

Projet TAN2

Pierre-Antoine Lambrecht

May 2023

1 Diagonalisation

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
import scipy.integrate as spi

A = np.array([[3, 3, 4], [6, -2, -12], [-2, 3, 9]])

vp, vep = np.linalg.eig(A)

print("Valeurs propres de A: ", vp)

vp = vp[idx]
vep = vep[:, idx]

D = np.diag(vp, 0)
P = np.array(vep)
Pinv = np.linalg.inv(P)

A_ = P @ D @ Pinv  # @ : dot product

print("\nP=\n", np.around(P, 2))
print("\nD=\n", np.around(D, 2))
print("\nP-1=\n", np.around(Pinv, 2))
print("\nPAP-1=\n", A_)

def diagonalize(A):
    """
    Diagonalise A en mettant les valeurs propre dans l'ordre croissant.
    Attention la fonction ne verifie pas que A est diagonalisable !

    Parameters:
    _____
```

A : array.
Matrice a diagonalise.

Returns :

(D, P) : couple of arrays.
D est la matrice diagonale avec les vp triee ,ie $d_{1,1} < d_{2,2} < \dots < d_{n,n}$
P est la matrice de passage.
 """

```
vp, vep = np.linalg.eig(A)
idx = vp.argsort()[::-1]
vp = vp[idx]
vep = vep[:, idx]
D = np.diag(vp, 0)
```

Avant toutes choses on importe les librairies nécessaires au projet en leur donnant des alias pour rendre leurs utilisations plus agréables.

On crée la matrice A avec *numpy* ce qui permet de manipuler les matrices plus facilement et peut être aussi plus efficacement qu'avec un simple tableau python de base de la forme $A = []$.

Pour diagonaliser la matrice A il va falloir :

1. vérifier que A est **diagonalisable**
2. calculer les **valeurs propres** de A
3. calculer un **vecteur propre** associé à chaque valeur propre

On utilise donc la commande *eig* de la librairie *numpy.linalg* (pour linear algebra) qui prend comme argument une matrice et renvoie un couple de vecteurs (valeurs propre, vecteur propre). Comme les valeurs propre sont distincts, χ_A le polynome caractéristique de A , est **scindé à racines simples**. Donc A est diagonalisable dans \mathbb{R} .

Pour avoir les valeurs propres triées par ordre croissant on utilise la procédure suivante trouvée sur StackOverflow:

1. $idx = vp.argsort()[::-1]$, calcul les indices de tri pour le vecteur vp et les met dans un vecteur idx .
2. $vp = vp[idx]$, tri le vecteur vp en utilisant les indices idx .
3. $vep = vep[:, idx]$, applique la même transformation au vecteur vep .

On a donc trié les valeurs propres et leurs vecteurs propres respectif.
 On met ensuite les valeurs propres sur la diagonale principale de D et les vecteurs

propres dans une matrice de passage P .

On calcul l'inverse de P , P_{inv} , puis on test que le produit matriciel PDP_{inv} soit bien égal à notre matrice A de départ.

On met tout dans une fonction pour pouvoir l'utiliser dans la dernière question.

Remarque : e_j est un vecteur propre associé à la valeur propre de la j -ème colonne de D . Les vecteurs propres sont libres et forment une base de \mathbb{R}^3 .

On obtient, après exécution du code, les résultats suivant :

Valeurs propres de A : [1. 5. 4.]

$P =$
 $\begin{bmatrix} 0.41 & 0.58 & -0.89 \\ -0.82 & -0.58 & 0. \\ 0.41 & 0.58 & -0.45 \end{bmatrix}$

$D =$
 $\begin{bmatrix} 1. & 0. & 0. \\ 0. & 4. & 0. \\ 0. & 0. & 5. \end{bmatrix}$

$P^{-1} =$
 $\begin{bmatrix} 2.45 & -2.45 & -4.9 \\ -3.46 & 1.73 & 6.93 \\ -2.24 & 0. & 2.24 \end{bmatrix}$

$PAP^{-1} =$
 $\begin{bmatrix} 3. & 3. & 4. \\ 6. & -2. & -12. \\ -2. & 3. & 9. \end{bmatrix}$

2 Recherche de valeur propre

