
    V = np.ones((n, n))
    for j in range(1, n):
        V[:,j] = V[:,j - 1] * R
    return V


# L est une liste ou un vecteur et la sortie est une valeur conjecturee du determinant de la matrice Vandermond dont la deuxieme colonne est donnee par L.
def determinantVandermond(L):
    D = 1
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            D = D * (L[j] - L[i])
    return D

# test
L = [-1, 1, 2, 4]
V = matriceVandermond([-1, 1, 2, 4])
print('V =', V, '\n')
print('det de V =', np.linalg.det(V))
print('det de V conjecture =', determinantVandermond(L))
```

SOLUTION DE L'EXERCICE 4 DU TP3

```python
import matplotlib.pyplot as plt

# Fonction carre
X = np.linspace(-2, 2, 1001)
Y = X**2
plt.plot(X, Y, color="red", linestyle="dashdot", linewidth=3)
plt.title("Graphique de la fonction carre")
plt.xlabel("Abscisses x")
plt.ylabel("Ordonnees y")
plt.grid()

# f(x) = cos x sur [0, 2pi]
X = np.linspace(0, 2*np.pi, 1001)
Y = np.cos(X)
plt.plot(X, Y, color="green")
plt.title("Graphique de f(x)=cos x")
plt.xlabel("Abscisses x")
plt.ylabel("Ordonnees y")
plt.grid()


# f(x) = sin x et g(x) = sin x + sin 2x sur [-A, A]
def plotSommeSin2(A):
    X = np.linspace(-A, A, 1001)
    Y1 = np.sin(X)
    Y2 = Y1 + np.sin(2*X)
    plt.plot(X, Y1, color="red", label="sin x")
    plt.plot(X, Y2, color="mediumpurple", label="sin x + sin 2x")
    plt.title(f"Superposition de sinus sur [{-A},{A}]")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    plt.legend()

plotSommeSin2(5)


# Somme de N composantes sinusoidales sur [-A, A]
def plotSommeSin(N, A):
    X = np.linspace(-A, A, 1001)
    Y = 0*X # Vecteur nul de meme dimension que X
    for j in range(1, N+1):
        Y += np.sin(j*X)
    plt.plot(X, Y, color="green")
    plt.title(f"{N} composantes sur [{-A},{A}]")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()

# test
plotSommeSin(10, 5)
```

SOLUTION DE L'EXERCICE 5 DU TP3

```python
import matplotlib.pyplot as plt

# Lemniscate de Bernoulli
def plotLemniscate(a):
    T = np.linspace(0, 2*np.pi, 1001)
    X = a*np.sqrt(2)*np.cos(T)/(1+np.sin(T)**2)
    Y = X*np.sin(T)
    plt.plot(X, Y)
    plt.title(f"Lemniscate de Bernoulli avec a={a}")
    plt.xlabel("x(t)")
    plt.ylabel("y(t)")
```

```
    plt.grid()

# test
plotLemniscate(-2)


# Plusieurs lemniscates
def plotLemniscates(LstA):
    T = np.linspace(0, 2*np.pi, 1001)
    for a in LstA:
        X = a*np.sqrt(2)*np.cos(T)/(1+np.sin(T)**2)
        Y = X*np.sin(T)
        plt.plot(X, Y, label=f"a={a}")
    plt.title("Plusieurs lemniscates")
    plt.xlabel("x(t)")
    plt.ylabel("y(t)")
    plt.grid()
    plt.legend()

# |a| < 1 contracte
# |a| > 1 dilate
# a = 0 : lemniscate degeneree sur l'origine (0, 0)
# le signe de a n'a aucune importance
# tests
plotLemniscates([0.2, 1, 3])
plotLemniscates([0, 0.2, 1, 3])
```

SOLUTION DE L'EXERCICE 6 DU TP3

```
import numpy as np
import matplotlib.pyplot as plt

# Les premiers pas (dessins de la parabole et de quelques rayons reflechis)
fig = plt.figure(figsize=(7, 8))
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-2.2, 2.2)
ax.set_ylim(-.5, 6)


nbSteps = 80
x_parabola = np.linspace(-2, 2, nbSteps)
ax.plot(x_parabola, x_parabola**2)


alpha = -np.pi/2 + .1  # angle mesure par rapport a Oy
a = -alpha + np.pi/2
print(f"angle du vecteur incident = {a*180/np.pi}")
u = np.array([np.cos(a), np.sin(a)])  # vecteur d'incident -- non utilise


