Calcul matriciel avancé

I. Présentation du TD

L'objet de ce TD est double :

- rappeler les principales fonctions de la bibliothèque `numpy` ; conformément au programme, les fonctions que je vais présenter ici doivent vous être rappelées au concours ;
- dans un second temps, nous utiliserons ces procédures pour résoudre certains problèmes vus dans le chapitre sur la réduction des endomorphismes.

II. Introduction à Numpy

- Il faut d'abord importer `numpy`.

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

On évitera d'écrire : `from numpy import *` car les fonctions importées surchargent des fonctions existantes.

- **Création basique d'un tableau**

Cela se fait au moyen de la fonction `np.array`.

```
>>> v = np.array([1, 2, 3, 4]); v
array([1, 2, 3, 4])
```

```
>>> A = np.array([ [1, 2, 3, ], [4, 5, 6] ])
>>> B = np.array([[i+j for j in range(3)] for i in range(4)])
>>> A; B
array([[1, 2, 3],
       [4, 5, 6]])
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

Ces objets sont de type `ndarray`. Un tableau `numpy` unidimensionnel n'est pas une liste !

```
>>> type(v), type([1, 2, 3, 4])
(<class 'numpy.ndarray'>, <class 'list'>)
```

- **Taille d'un array**

La fonction `np.size()` ou la propriété `.size` renvoient le nombre d'éléments d'un tableau.

```
>>> np.size(v), np.size(A), B.size
(4, 6, 12)
```

La fonction `np.ndim()` ou la propriété `.ndim` renvoie la dimension (nombre d'indices nécessaires), la fonction `np.shape()` ou la propriété `.shape` renvoie le type de la matrice.

```
>>> np.shape(v), A.shape, B.ndim
((4,), (2, 3), 2)
```

- **Typage**

Attention : le type de TOUS les éléments d'un tableau est le même et est fixé implicitement lors de sa création. On peut le consulter à l'aide de la propriété `.dtype`.

```
>>> A.dtype
dtype('int32')
>>> A[0,0] = "Bonjour"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Bonjour'
```

Cependant, on peut définir le type de manière explicite lors de la création du tableau.

```
>>> M = np.array([[1, 2], [3, 4]], dtype=complex); M
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Types possibles : `int`, `float`, `complex`, `bool` etc.... et on peut aussi préciser la taille en bits : `int8`, `int64`, `float128`, `complex128` etc...

- **Création avancée de tableaux**
  - **arange**

```
>>> x = np.arange(0, 10, 1) # arguments: start, stop, step. stop n'est pas inclus
>>> y = np.arange(0, 1.5, 0.2)
>>> x; y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4])
```

  - **linspace**

```
>>> np.linspace(0, 10, 10)
array([  0.        ,   1.11111111,   2.22222222,   3.33333333,
         4.44444444,   5.55555556,   6.66666667,   7.77777778,
         8.88888889,  10.        ])
>>> # avec linspace, le début et la fin sont inclus
>>> #ici, 10 est le nombre total de points, ce qui fait 9 intervalles
```

  - **shape et reshape**

```
>>> A = np.arange(15).reshape(3,5); A
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> B = np.arange(5,21)
>>> B.shape = (4,4)
>>> B
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20]])
```

  - **Transformation en listes**
    On peut transformer un tableau en liste (ou liste de listes) et vice-versa.

```
>>> L = B.tolist() # tolist est une méthode applicable aux objets de type array
>>> L
[[5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16], [17, 18, 19, 20]]
>>> np.array(L)
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20]])
```

  - **diag**
    Création d'une matrice diagonale généralisée.

```
>>> # matrice diagonale classique
>>> np.diag([1, 2, 3, 4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
>>> # diagonale avec décalage par rapport à la diagonale principale
>>> np.diag([1, 2, 3], k=1) # k peut prendre aussi des valeurs <0
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

. **identity et eye**

```
>>> # matrice identité
>>> np.identity(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> # matrice identité + décalage des 1
>>> np.eye(4, k=-1, dtype=int)
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0]])
```

. **zeros, ones, empty et fill**

```
>>> np.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((3,4), dtype = 'int')
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
>>> A = np.empty([2,3], dtype = int)
>>> A.fill(2)
>>> A
array([[2, 2, 2],
       [2, 2, 2]])
```

- **Manipulation d'arrays**

. **Indexation**

Pour accéder aux éléments d'un tableau, on indique le ou les indices de cet élément entre crochets. Comme d'habitude en Python, les indices commencent à 0.