```
def power_iteration(A, max_iter=1000, tol=1e-8):
    """
```

Calcule la plus grande valeur propre de la matrice A ainsi que le vecteur propre associé en utilisant la méthode de la puissance itérée.

Parameters:

A : *numpy.ndarray*

La matrice d'entree de taille (n, n)
max_iter : int, optional
Le nombre maximum d'iterations de la methode de la puissance iteree.
Default is 1000.
tol : float, optional
La tolerance pour la convergence de la methode de la puissance iteree.
Default is 1e-8.

Returns

(lam, v) : tuple of (float, numpy.ndarray)
La plus grande valeur propre de A et le vecteur propre associe.
 """

```
n = A.shape[0]          # extrait la taille de A pour creer le vecteur propre
v = np.random.rand(n)   # on part d'un vecteur propre random
lam = 0.0                # initialisation de lambda la vp recherche
```

```
for i in range(max_iter):          # algorithme de la puissance iteree
    v_new = A @ v                  # v_k+1 = A dot v_k
    lam_new = np.linalg.norm(v_new)
    v_new = v_new / lam_new        # v_k+1 = v_k+1 / norm(v_k+1)

    if np.abs(lam_new - lam) < tol: # critere d'arret de precision
        break
```

```
v = v_new          # mise a jour des valeurs
lam = lam_new
```

```
return (lam, v)
```

```
def inverse_power_iteration(A, max_iter=1000, tol=1e-8):
    """
```

Calcule la plus petite valeur propre de la matrice A
ainsi que le vecteur propre associe en utilisant la methode
de la puissance iteree inverse.

Parameters

A : numpy.ndarray
La matrice d'entree de taille (n, n)
max_iter : int, optional
Le nombre maximum d'iterations de la methode de la puissance iteree inve
Default is 1000.
tol : float, optional
La tolerance pour la convergence de la methode de la puissance iteree in

Default is 1e-8.

Returns

(lam, v) : tuple of (float, numpy.ndarray)
La plus petite valeur propre de A et le vecteur propre associe.
"""

```
n = A.shape[0]          # extrait la taille de A pour creer le vecteur propre
v = np.random.rand(n)   # on par d'un vecteur propre random
lam = 0.0

for i in range(max_iter):
    v_new = np.linalg.solve(A, v)
    lam_new = np.linalg.norm(v_new)
    v_new = v_new / lam_new          # v-k+1 / norme(v-k+1)

    if np.abs(lam_new - lam) < tol:    # critere d'arret de precision
        break

    v = v_new
    lam = lam_new

return (1/lam, v)

# Plus grande valeur propre et vecteur propre associe
lam1, v1 = power_iteration(A)
print("Puissance_iterée:")
print("Plus_grande_valeur_propriée:", np.around(lam1, 4))
print("Vecteur_proprié_associe:", np.around(v1, 4))

# Plus petite valeur propre et vecteur propre associe
lam2, v2 = inverse_power_iteration(A)
print("\nPlus_petite_valeur_propriée:", np.around(lam2, 4))
print("Vecteur_proprié_associe:", np.around(v2, 4))
```

On crée les fonctions *power_iteration* et *inverse_power_iteration* qui utilisent l'algorithme de la puissance itérée vu en cours. Ces fonctions ont d'abord été générées par ChatGPT puis légèrement modifiées et testées.

On part d'un vecteur "aléatoire" et on itère en utilisant l'algorithme de la puissance itérée jusqu'à atteindre la précision voulu ou que le nombre d'itération maximale soit atteint.

On aurait aussi pu utiliser une méthode avec la décomposition QR.

L'exécution du code nous fournit les résultats suivant :

Puissance itérée :

```

Plus grande valeur propre : 5.0
Vecteur propre associé : [0.8944 0. 0.4472]

Plus petite valeur propre : 1.0
Vecteur propre associée : [-0.4082 0.8165 -0.4082]

```

3 Graphe d'une fonction paramétrée