N = 11
X = np.linspace(-2, 2, N)
Y = X**2
title = f"L'angle en degres est {round(alpha, 2)} " + \
    f"et le nombre de rayons lumineux est {N}"
fig.suptitle(title)


for j in range(N):
    print('\nj =', j)
    x = X[j]
    y = Y[j]
    t = np.arctan2(2*x, 1)
    n = np.pi/2 + t  # angle du vecteur normal
    print(f"angle du vecteur normal = {n*180/np.pi}")
    r = 2*n - a
    print(f"angle du vecteur reflechi = {r*180/np.pi}")
```

```
    refl = np.array([np.cos(r), np.sin(r)])  # Vecteur de reflexion -- non utilise

    i_xx = [x - np.cos(a), x + np.cos(a)]
    i_yy = [y - np.sin(a), y + np.sin(a)]
    r_xx = [x - 5*np.cos(r), x + 5*np.cos(r)]
    r_yy = [y - 5*np.sin(r), y + 5*np.sin(r)]

    ax.plot(i_xx, i_yy, c = 'g', alpha=.2, linewidth=.8)
    ax.plot(r_xx, r_yy, c = 'r', alpha=.4, linewidth=.8)

plt.show()

# Dessins de la configuration pour un angle donne (apparition des fonctions dans le code
    precedent)

# arguments
nbSteps = 80  # pour la parabole
alpha = -np.pi/2 + .1  # angle mesure par rapport a l'axe Oy
N = 90  # rayons lumineux incidents
d = 5  # demi-longueur du rayon reflechi


# parabole
x_parabola = np.linspace(-2, 2, nbSteps)
y_parabola = x_parabola**2


# rayons lumineux
i_angle = -alpha + np.pi/2 # angle d'incidence mesure par rapport a l'axe Ox
X = np.linspace(-2, 2, N) # intersections avec la parabole
Y = X**2

t_angles = np.arctan2(2*X, 1)
n_angles = t_angles + np.pi/2
r_angles = 2*n_angles - i_angle

i_xx = np.zeros([2, N])  # matrice des vecteurs incidents ' x (en colonnes)
i_xx[0, :] = X - np.cos(i_angle)
i_xx[1, :] = X + np.cos(i_angle)

i_yy = np.zeros([2, N])  # matrice des vecteurs incidents ' x (en colonnes)
i_yy[0, :] = Y - np.sin(i_angle)
i_yy[1, :] = Y + np.sin(i_angle)

r_xx = np.zeros([2, N])  # matrice des vecteurs reflechis ' x (en colonnes)
r_xx[0, :] = X - d*np.cos(r_angles)
r_xx[1, :] = X + d*np.cos(r_angles)

r_yy = np.zeros([2, N])  # matrice des vecteurs reflechis ' x (en colonnes)
r_yy[0, :] = Y - d*np.sin(r_angles)
r_yy[1, :] = Y + d*np.sin(r_angles)


# Dessins

# figure et axes
fig = plt.figure(figsize=(7, 8))
ax = fig.add_subplot(111, aspect='equal')

ax.set_xlim(-2.2, 2.2)
ax.set_ylim(-.5, 6)
title = f"L'angle en degres est {round(alpha, 2)} " + \
    f"et le nombre de rayons lumineux est {N}"
fig.suptitle(title)

ax.plot(x_parabola, y_parabola)
# ax.plot(i_xx, i_yy, c = 'g', alpha=.5, linewidth=.8)
ax.plot(r_xx, r_yy, c = 'r', alpha=.3, linewidth=.8)

```

```
plt.show()


# Animation de la configuration en fonction de l'angle des rayons de lumiere (on fait varier l
    'angle dans le code precedent)

# l'angle alpha variera et, grace a cette variation, nous definirons une animation des rayons
    reflechis, donc de la caustique.