```
>>> v[0]
1
>>> A[1,2]
2
>>> A[3,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 2
```

Pour un tableau à 2 dimensions, on peut aussi obtenir directement lignes ou colonnes.

```
>>> B = np.arange(12).reshape(3,4); B
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> B[1] # 2ème ligne
array([4, 5, 6, 7])
>>> B[0,:] # 1ère ligne
array([0, 1, 2, 3])
>>> B[:,2] # 3ème colonne
array([ 2,  6, 10])
```

On peut évidemment affecter des valeurs à des éléments particuliers, mais aussi à des lignes ou colonnes.

```
>>> B[1,:] = -1
>>> B[:,2] = -2
>>> B
array([[ 0,  1, -2,  3],
       [-1, -1, -2, -1],
       [ 8,  9, -2, 11]])
```

. **Slicing et extraction**
Comme pour les listes, le slicing consiste à extraire une partie d'un tableau ; la syntaxe est `M[start1:stop1:step1, start2:stop2,step2]` . Voici quelques exemples.

```
>>> A = np.array([[n+m*10 for n in range(6)] for m in range(5)]); A
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45]])
>>> # un bloc
>>> A[2:5, 0:4]
array([[20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> # une matrice extraite
>>> A[::2, 1::2]
array([[ 1,  3,  5],
       [21, 23, 25],
       [41, 43, 45]])
>>> # à l'envers
>>> A[:, ::-1]
array([[ 5,  4,  3,  2,  1,  0],
       [15, 14, 13, 12, 11, 10],
       [25, 24, 23, 22, 21, 20],
       [35, 34, 33, 32, 31, 30],
       [45, 44, 43, 42, 41, 40]])
>>> A[::-1, ::-1]
array([[45, 44, 43, 42, 41, 40],
       [35, 34, 33, 32, 31, 30],
       [25, 24, 23, 22, 21, 20],
       [15, 14, 13, 12, 11, 10],
       [ 5,  4,  3,  2,  1,  0]])
>>> # on peut modifier un bloc
>>> A[1:4,3:5]=0;A
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12,  0,  0, 15],
       [20, 21, 22,  0,  0, 25],
       [30, 31, 32,  0,  0, 35],
       [40, 41, 42, 43, 44, 45]])
```

On peut utiliser des listes ou des array pour définir des tranches (*fancy indexing*).

```
>>> indices_lignes = [1, 4, 2]
>>> A[indices_lignes]
array([[10, 11, 12,  0,  0, 15],
       [40, 41, 42, 43, 44, 45],
       [20, 21, 22,  0,  0, 25]])
>>> indices_colonnes = [1, 2, -1] # rappel : l'indice -1 fait référence au dernier élément
>>> A[:, indices_colonnes]
array([[ 1,  2,  5],
       [11, 12, 15],
       [21, 22, 25],
       [31, 32, 35],
       [41, 42, 45]])
>>> A[indices_lignes][:,indices_colonnes]
array([[11, 12, 15],
       [41, 42, 45],
       [21, 22, 25]])
>>> A[indices_lignes, indices_colonnes] # attention!
array([11, 42, 25])
>>> # on peut aussi extraire une diagonale
>>> np.diag(A)
array([ 0, 11, 22,  0, 44])
>>> np.diag(A, -2)
array([20, 31, 42])
>>> np.diag(A, 1)
array([ 1, 12,  0,  0, 45])
```

- **Transposition**