```

def gamma(t):
    return np.exp((2j*np.pi*t)) # j est l'unité imaginaire en python

t = np.linspace(0, 1, 1000) # subdivision de [0;1]
z = gamma(t) # notre nombre complexe évalué sur la sub [0,1]
x, y = np.real(z), np.imag(z)
plt.title("Courbe_parametree_de_Gamma")
plt.xlabel("Re")
plt.ylabel("Im")
plt.plot(x, y)
plt.show()

```

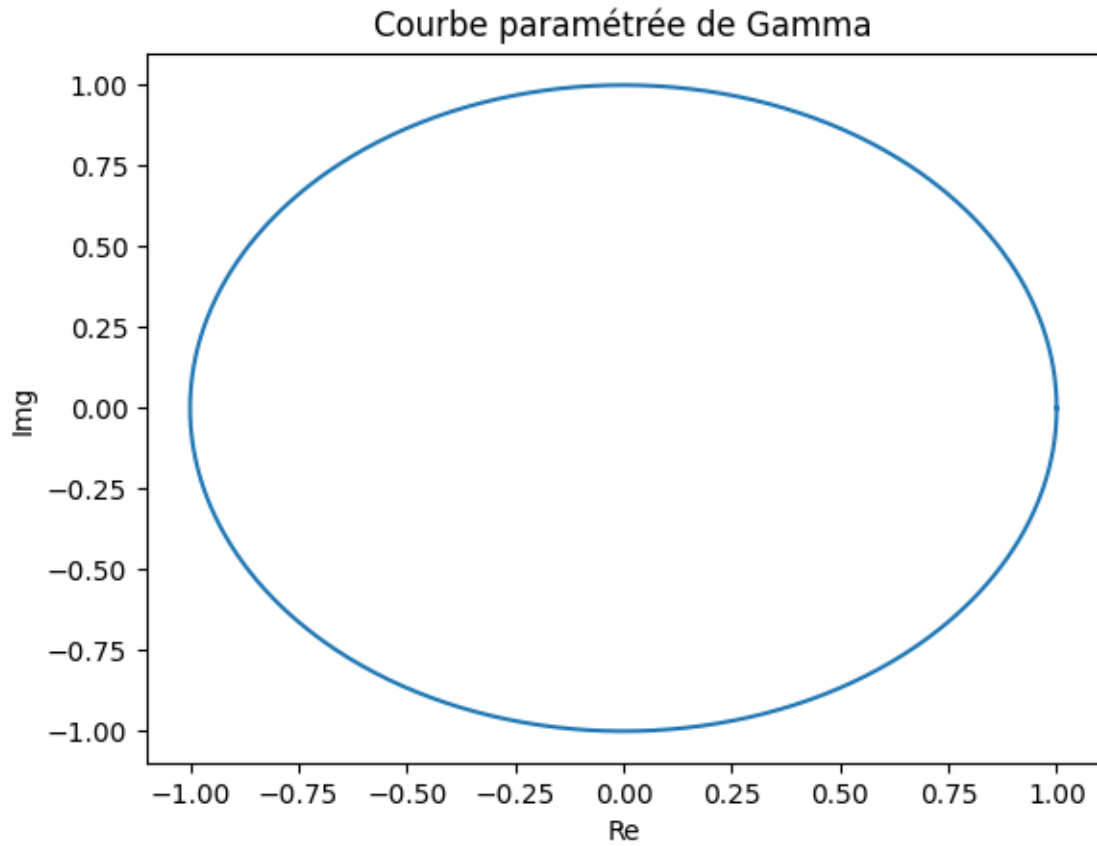
On commence par créer notre fonction *gamma* à valeurs dans \mathbb{C} à l'aide de l'unité complexe *i* codée par *j* en python.

On évalue ensuite *gamma* sur une subdivision régulière de l'intervalle $[0,1]$ créée à l'aide de la commande *linspace*.

On extrait ensuite la partie réelle et la partie imaginaire puis on trace la courbe paramétrée avec en abscisse les parties réelles et en ordonnée les parties imaginaires.

On remarque que *gamma* est un *lacet*.

On obtient ainsi la figure suivante :



4 Intégrale curviligne

```
def dgamma(t):
    """ Derivee de la fonction gamma definie plus haut."""
    return 2j*np.pi * gamma(t)

def integrale_curviligne(f, gamma, dgamma, a=0, b=1):
    """
    L'integrale curviligne de f le long de gamma.