# arguments
nbSteps = 80  # pour la parabole
nbImag = 200
alpha = np.linspace(-np.pi/2, np.pi/2, nbImag)  # angles mesures par rapport a Oy
N = 90  # rayons lumineux incidents
d = 5  # demi-longueur du rayon reflechi


# parabole
x_parabola = np.linspace(-2, 2, nbSteps)
y_parabola = x_parabola**2


# rayons lumineux
rX_list = []
rY_list = []
X = np.linspace(-2, 2, N) # intersections avec la parabole
Y = X**2

for k in range(nbImag):
    i_angle = -alpha[k] + np.pi/2 # angle d'incidence mesure par rapport a Ox

    t_angles = np.arctan2(2*X, 1)
    n_angles = t_angles + np.pi/2
    r_angles = 2*n_angles - i_angle

    r_xx = np.zeros([2, N])  # matrice des vecteurs reflechis ' x (en colonnes)
    r_xx[0, :] = X - d*np.cos(r_angles)
    r_xx[1, :] = X + d*np.cos(r_angles)
    rX_list.append(r_xx)

    r_yy = np.zeros([2, N])  # matrice des vecteurs reflechis ' x (en colonnes)
    r_yy[0, :] = Y - d*np.sin(r_angles)
    r_yy[1, :] = Y + d*np.sin(r_angles)
    rY_list.append(r_yy)

# dessins

# figure et axes
fig = plt.figure(figsize=(7, 8))
ax = fig.add_subplot(111, aspect='equal')

ax.set_xlim(-2.2, 2.2)
ax.set_ylim(-.5, 6)
title = f"Le nombre de rayons lumineux est {N}"
fig.suptitle(title)

ax.plot(x_parabola, y_parabola)

# initialisation de l'animation
rLines_list = []
for j in range(N):
    x = rX_list[0][:, j]
    y = rY_list[0][:, j]
    rLine = ax.plot(x, y, c='g', alpha=.3, linewidth=.8)[0]
    rLines_list.append(rLine)

# animation
for k in range(1, nbImag):
    for j in range(N):
```

```
        rLine = rLines_list[j]
        rLine.set_xdata(rX_list[k][:, j])
        rLine.set_ydata(rY_list[k][:, j])
    plt.pause(0.01)

plt.show()
```

SOLUTION DE L'EXERCICE 7 DU TP3

```
import numpy as np
import matplotlib.pyplot as plt

# Mettre l'adresse de l'image
Img = plt.imread("...adresse de l'image...")
plt.imshow(Img)

# Une image = une matrice en 3 dimensions (hauteur/largeur/RGB)
Img.shape

# Un pixel isole : ses niveaux RGB
Img[325, 400]

# Decoupage d'une partie de l'image
Img2 = Img[70:271, 10:151, :]
plt.imshow(Img2)

# Quelques passages en negatif (R, G, B et RGB)
ImgNeg = Img.copy()
ImgNeg[:, :, 0] = 255-ImgNeg[:, :, 0]
plt.imshow(ImgNeg)

ImgNeg = Img.copy()
ImgNeg[:, :, 1] = 255-ImgNeg[:, :, 1]
plt.imshow(ImgNeg)

ImgNeg = Img.copy()
ImgNeg[:, :, 2] = 255-ImgNeg[:, :, 2]
plt.imshow(ImgNeg)

ImgNeg = Img.copy()
ImgNeg[:, :, 0] = 255-ImgNeg[:, :, 0]
ImgNeg[:, :, 1] = 255-ImgNeg[:, :, 1]
ImgNeg[:, :, 2] = 255-ImgNeg[:, :, 2]
plt.imshow(ImgNeg)

# Niveaux de R, de G et de B (par annulation ou saturation des autres composantes)
ImgR = Img.copy()
ImgR[:, :, 1:] = 0
#ImgR[:, :, 1:] = 255
plt.imshow(ImgR)

ImgG = Img.copy()
ImgG[:, :, ::2] = 0
#ImgG[:, :, ::2] = 255
plt.imshow(ImgG)

ImgB = Img.copy()
ImgB[:, :, :2] = 0
#ImgB[:, :, :2] = 255
plt.imshow(ImgB)

# Passage en niveaux de gris par une combinaison lineaire specifique
ImgNG = 0.2126*Img[:, :, 0] + 0.7152*Img[:, :, 1] + 0.0722*Img[:, :, 2]
ImgNG.shape
plt.imshow(ImgNG, cmap="Greys_r")
```

SOLUTION DE L'EXERCICE 8 DU TP3

```python
import numpy as np

# Determinant d'une matrice 2x2
def det2d(M):
    if M.shape != (2,2):
        raise ValueError("La matrice n'est pas de dimension 2x2 !")

    return M[0,0]*M[1,1]-M[0,1]*M[1,0]

# test
M = np.array( [ [1, -1], [3, 2] ] )
det2d(M)


# Determinant d'une matrice 3x3
def det3d(M):
    if M.shape != (3,3):
        raise ValueError("La matrice n'est pas de dimension 3x3 !")