```
>>> A = np.array([ [1, 2, 3], [4, 5, 6] ])
>>> A.transpose()
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> A # A n'est pas modifiée
array([[1, 2, 3],
       [4, 5, 6]])
>>> A.T # c'est pareil!
array([[1, 4],
       [2, 5],
       [3, 6]])
```

- **Suppression de rangées**

```
>>> B = np.delete(A, 1, axis=0); B # suppression de la ligne n°1
array([[1, 2, 3]])
>>> C = np.delete(A, 2, axis=1); C # suppression de la colonne n°2
array([[1, 2],
       [4, 5]])
```

- **Insertion de rangées**

```
>>> B = np.insert(A, 0, [-2, -1, 0], axis = 0); B # insertion d'une ligne avant celle de n°0
array([[-2, -1,  0],
       [ 1,  2,  3],
       [ 4,  5,  6]])
>>> C = np.insert(A, 1, [2018, 2018], axis=1); C # insertion d'une colonne avant la colonne n°1
array([[   1, 2018,    2,    3],
       [   4, 2018,    5,    6]])
```

- **Concaténation**

```
>>> A = np.array(range(8)).reshape(2,4); A
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> B = np.zeros(A.shape, int); B
array([[0, 0, 0, 0],
       [0, 0, 0, 0]])
>>> C = np.ones(A.shape, int); C
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
>>> np.concatenate((A, B), axis=1)
array([[0, 1, 2, 3, 0, 0, 0, 0],
       [4, 5, 6, 7, 0, 0, 0, 0]])
>>> np.concatenate((A, B, C), axis=1)
array([[0, 1, 2, 3, 0, 0, 0, 0, 1, 1, 1, 1],
       [4, 5, 6, 7, 0, 0, 0, 0, 1, 1, 1, 1]])
>>> np.concatenate((A, C), axis=0)
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

- **Le problème de l'affectation**

Comme les listes, les tableaux sont des objets *mutables*, c'est-à-dire qu'on peut les modifier en place sans avoir besoin de créer une copie, contrairement aux objets *non mutables*, comme ceux de type `int`, `float`, `bool`, `str` etc...
La taille, le type et l'adresse d'un tableau sont définis une fois pour toutes à l'initialisation. Ainsi, on applique un principe d'économie qui consiste à effectuer aussi peu que possible de recopies de tableaux en mémoire, sauf demande explicite.
L'exemple suivant est très important, parce que la situation évoquée ici revient souvent. Le vecteur a contenant un tableau `numpy`, la variable `A` pointe en fait sur la position de ce tableau en mémoire. L'instruction `B = A` ne crée pas une nouvelle copie du tableau a, mais elle demande à la variable `B` de pointer sur le même tableau physique que `A`. On exprime cette situation en disant que `A` et `B` constituent une même vue d'un tableau.
La conséquence immédiate est que toute modification apportée à cette vue, que ce soit par l'intermédiaire de la variable `A` ou de la variable `B`, affecte simultanément les deux variables.

```
>>> A = np.array([[i**j for j in range(3)] for i in range(4)]); A
array([[1, 0, 0],
       [1, 1, 1],
       [1, 2, 4],
       [1, 3, 9]])
>>> B = A
>>> B[:, 1]=0
>>> A
array([[1, 0, 0],
       [1, 0, 1],
       [1, 0, 4],
       [1, 0, 9]])
```

Et le même phénomène se produit pour les matrices extraites.

```
>>> A
array([[1, 0, 0],
       [1, 0, 1],
       [1, 0, 4],
       [1, 0, 9]])
>>> B = A[0:2,1:3]
>>> B[0,0] = 2018
>>> A
array([[   1, 2018,    0],
```

```
       [   1,    0,    1],
       [   1,    0,    4],
       [   1,    0,    9]])
```