    Parameters:
    _____
    f : fonction complexe.
    gamma : fonction.
           Chemin de classe C1 par morceaux.
```

```

dgamma : fonction.
Derivee de gamma.
a, b : float.
Bornes d'integrations, domaine de gamma.

Returns:


---


(res, err) : couple of (complexe, float).
Valeur de l'integrale de f le long de gamma et l'erreur d'integration.
"""
def integrale_curviligne(f, gamma, dgamma, a, b):
    return f(gamma(a)) * dgamma(a)

# Integration de la partie reelle de l'integrale curviligne
Re, Re_err = spi.quad(lambda t: np.real(integrande(t)), a, b)

# Integration de la partie imaginaire de l'integrale curviligne
Im, Im_err = spi.quad(lambda t: np.imag(integrande(t)), a, b)

# Calcul de l'integrale curviligne complexe et de l'erreur
res = Re + 1j * Im
err = np.sqrt(Re_err**2 + Im_err**2)

return res, err

return (z - z0)**3

z0 = 0.5 * (1 + 1j)
return 1. / (z - z0)

I_1, err1 = integrale_curviligne(f1, gamma, dgamma, 0, 1)
I_2, err2 = integrale_curviligne(f2, gamma, dgamma, 0, 1)
print("Integrales curvilignes complexes: ")
print("Jc(f1): ", I_1, " ", "Erreur d'integration: ", err1)
print("Jc(f2): ", I_2, " ", "Erreur d'integration: ", err2)

```

Soit $f : \mathbb{R} \rightarrow \mathbb{C}$ une fonction à valeurs complexes d'une variable réelle t . Les parties réelles et imaginaires de f sont souvent dénotées $u(t)$ et $v(t)$, respectivement, de sorte que

$$f(t) = u(t) + iv(t). \quad (1)$$

Alors l'intégrale de f sur l'intervalle $[a, b]$ est donnée par

$$\int_a^b f(t) dt = \int_a^b (u(t) + iv(t)) dt = \int_a^b u(t) dt + i \int_a^b v(t) dt. \quad (2)$$

Soit $f : \mathbb{C} \rightarrow \mathbb{C}$ une fonction continue sur une courbe lisse orientée γ . Soit $z : \mathbb{R} \rightarrow \mathbb{C}$ une paramétrisation de γ consistante avec sa direction. Alors l'intégrale

le long de γ est notée

$$\int_{\gamma} f(z) dz \quad (3)$$

et se calcule par

$$\int_{\gamma} f(z) dz = \int_a^b f(\gamma(t)) \gamma'(t) dt. \quad (4)$$

(https://fr.wikipedia.org/wiki/M%C3%A9thodes_de_calcul_d%27int%C3%A9grales_de_contour)

Pour appliquer la formule de l'intégrale curviligne d'une fonction le long d'un chemin on va avoir besoin de la dérivée du chemin. On commence donc par coder la fonction *dgamma*, la dérivée de *gamma*.

On crée notre fonction *integrale_curviligne* en appliquant la formule d'une intégrale curviligne.

Dans cette fonction on crée la fonction *integrande* qui est la composée

$$f(\gamma(t))\gamma'(t)$$

On décompose la fonction *integrande* en parties réelle et imaginaire, et intègre chacune d'entre elles séparément en utilisant la fonction *quad* de la librairie *Scipy*. Finalement, on combine les deux intégrales pour obtenir l'intégrale curviligne complexe et on calcule l'erreur comme la somme des erreurs des intégrales réelles et imaginaires.

On test sur les 2 fonctions données en exemples.

On obtient les résultats suivant :

Integrales curvilignes complexe :

Jc(f1): (-1.7763568394002505e-15-1.9984014443252818e-15j)
 Erreur d'integration: 3.226314840937301e-09

Jc(f2): (-3.3306690738754696e-16+6.283185307179586j)
 Erreur d'integration: 3.836968120193684e-09

5 Intégration numérique

```
def rectangle_gauche(f, a, b, N=1000):
    """
```

Integration numerique par la methode des rectangles a gauches.

Parameters:

f : fonction a valeurs complexe.