    M1 = M[1:, 1:]
    M2 = M[1:, ::2]
    M3 = M[1:, :2]

    return M[0,0]*det2d(M1) - M[0,1]*det2d(M2) + M[0,2]*det2d(M3)

M = np.array( [ [2, 3, -1], [0, 1, 2], [4, -1, -2] ] )
det3d(M)


# Determinant d'une matrice nxn calcule de maniere recursive
def detNd(M):
    if M.shape[0] != M.shape[1]:
        raise ValueError("La matrice n'est pas carree !")

    # A ce stade, on sait que la matrice est carree
    n = M.shape[0]

    # Niveau le plus bas de la recursivite : les matrices 1x1
    if n==1:
        return M[0, 0]

    # Niveau superieur : les matrices sont encore nxn avec n > 1
    # On va developper par rapport a la 1ere colonne (choix arbitraire)
    d = 0
    for i in range(1, n+1):
        # Indices lignes retenus pour extraire la i-eme sous-matrice
        Li = [j for j in range(n)]
        Li.remove(i-1)

        # Extraction de la i-eme sous-matrice
        Mi = M[Li, 1:]

        # Mise a jour du determinant
        d += (-1)**(i-1)*M[i-1, 0]*detNd(Mi)

    return d


# tests
M = np.array( [ [1, -1], [3, 2] ] )
detNd(M)

M = np.array( [ [2, 3, -1], [0, 1, 2], [4, -1, -2] ] )
detNd(M)

# Grande matrice (10x10) generee aleatoirement
n = 10
```

```
M = np.array(np.random.rand(n**2)).reshape(n, n)
detNd(M)
np.linalg.det(M)

# La difference est flagrante entre notre methode recursive (tres mauvais choix ici) et les
    methodes optimisees de numpy
%timeit detNd(M)
%timeit np.linalg.det(M)
```

SOLUTION DE L'EXERCICE 9 DU TP3

```
import numpy as np

# Calcul du spectre de M
M = np.array( [ [1, 2, 3], [0, 5, 6], [1, -1, 3] ] )
Sp = np.linalg.eig(M)
D = np.diag(Sp[0]) # Valeurs propres mises dans une matrice diagonale
P = Sp[1] # Vecteurs propres (matrice de passage)
P1 = np.linalg.inv(P)

# M = PDP^(-1) aux approx numeriques pres
P.dot(D).dot(P1)
# D = P^(-1)MP aux approx numeriques pres
P1.dot(M).dot(P)

# On resoud MX=b
M = np.array( [ [1, -1, 2], [-1, 2, 3], [0, -1, 1] ])
b = np.array( [3, -7, 1] )
X = np.linalg.inv(M).dot(b)
# x=5/2, y=-3/2, z=-1/2
# On peut utiliser la commande solve
X=np.linalg.solve(M,b)
# On peut aussi exploiter la matrice diagonale : MX=b est equivalent a DX'=b' avec X'=P1X et b
    '=P1b (coordonnees respectives de X et b dans  la base des vecteurs propres). On en deduit
    que chaque coordonnee de X' est une coordonnee de b' divisee par une valeur propre et X=PX'
```

SOLUTION DE L'EXERCICE 10 DU TP3

```
import numpy as np

# Calcul du rang d'une matrice
M = np.array([ [1, -1, 2, 1, 2],
               [-1, 2, 3, -4, 1],
               [0, -1, 1, 0, 0] ])
np.linalg.matrix_rank(M)

N = np.array([ [1, 2, 3, 16, 0, 17],
               [-2, -1, -3, -14, 0, -16],
               [-1, -2, -3, -16, 0, -17] ])
np.linalg.matrix_rank(N)
```

SOLUTION DE L'EXERCICE 11 DU TP3

```
import numpy as np

# Calcul de la puissance n-eme d'une fonction par iteration
def puissMat(M, n):
    if M.shape[0] != M.shape[1]:
        raise ValueError("La matrice n'est pas carree !")