Pour éviter cela, on peut utiliser la méthode `copy()`.

```
>>> A
array([[   1, 2018,    0],
       [   1,    0,    1],
       [   1,    0,    4],
       [   1,    0,    9]])
>>> B = A[0:2,1:3].copy()
>>> B[0,0] = 2019
>>> A
array([[   1, 2018,    0],
       [   1,    0,    1],
       [   1,    0,    4],
       [   1,    0,    9]])
```

III. Utilisation de Numpy en algèbre linéaire

La performance des programmes écrit en Python/Numpy dépend de la capacité à vectoriser les calculs (les écrire comme des opérations sur des vecteurs/matrices) en évitant au maximum les boucles `for` et `while`.

- **Opérations scalaires**
  On peut effectuer les opérations arithmétiques habituelles pour multiplier, additionner, soustraire et diviser des arrays avec/par des scalaires.

```
>>> v = np.arange(6); v
array([0, 1, 2, 3, 4, 5])
>>> w = np.arange(7,13); w
array([ 7,  8,  9, 10, 11, 12])
>>> 2*v+3*w
array([21, 26, 31, 36, 41, 46])
>>> A = np.arange(16).reshape(4,4); A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> A/2 ; A+2
array([[ 0. ,  0.5,  1. ,  1.5],
       [ 2. ,  2.5,  3. ,  3.5],
       [ 4. ,  4.5,  5. ,  5.5],
       [ 6. ,  6.5,  7. ,  7.5]])
array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13],
       [14, 15, 16, 17]])
```

Les opérations par défaut sont des opérations terme-à-terme.

```
>>> v * w # multiplication terme-à-terme
array([ 0,  8, 18, 30, 44, 60])
>>> A * A # multiplication terme-à-terme
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121],
       [144, 169, 196, 225]])
```

`np.trace( )` calcule la trace d'une matrice, avec un argument optionnel supplémentaire pour représenter l'offset (permet de sélectionner une sur-diagonale ou une sous-diagonale).

```
>>> M = np.arange(9).reshape(3,3); M
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.trace(M)
12
>>> np.trace(M, 1)
6
>>> np.trace(M, -1)
10
```

- **Fonctions mathématiques**
  Toutes les fonctions mathématiques usuelles sont présentes sous forme universelle dans le module `numpy`. Elles gardent le même nom, mais il faut bien penser à les préfixer par `np`. Les opérations se font là aussi terme à terme. Donnons juste quelques exemples.

```
>>> x = 10**np.arange(-10,-15,-1.); x
array([  1.00000000e-10,   1.00000000e-11,   1.00000000e-12,
         1.00000000e-13,   1.00000000e-14])
>>> np.log(1+x)/x
array([ 1.00000008,  1.00000008,  1.0000889 ,  0.99920072,  0.99920072])
>>> A = np.arange(1,7).reshape(2,3); A
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.sqrt(A)
array([[ 1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974]])
>>> np.exp(A)
array([[   2.71828183,    7.3890561 ,   20.08553692],
       [  54.59815003,  148.4131591 ,  403.42879349]])
```

- **Produit matriciel**
  C'est la commande `np.dot`.
  `np.dot(A,B)` fait le produit des matrices A et B (de tailles compatibles bien sûr !).
  Si `v` est un vecteur (array uni-dimensionnel) et `A` une matrice (array bi-dimensionnel) alors `np.dot(A,v)` renvoie le produit Av tandis que `np.dot(v,A)` renvoie v^tA.
  Un vecteur `v` est considéré comme une ligne à gauche, et une colonne à droite. Ainsi, si `v` et `w` sont deux vecteurs, `np.dot(v,w)` renvoie leur produit scalaire usuel. Pour obtenir la matrice carrée V^tW, on utilise `np.outer(v,w)`.

```
>>> a = np.arange(1,5); a
array([1, 2, 3, 4])
>>> b = np.arange(11,15); b
array([11, 12, 13, 14])
>>> c = np.arange(21,26); c
array([21, 22, 23, 24, 25])
>>> m = np.arange(20).reshape(4,5); m
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> np.dot(a,b) # produit scalaire
130
>>> np.dot(a,m) # vecteur-ligne par matrice
array([100, 110, 120, 130, 140])
>>> np.dot(m,c) # matrice par vecteur-colonne
array([ 240,  815, 1390, 1965])
>>> np.dot(m,m.T) # calcule m * transp(m)
array([[  30,   80,  130,  180],
       [  80,  255,  430,  605],
       [ 130,  430,  730, 1030],
```

```
       [ 180,  605, 1030, 1455]])
>>> np.outer(a,b)
array([[11, 12, 13, 14],
       [22, 24, 26, 28],
       [33, 36, 39, 42],
       [44, 48, 52, 56]])
```