```

a, b : floats.
      Bornes d'integrations.
N : int.
      Nombre de pas dans la subdivision.

```

Returns

```

res : complexe.
      La valeur approcher de l'integrale de f sur [a,b].
"""

```

```

res = 0j
h = (b - a) / float(N) # pas de la subdivision

```

```

for i in range(N):
    x = a + i*h # i commence a 0, on prend le point a gauche du rectangle
    res += f(x) * h # hauteur * base

```

```

return res

```

Test

```

def exp(x):
    return np.exp(x)

```

```

print("Test_rectangle_gauche: ", rectangle_gauche(exp, 0, 1, 10000))

```

```

J_1 = rectangle_gauche(lambda t: f1(gamma(t))*dgamma(t), 0, 1, 10000)
J_2 = rectangle_gauche(lambda t: f2(gamma(t))*dgamma(t), 0, 1, 10000)
print("\nMethodes_des_rectangles_a_gauches: ")
print("J(f1)=", J_1)
print("J(f2)=", J_2)

```

On crée une fonction *rectangle_gauche* qui utilise la méthode des rectangles à gauches pour calculer numériquement l'intégrale d'une fonction donnée. On prend bien soins d'initailiser notre résultat avec une valeur complexe. On test d'abord notre fonction avec la fonction exponentielle puis en faisant des intégrales curviligne (on donne les composés $f(\gamma(t))\gamma'(t)$ en argument à *rectangle_gauche*)

Remarque : La méthode est d'ordre > 1 car d'apres la question 3, gamma est une fonction régulière définie sur l'intervalle $[0,1]$ et d'apres le papier cité en source du sujet (<https://irma.math.unistra.fr/~helluy/PREPRINTS/cras1998.pdf>) on peut utiliser la méthode dite de "périodisation".

L'exécution nous fournis les résultats suivant :

```

Test rectangle gauche : (1.7181959157995237+0j)

```

```

Methodes des rectangles a gauches :
J(f1) = (-4.176888869500406e-16+4.6182133638106304e-15j)
J(f2)= (1.9016906105395748e-16+6.283185307179535j)

```

6 Calcul symbolique

```

# Pour le membre de gauche, on calcul l'integrale curviligne

t = sp.Symbol('t')
# Chemin gamma
gamma = 2 * sp.exp(2 * sp.I * sp.pi * t)
# La derivee de gamma
dgamma = 4 * sp.I * sp.pi * sp.exp(2 * sp.I * sp.pi * t)

# Notre matrice A
A_symb = sp.Matrix([[3, 3, 4], [6, -2, -12], [-2, 3, 9]])

B = gamma * sp.eye(3) - A_symb
f = B.inv() * dgamma

# L'integrale curviligne recherche
int_f = (1 / (2*sp.I*sp.pi)) * sp.integrate(f, (t, 0, 1))

print("Integrale symbolique : \n")
sp.pprint(int_f)

# Pour le membre de droite, on doit creer la projection

# La matrice J_1 de l'enoncer
J = sp.Matrix.zeros(3)
J[0, 0] = 1

# Pour obtenir P et P**-1 on diagonalise A avec sympy

# Diagonalisation de la matrice A
# Les vp sont dans l'ordre croissant
P, D = A_symb.diagonalize(sort=True)
# sp.pprint(D)

# Verification :
# sp.pprint(P * D * P**-1)

Pi_1 = P * J * P**-1

```

```
sp.pprint(Pi_1)
```

On utilise la librairie *sympy* pour définir un symbol t puis les fonctions $gamma$, $dgamma$ (la dérivée de $gamma$), la matrice A de la question (1), puis notre intégrande f de l'intégrale curviligne

$$f = (\gamma(t)I_n - A)^{-1}\gamma'(t) \quad (5)$$

enfin on intègre sur l'intervalle $[0,1]$.

Pour la somme des projections, comme γ entoure uniquement la valeur propre λ_1 on a,

$$\sum_{k \in \{1\}} \Pi_k = \Pi_1 = PJ_1P^{-1}$$

avec,

$$J_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Pour obtenir P on diagonalise A avec *sympy* et l'option *sort = True* pour avoir les valeurs propres dans le même ordre qu'à la question 1.

On test en effectuant le produit matriciel PDP^{-1} .

On obtient les résultats quelque peu anormaux suivant :