    # On recupere la dimension de la matrice (qui est carree a ce stade du programme)
    d = M.shape[0]

```

```
    P = np.eye(d) # M^0 est la matrice identite
    if n > 0:
        for i in range(n):
            P = M.dot(P) # Iterativement : I -> M I -> M^2 I -> ...

    return P

# tests
M = np.array( [ [1, -1, 2], [-1, 2, 3], [0, -1, 1] ])
puissMat(M, 0)
puissMat(M, 1)
puissMat(M, 5)

# Sur des grandes matrices et des grandes puissances, linalg.matrix_power fait mieux: normal, il utilise des techniques de diagonalisation
n = 100
M = np.array(np.random.rand(n**2)).reshape(n, n)
puissMat(M, 100)
np.linalg.matrix_power(M, 100)

%timeit puissMat(M, 100)
%timeit np.linalg.matrix_power(M, 100)


# Cayley-Hamilton dans quelques cas simples (2x2 et 3x3)
M = np.array([ [2, -3], [1, -1] ])
puissMat(M, 2) - puissMat(M, 1) + puissMat(M, 0)

M = np.array([ [1, 2, 3], [1, 2, 0], [0, 1, 3] ])
-puissMat(M, 3) + 6*puissMat(M, 2) - 9*puissMat(M, 1) + 3*puissMat(M, 0)

# Fonctions evaluant le polynome caracteristique d'une matrice A en une matrice M (2x2 et 3x3)
def cayleydim2(A,M):
  if np.shape(A)!=(2,2) or np.shape(M)!=(2,2):
    raise ValueError("Une des matrices n'est pas 2x2")
  else:
    # P=X**2-trace(A)*X+det(A)
    return np.linalg.matrix_power(M,2)-np.trace(A)*M+np.linalg.det(A)*np.eye(2)

# test
Adim2=np.ones((2,2))+2*np.eye(2)
Mdim2=np.ones((2,2))+2*np.eye(2)
cayleydim2(Adim2,Mdim2)

def cayleydim3(A,M):
  if np.shape(A)!=(3,3) or np.shape(M)!=(3,3):
    raise ValueError("Une des matrices n'est pas 3x3")
  else:
    # On sait que les coefficients d'un polynome peuvent etre obtenus comme foctions
    symetriques elementaires de ses racines : dans le cas du polynome caracteristique, les
    racines sont les valeurs propres.
    vp=np.linalg.eig(A)[0]
    # P=-X**3+a2*X**2+a1*X+a0 avec a2=trace(A), a1=-(vp[0]*vp[1]+vp[0]*v[2]+vp[1]*vp[2]) et a0
    =det(A)
    a2=vp[0]+vp[1]+vp[2]
    a1=-vp[0]*vp[1]-vp[0]*vp[2]-vp[1]*vp[2]
    a0=vp[0]*vp[1]*vp[2]
    return -np.linalg.matrix_power(M,3)+a2*np.linalg.matrix_power(M,2)+a1*M+a0*np.eye(3)

# test
Adim3=np.ones((3,3))+2*np.eye(3)
Mdim3=np.ones((3,3))+2*np.eye(3)
cayleydim3(Adim3,Mdim3)
```

SOLUTION DE L'EXERCICE 12 DU TP3

```
# Evalue le polynome caracteristique de M en les points LstL
```

```python
def evalP(M, LstL):
    if M.shape[0] != M.shape[1]:
        raise ValueError("La matrice n'est pas carree !")

    d = M.shape[0]
    I = np.eye(d)
    n = len(LstL)
    P = np.zeros(n)

    # Evaluation du polynome pour chaque valeur de lambda dans LstL
    for i, l in enumerate(LstL):
        P[i] = np.linalg.det(M - l*I)

    return P

# Test sur une matrice dont on connait le polynome caracteristique
M = np.array([ [1, 2, 3], [1, 2, 0], [0, 1, 3] ])
evalP(M, [0, 1, -1])

#n = 3
#M = np.array([ [1, 2, 3], [1, 2, 0], [0, 1, 3] ])

#n = 2
#M = np.array([ [2, -3], [1, -1] ])

n = 10
M = np.array(np.random.rand(n**2)).reshape(n, n)

# On choisit n lambda aleatoirement
LstL = np.random.rand(n)

# On construit C et v
C = np.zeros( (n, n) )
for i in range(n-1, -1, -1):
    C[:, n-1-i] = LstL**i
v = evalP(M, LstL) - (-1)**n*LstL**n