- **Le module linalg**
  - **Rang d'une matrice**

```
>>> A = np.arange(15).reshape(3,5); A
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> np.linalg.matrix_rank(A)
2
```

  - **Inverse d'une matrice**

```
>>> A = np.array([2,4,6,8],float).reshape(2,2)
>>> np.linalg.inv(A)
array([[-1.  ,  0.5 ],
       [ 0.75, -0.25]])
```

Bien sûr, si la matrice n'est pas inversible vous aurez un message d'erreur `Singular matrix` dans le meilleur des cas, ou bien des résultats aberrants.

  - **Résolution d'un sytème**

```
>>> A = np.array([2,4,6,8],float).reshape(2,2); A
array([[ 2.,  4.],
       [ 6.,  8.]])
>>> B =np.array([1, 4],float); B
array([ 1.,  4.])
>>> np.linalg.solve(A,B)
array([ 1.  , -0.25])
>>> B = np.array([[1, 1],[4, -4]],float); B
array([[ 1.,  1.],
       [ 4., -4.]])
>>> np.linalg.solve(A,B)
array([[ 1.  , -3.  ],
       [-0.25,  1.75]])
```

  - **Déterminant**
    `np.linalg.det(A)`, tout simplement !
  - **Puissances itérées**
    `np.linalg.power(A,n)` calcule A^n. n peut être négatif, masi il faut que A soit inversible !
  - **Éléments propres**
    `np.linalg.eigenvals(A)` renvoie la liste (sous forme d'un array unidimensionnel) des valeurs propres de A.
    `np.linalg.eig(A)` renvoie en plus les vecteurs propres, sous la forme d'un couple (valeurs propres, matrice de passage).

```
>>> A = np.array([  [ 1, 1, -1, 2, -1],
...                 [ 2, 0, 1, -4, -1],
...                 [ 0, 1, 1, 1, 1],
...                 [ 0, 1, 2, 0, 1],
...                 [ 0, 0, -3, 3, -1]])
>>> valp = np.linalg.eigvals(A); valp
array([ 1.00001556 +0.00000000e+00j,  0.99999222 +1.34784311e-05j,
        0.99999222 -1.34784311e-05j, -1.00000000 +0.00000000e+00j,
       -1.00000000 +0.00000000e+00j])
```

```
>>> valp = np.round(valp.real,4); valp # ici on a arrondi les résultats
array([ 1.,  1.,  1., -1., -1.])
>>> A = np.array([ [ 5, 4, 8, -4],
...                [-2, -1, 6, 0],
...                [-2, -1, 0, 1],
...                [ 2, 4, 20, -5]])
>>> np.linalg.eig(A)
(array([-3.,  2.,  1., -1.]), array([[ -3.01511345e-01,  -1.06787647e-16,   1.79605302e-01,
         1.81298661e-16],
       [ -3.01511345e-01,   4.36435780e-01,   3.59210604e-01,
        -7.07106781e-01],
       [  1.96681718e-16,   2.18217890e-01,   1.79605302e-01,
         3.62969143e-16],
       [ -9.04534034e-01,   8.72871561e-01,   8.98026510e-01,
        -7.07106781e-01]]))
```

- **Calculs de normes**
  `np.linalg.norm(x [, t])` renvoie la norme d'un vecteur ou d'une matrice. Le deuxième argument t optionnel désigne le type de norme : 'fro' pour la $\|.\|_2$ (norme de Frobenius), `np.inf` pour la $\|.\|_\infty$ et `1` pour la $\|.\|_1$.