```
0      0      0
0      0      0
0      0      0
1      -1     -2
-2     2      4
1      -1     -2
```

7 Intégrale de contour numérique

*# on va cree notre matrice integrande puis utiliser la methode des rectangles
a gauche pour integrer chaque entree de la matrice*

```
def contour(gamma, dgamma, A, a, b, N=1000):
    """
```

Calcule la formule de l'integrale de contour d'une matrice.

Parameters:

gamma : function.
Chemin C1 par morceaux
dgamma : function.
D r i v e de gamma.
A : array.
Matrice.
a, b : float.
Borne d'integration.
N : int.
Nombre de pas dans la subdivision.

Returns:

res : array.
Matrice des integrales.
"""

```
n, m = np.shape(A)
res = np.zeros((n, m), dtype=complex)
```

```
def integrande(t):
    return np.linalg.inv(gamma(t)*np.eye(n) - A) * dgamma(t)
```

```
for i in range(n):
    for j in range(m): # c'est une matrice carre de toute facon mais bon
        res[i, j] = rectangle_gauche(
            lambda t: integrande(t)[i, j], a, b, N)
```

```
return res
```

Chemin gamma

```
def gamma2(t):
    return 2 * np.exp(2j * np.pi * t)
```

La derivee de gamma

```
def dgamma2(t):
    return 4j * np.pi * np.exp(2j * np.pi * t)
```

```
contour = contour(gamma2, dgamma2, A, 0, 1, 1000)
print("Contour num rique : \n")
print(contour)
```

```
def projection(A, K):
```

"""
Calcule la formule de la somme sur k dans K des projections PI_k.

Parameters:

A : array.
Matrice.
K : array.
Vecteur contenant les indices.

Returns:

res : array.
Matrice de la somme des projections.

"""

```
def proj(i, A):
    D, P = diagonalize(A)
    n, m = np.shape(P)
    J = np.zeros((n, m))
    J[i-1, i-1] = 1
    proj = P @ J @ np.linalg.inv(P)
    return proj
res = np.zeros(np.shape(A))
for i in range(len(K)):
    res += proj(K[i], A)
return res
```

```
print("\nProjection : \n")
K = [1]
print(projection(A, K))
```

On crée la fonction *contour* qui va calculer numériquement l'intégrale curviligne,

$$\frac{1}{2i\pi} \oint_{\gamma} (zI - A)^{-1} dz$$

Le resultat est une matrice à valeurs complexes, on initialise donc dans notre fonction le résultat *res* avec une matrice nulle qu'on prend soins de spécifier complexe qu'on va ensuite remplir.

On définit une fonction intégrande qui est la composée (5).

On intègre ensuite chaque entrée de la matrice intégrande en utilisant notre fonction *rectangle_gauche* définie à la question 5 qui prend bien des fonctions à valeurs complexes et on place le résultat dans *res*.

Pour la somme des projections,

$$\sum_{k \in K} \Pi_k = \sum_{k \in K} P J_k P^{-1}$$

on crée une fonction *projection* qui prend comme arguments une matrice et un vecteur d'indices K .

Dans la fonction on commence par définir une projection individuel Π_i en utilisant la définition d'une projection et la fonction *diagonalize* définie à la question 1 pour récupérer P .

Il ne reste plus qu'à faire la somme des projections Π_i pour $i \in K$ et retourner le résultat.

On obtient les résultats suivant :

Contour numerique :

```
[ [ 4.62564692e-17 +6.28318531j  4.65456113e-17 -6.28318531j
   -2.61672194e-16-12.56637061j ]
 [ 6.01225704e-17-12.56637061j  4.31078784e-16+12.56637061j
   -6.69752523e-17+25.13274123j ]
 [ 6.72205347e-17 +6.28318531j -4.51517148e-19 -6.28318531j
   -2.01011083e-16-12.56637061j ] ]
```

Projection :