# On en deduit les coefficients
a = np.linalg.inv(C).dot(v)

# Illustration de Cayley-Hamilton
P = np.zeros( (n, n) )
for i in range(n):
    P += a[n-1-i]*puissMat(M, i)
P += (-1)**n*puissMat(M, n)

# Fonction evaluant le polynome caracteristique d'une matrice A en une matrice M et retournant
    cette evalution et le vecteur des coordonnees du polynome caracteristique de la matrice A
    dans la base {1,X,X**2,...,X**d} avec d le nombre de lignes de A
def cayleydimd(A,M):
  # Nombre de lignes de A (d>=1)
  d=np.shape(A)[0]
  if np.shape(A)!=(d,d) or np.shape(M)!=(d,d):
    raise ValueError("Une des matrices n'est pas "+str(d)+"x"+str(d))
  else:
    # Generation aleatoire (suivant une loi uniforme sur un intervalle) des d points d'
    evalution
    xmin=0
    xmax=d
    v=(xmax-xmin)*np.random.rand(d)+xmin
    # Evaluations du polynome caracteristique : P(x)=det(A-xId)
    y=np.array([np.linalg.det(A-v[i]*np.eye(d)) for i in range(d)])-(-1)**d*v**d
    # Creation de la matrice de Vandermond (progressions geometriques sur les lignes de
    raisons v[i] pour i=0,1,...,d-1)
    C=np.ones((d,d))
    for j in range(1,d):
      C[:,j]=v**j # theoriquement, la matrice est inversible vu que la probabilite que deux
    coordonnees de v soient egales est nulle.
    # Resoudre Ca=y equivaut a P(v[i])=y[i]+(-1)**d*v[i]**d pour i=0,1,...,d-1
```

```
    a=np.linalg.solve(C,y) # coefficients du polynome caracteristique : P=(-1)**d*X**d+a[d-1]*X**(d-1)+...+a[1]*X+a[0]
    # Evalution du polynome caracteristique en M
    S=(-1)**d*np.linalg.matrix_power(M,d)
    for i in range(d):
      S+=a[d-1-i]*np.linalg.matrix_power(M,d-1-i)
    # Retourne la matrice P(M) et le vecteur a=(a[0],a[1],...,a[d-1],(-1)**d) des coordonnees de P dans la base {1,X,X**2,...,X**d}
    return S,np.append(a,(-1)**d)

# tests
Adim4=np.ones((4,4))+2*np.eye(4)
Mdim4=np.ones((4,4))+2*np.eye(4)
cayleydimd(Adim4,Mdim4)

Adim5=np.ones((5,5))+2*np.eye(5)
Mdim5=np.ones((5,5))+2*np.eye(5)
cayleydimd(Adim5,Mdim5)

Adim6=np.ones((6,6))+2*np.eye(6)
Mdim6=np.ones((6,6))+2*np.eye(6)
cayleydimd(Adim6,Mdim6)

Adim7=np.ones((7,7))+2*np.eye(7)
Mdim7=np.ones((7,7))+2*np.eye(7)
cayleydimd(Adim7,Mdim7)
```

QUELQUES DÉFINITIONS ET RAPPELS

Dans tout ce qui suit, $\mathbb{K}$ est un corps (penser $\mathbb{Q}$, $\mathbb{R}$ ou $\mathbb{C}$).

Définition 1 (Valeurs propres). Soit $M \in \mathscr{M}_d(\mathbb{K})$ une matrice carrée à coefficients dans $\mathbb{K}$. On dit que $\lambda \in \mathbb{K}$ est une valeur propre pour M si, et seulement si, $M - \lambda \mathbb{I}_d$ n'est pas injective :

$$\mathbf{Ker}(M - \lambda \mathbb{I}_d) \neq \{0_{\mathbb{K}^d}\} \Longleftrightarrow \mathbf{rang}(M - \lambda \mathbb{I}_d) < d \Longleftrightarrow \mathbf{det}(M - \lambda \mathbb{I}_d) = 0.$$

Définition 2 (Vecteurs propres). On appelle vecteur propre un vecteur v non nul tel qu'il existe un scalaire $\lambda \in \mathbb{K}$ pour lequel $Mv = \lambda v$: $\exists \lambda \in \mathbb{K},\ v \in \mathbf{Ker}(M - \lambda \mathbb{I}_d) \neq \{0_{\mathbb{K}^d}\}$. $\mathbf{Ker}(M - \lambda \mathbb{I}_d)$ est appelé sous-espace propre associé à la valeur propre λ.

Définition 3 (Spectre). On appelle spectre sur $\mathbb{K}$ de M l'ensemble des valeurs propres :

$$\mathbf{Sp}_{\mathbb{K}}(M) := \Big\{\lambda \in \mathbb{K}, \quad \mathbf{det}(M - \lambda \mathbb{I}_d) = 0\Big\}.$$

On rappelle que l'application **det** est une forme multilinéaire alternée (sur les lignes et les colonnes d'une matrice) : elle est linéaire sur chaque ligne (et chaque colonne) et nulle si la matrice a deux lignes (ou deux colonnes) colinéaires.

Définition 4 (Polynôme caractéristique). L'application $x \in \mathbb{K} \mapsto \mathbf{det}(M - x\mathbb{I}_d) \in \mathbb{K}$ définit un polynôme de $\mathbb{K}[X]$ de degré d appelé polynôme caractéristique de M qu'on note souvent χ_M.

En particulier, M a au plus d valeurs propres sur $\mathbb{K}$. Si $\mathbb{K}$ est algébriquement clos (penser $\mathbb{C}$), le polynôme caractéristique est scindé :

$$\chi_M = (-1)^d X^d + a_{d-1}X^{d-1} + \ldots + a_1 X + a_0 = (-1)^d \prod_{k=1}^{d} (X - \lambda_k)$$

avec $\lambda_1, \ldots, \lambda_d$ les valeurs propres comptées avec multiplicité. Les coefficients de χ_M sont des fonctions symétriques élémentaires de ses racines :

$$\forall k \in \{1, \ldots, d\}, \quad a_{d-k} = (-1)^{d-k} \sum_{I \subset \mathscr{P}_{k,d}} \prod_{i \in I} \lambda_i$$

avec $\mathscr{P}_{k,d}$ l'ensemble des parties à k éléments de $\{1,2,\dots,d\}$ dont le cardinal est donné par le coefficient binomial $\binom{d}{k} := \frac{d!}{k!(n-k)!}$. Par exemple, $\mathscr{P}_{1,d} = \Big\{\{1\},\{2\},\dots,\{d-1\},\{d\}\Big\}$, $\mathscr{P}_{d,d} = \Big\{\{1,2,\dots,d\}\Big\}$,

$$\sum_{\mathrm{I}\subset\mathscr{P}_{1,d}}\prod_{i\in\mathrm{I}}\lambda_i = \sum_{i=1}^{d}\lambda_i = \mathbf{trace}(\mathrm{M}) \quad \text{et} \quad \sum_{\mathrm{I}\subset\mathscr{P}_{d,d}}\prod_{i\in\mathrm{I}}\lambda_i = \prod_{i=1}^{d}\lambda_i = \mathbf{det}(\mathrm{M}).$$

Notons que le spectre est l'ensemble des valeurs propres comptées sans multiplicité et que pour deux valeurs propres distinctes, les sous-espaces propres sont en somme directe :

$$\bigoplus_{\lambda\in\mathbf{Sp}_{\mathbb{K}}(\mathrm{M})}\mathbf{Ker}(\mathrm{M}-\lambda\mathbb{I}_d).$$

Soit $\mathrm{P} = b_n\mathrm{X}^n + b_{n-1}\mathrm{X}^{n-1} + \ldots + b_1\mathrm{X} + b_0 \in \mathbb{K}[\mathrm{X}]$. On définit l'évaluation sur une matrice carrée A par $\mathrm{P}(\mathrm{A}) := b_n\mathrm{A}^n + b_{n-1}\mathrm{A}^{n-1} + \ldots + b_1\mathrm{A} + b_0\mathbb{I}_d \in \mathbb{K}[\mathrm{A}]$