```
[ [ 1. -1. -2.]
  [-2.  2.  4.]
  [ 1. -1. -2.]]
```

8 Annexe

- **Person:** - Écrire un programme Python utilisant numpy qui permet de retrouver la plus grande et la plus petite valeur propre d'une matrice A au moyen de la méthode la puissance itérée, ainsi que des vecteurs propres associés

- **ChatGPT:** - Voici un exemple de programme Python qui utilise numpy pour calculer la plus grande et la plus petite valeur propre d'une matrice A en utilisant la méthode de la puissance itérée :

```
import numpy as np
```

```
def power_iteration(A, max_iter=1000, tol=1e-8):
    """
```

*Calcule la plus grande valeur propre de la matrice A
ainsi que le vecteur propre associé en utilisant la méthode
de la puissance itérée.*

Parameters

*A : numpy.ndarray
La matrice d'entrée de taille (n, n)*

max_iter : int, optional
Le nombre maximum d'itérations de la méthode de la puissance itérative.
Default is 1000.

tol : float, optional
La tolérance pour la convergence de la méthode de la puissance itérative.
Default is 1e-8.

Returns

(lam, v) : tuple of (float, numpy.ndarray)
La plus grande valeur propre de A et le vecteur propre associé.
 """

```
n = A.shape[0]
v = np.random.rand(n)
lam = 0.0

for i in range(max_iter):
    v_new = A @ v
    lam_new = np.linalg.norm(v_new)
    v_new = v_new / lam_new

    if np.abs(lam_new - lam) < tol:
        break

    v = v_new
    lam = lam_new

return (lam, v)
```

```
def inverse_power_iteration(A, max_iter=1000, tol=1e-8):
    """
```

Calcule la plus petite valeur propre de la matrice A
ainsi que le vecteur propre associé en utilisant la méthode
de la puissance itérative inverse.

Parameters

A : numpy.ndarray
La matrice d'entrée de taille (n, n)

max_iter : int, optional
Le nombre maximum d'itérations de la méthode de la puissance itérative.
Default is 1000.

tol : float, optional
La tolérance pour la convergence de la méthode de la puissance itérative.
Default is 1e-8.

Returns

(lam, v) : tuple of (float, numpy.ndarray)
La plus petite valeur propre de A et le vecteur propre associé.
"""

```
n = A.shape[0]
v = np.random.rand(n)
lam = 0.0

for i in range(max_iter):
    v_new = np.linalg.solve(A, v)
    lam_new = np.linalg.norm(v_new)
    v_new = v_new / lam_new

    if np.abs(lam_new - lam) < tol:
        break

    v = v_new
    lam = lam_new

return (1/lam, v)
```

```
# Exemple d'utilisation
A = np.array([[4, 2], [2, 3]])

# Plus grande valeur propre et vecteur propre associé
lam1, v1 = power_iteration(A)
print("Plus_grande_valeur_propre:", lam1)
print("Vecteur_propre_associe:", v1)

# Plus petite valeur propre et vecteur propre associé
lam2, v2 = inverse_power_iteration(A)
print("Plus_petite_valeur_propre:", lam2)
print("Vecteur_propre_associe:", v2)
'''
```

Dans cet exemple, la matrice d'entrée 'A' est définie comme une matrice '2x

https://fr.wikipedia.org/wiki/M%C3%A9thodes_de_calcul_d%27int%C3%A9grales_de_contour

- **Person:** corrige cette fonction :

```
def integrale_curviligne(f, gamma, dgamma, a=0, b=1):
    """
    L'intégrale curviligne de f le long de gamma.
```

Parameters:

f : fonction.
gamma : fonction.
Chemin de classe C1 par morceaux.
Dgamma : fonction.
D r i v e de gamma.
a, b : float.
Borne d'integrations , domaine de gamma.

Returns:

(res , err) : couple of (complexe , float).
Valeur de l'int grale de f le long de gamma et l'erreur d'integration.
"""
Re = np.real(f(gamma(t)) * dgamma(t))
Im = np.imag(f(gamma(t)) * dgamma(t))

res1 , e1 = sc.integrate.quad(**lambda** t: Re, a, b)
res2 , e2 = sc.integrate.quad(**lambda** t: Im, a, b)
return res1 + 1j*res2 , e1+e2

- **ChatGpt:** Il y a plusieurs erreurs dans cette fonction. Voici une version corrigée :

```
import numpy as np
import scipy.integrate as spi
```

```
def integrale_curviligne(f, gamma, dgamma, a=0, b=1):
    """
```

L'int grale curviligne de f le long de gamma.

Parameters:

f : fonction.
gamma : fonction.
Chemin de classe C1 par morceaux.
dgamma : fonction.
D r i v e de gamma.
a, b : float.
Borne d'int grations , domaine de gamma.

Returns:

(res , err) : couple of (complexe , float).
Valeur de l'int grale de f le long de gamma et l'erreur d'integration.

```

"""
def integrande(t):
    return f(gamma(t)) * dgamma(t)

# Int gration de la partie r elle de l'int grale curviligne
Re, Re_err = spi.quad(lambda t: np.real(integrande(t)), a, b)

# Int gration de la partie imaginaire de l'int grale curviligne
Im, Im_err = spi.quad(lambda t: np.imag(integrande(t)), a, b)

# Calcul de l'int grale curviligne complexe et de l'erreur
res = Re + 1j * Im
err = np.sqrt(Re_err**2 + Im_err**2)

return res, err

```

Voici les corrections que j'ai apportées :

- J'ai ajouté l'importation de la bibliothèque 'scipy.integrate' sous le nom 'spi', pour éviter de devoir taper 'scipy.integrate' à chaque fois.
- J'ai corrigé l'orthographe de 'dgamma' dans la liste des paramètres.
- J'ai défini une fonction 'integrande' qui calcule l'intégrande 'f(gamma(t)) * dgamma(t)'.
- J'ai utilisé la fonction 'quad' de la bibliothèque 'scipy.integrate' pour calculer l'intégrale curviligne de la partie réelle et de la partie imaginaire de 'integrande'.
- J'ai calculé l'intégrale curviligne complexe en combinant les résultats de l'intégration de la partie réelle et de la partie imaginaire, et j'ai calculé l'erreur en combinant les erreurs de l'intégration de la partie réelle et de la partie imaginaire.

- **Person:** diagonalize $\begin{bmatrix} 3 & 3 & 4 \\ 6 & -2 & -12 \\ -2 & 3 & 9 \end{bmatrix}$ using sympy

- **ChatGPT:** Voici comment diagonaliser la matrice $\begin{bmatrix} 3 & 3 & 4 \\ 6 & -2 & -12 \\ -2 & 3 & 9 \end{bmatrix}$ en utilisant SymPy :

```

import sympy as sp

# D finir la matrice A
A = sp.Matrix([[3, 3, 4], [6, -2, -12], [-2, 3, 9]])

# Obtenir les vecteurs propres et les valeurs propres de A
eigenvectors = A.eigenvecs()

# Diagonaliser la matrice A
D, P = A.diagonalize()

# Afficher les vecteurs propres et les valeurs propres
print("Vecteurs propres et valeurs propres :")
for eigenvalue, multiplicity, eigenvects in eigenvectors:
    print("Valeur propre :", eigenvalue)

```

```

    for eigenvect in eigenvects:
        print("Vecteur propre :", eigenvect)
    print()

# Afficher la matrice diagonalis e de A
print("Matrice diagonalis e de A :")
print(P**-1 * A * P)

    La sortie sera :
    Vecteurs propres et valeurs propres :
    Valeur propre : -4
    Vecteur propre : Matrix([[ -1], [ 2], [ 1]])

    Valeur propre : 6
    Vecteur propre : Matrix([[ 2], [ 0], [ 1]])

    Valeur propre : 8
    Vecteur propre : Matrix([[ 1], [ -1], [ 1]])

    Matrice diagonalis e de A :
    Matrix([ [ -4, 0, 0], [ 0, 6, 0], [ 0, 0, 8]])

http://galusins.univ-tln.fr/ENSEIGN/M43-sympy.html

https://docs.sympy.org/latest/modules/matrices/matrices.html

```