Définition 5 (Polynôme minimal). Il existe un (unique modulo normalisation du terme de degré dominant) polynôme de $\mathbb{K}[\mathrm{X}]$ de degré minimal qui annule M : $\mathrm{P}(\mathrm{M}) = 0_{d\times d}$. On note souvent $\chi^{\mathrm{M}}_{\min}$ le polynôme normalisé.

De plus, on peut montrer que l'ensemble des polynômes annulant M est exactement l'ensemble des multiples du polynôme minimal (division euclidienne) :

$$\mathbf{nul}(\mathrm{M}) = \Big\{\mathrm{P}\chi^{\mathrm{M}}_{\min}, \quad \mathrm{P}\in\mathbb{K}[\mathrm{X}]\Big\}.$$

Le théorème suivant donne un multiple du polynôme minimal.

Theorème 1 (Cayley-Hamilton)**.** *Tout endomorphisme de* $\mathbb{K}^d$ *annule son polynôme caractéristique :* $\chi_{\mathrm{M}}(\mathrm{M}) = 0_{d,d}$.

De plus, les racines de $\chi^{\mathrm{M}}_{\min}$ sont exactement les valeurs propres de M.

Définition 6 (Matrice diagonalisable). On dit que M est diagonalisable sur $\mathbb{K}$ si, et seulement si, elle est semblable à une matrice diagonale $\mathrm{D}\in\mathscr{M}_d(\mathbb{K})$: il existe une matrice de passage $\mathrm{P}\in\mathrm{GL}_d(\mathbb{K})$ telle que $\mathrm{M} = \mathrm{PDP}^{-1}$.

Theorème 2. *Les assertions suivantes sont équivalentes.*

1. M *est diagonalisable sur* $\mathbb{K}$.
2. *Il existe une base de* $\mathbb{K}^d$ *de vecteurs propres de* M.
3. *Les sous-espaces propres sont supplémentaires :*

$$\mathbb{K}^d = \bigoplus_{\lambda\in\mathbf{Sp}_{\mathbb{K}}(\mathrm{M})}\mathbf{Ker}(\mathrm{M}-\lambda\mathbb{I}_d).$$

4. *Le polynôme caractéristique* χ_{M} *est scindé sur* $\mathbb{K}$ *et la multiplicité géométrique de chaque valeur propre est égale à sa multiplicité algébrique :*

$$\sum_{\lambda\in\mathbf{Sp}_{\mathbb{K}}(\mathrm{M})}\mathbf{dim}(\mathbf{Ker}(\mathrm{M}-\lambda\mathbb{I}_d)) = d.$$

5. *Le polynôme minimal de* M *est scindé sur* $\mathbb{K}$ *à racines simples (lemme des noyaux).*
6. M *a un polynôme annulateur scindé sur* $\mathbb{K}$ *à racines simples.*

À titre d'application, on va diagonaliser $\mathrm{M} = \mathrm{J} + 2\mathbb{I}_d$ avec J la matrice $d\times d$ avec $d\geq 2$ et que des 1. on a $\mathbf{Ker}(\mathrm{J}) = \{v, \quad v_1+v_2+\ldots+v_d = 0\}$, $\mathbf{Im}(\mathrm{J}) = \mathbf{Vect}\{(1,1,\dots,1)^{\mathrm{T}}\}$ et $\mathbf{Ker}(\mathrm{J})\bigoplus\mathbf{Im}(\mathrm{J}) = \mathbb{K}^d$. D'autre part, on a : pour tout $v\in\mathbf{Ker}(\mathrm{J})$, $\mathrm{M}v = 2v$ et pour tout $w\in\mathbf{Im}(\mathrm{J})$, $\mathrm{M}w = (d+2)w$. On en déduit que M est diagonalisable avec P donnée par la matrice de passage dont chaque colonne est un vecteur propre (coordonnées dans la base canonique) et D la matrice diagonale des valeurs propres comptées avec multiplicité et dans l'ordre d'apparition des vecteurs propres associés dans la matrice de passage P : $\mathbf{Sp}_{\mathbb{K}}(\mathrm{M}) = \{2, d+2\}$ avec 2 de multiplicité la dimension du noyau $(d-1)$ et $d+2$ de multiplicité la dimension de l'image (1).
Concernant d'autres exemples, toute matrice de projection ($\mathrm{M}^2-\mathrm{M} = 0_{d,d}$) est diagonalisable. Quant à la symétrie ($\mathrm{M}^2-\mathbb{I}_d = 0$), si $1+1\neq 0$ (penser $\mathbb{Q}$, $\mathbb{R}$ ou $\mathbb{C}$), elle est aussi diagonalisable. Pour aller plus loin, il y a aussi le théorème spectral dont une des conséquences est la diagonalisabilité (sur $\mathbb{R}$) de toute matrice symétrique ($\mathrm{M}^{\mathrm{T}} = \mathrm{M}$) réelle.