IV. Calcul du polynôme caractéristique d'une matrice par la méthode de Le Verrier

IV.1. La méthode originale (Le Verrier, 1840)

Soit $A \in \mathcal{M}_n(\mathbb{C})$, et $(\lambda_1, \dots, \lambda_n)$ ses valeurs propres dans $\mathbb{C}$ (distinctes ou non). On sait que le polynôme caractéristique de A s'écrit :

$$\chi_A = X^n - \sigma_1 X^{n-1} + \sigma_2 X^{n-2} - \cdots + (-1)^{n-1}\sigma_{n-1}X + (-1)^n\sigma_n,$$

où les σ_i sont les fonctions symétriques élémentaires des racines.

De plus, en trigonalisant A, on montre facilement que : $\forall k \in \mathbb{N},\ \operatorname{tr}(A^k) = \sum_{i=1}^{n} \lambda_i^k$. Notons alors $s_k = \sum_{i=1}^{n} \lambda_i^k$.

On a :

- $s_1 = \sigma_1$
- $s_2 = \sum_{i=1}^{n} \lambda_i^2 = \left(\sum_{i=1}^{n} \lambda_i\right)^2 - 2\sum_{i<j} \lambda_i\lambda_j = \sigma_1^2 - 2\sigma_2$.
- Cela se complique ensuite, mais l'on peut montrer les formules suivantes (*de Newton*) :

$$\forall k \leqslant n,\ s_k - \sigma_1 s_{k-1} + \cdots + (-1)^{k-1}\sigma_{k-1}s_1 + (-1)^k k\sigma_k = 0.$$

Ces formules permettent donc de calculer par récurrence les σ_i connaissant les s_k, donc de calculer les coefficients du polynôme caractéristique de A connaissant les $\operatorname{tr}(A^k)$.

1er programme

1. Quelle est la complexité de cet algorithme pour calculer χ_A ?

2. Écrire le programme Python correspondant, en utilisant `numpy`.

✎ *Solution:*

Le nombre d'opérations élémentaires pour calculer un produit matriciel à l'aide des formules vues en cours :

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j}b_{j,k}$$

est de $n^2(2n-1)$ (n^2 termes, et à chaque fois n multiplications et $n-1$ additions).
Dans l'algorithme de Le Verrier, il faut calculer $A^2, A^3, \dots, A^{n-1}$ puis seulement les termes diagonaux de A^n, ce qui va donner

$$(n-2)n^2(2n-1) + n(2n-1)$$

opérations arithmétiques.
Ensuite il faut calculer les traces de n matrices, soit $n(n-1)$ additions.
Enfin, la résolution de la récurrence exige $2\sum_{k=1}^{n}(k-1) = n(n-1)$ opérations.

En conclusion, la complexité de l'algorithme est en $O(n^4)$.

Corrigé :

```
import numpy as np

def chi(A):
    n = A.shape[0]
    Ak = A.copy() # termes de la suite
    s = [] # liste des sk
    P = [1] # coeff dominant
    for k in range(1,n+1):
        s.append(np.trace(Ak))
        Ak = np.dot(Ak, A)
        somme = 0
        for i in range(len(P)):
            somme -= P[i]*s[-1-i]
        P.append(somme/k)
    return P

for n in range (2,10):
    J = np.ones(n)
    I = np.identity(n)
    A = I + J
    print(chi(A))
```

```
[1, -4.0, 3.0]
[1, -6.0, 9.0, -4.0]
[1, -8.0, 18.0, -16.0, 5.0]
[1, -10.0, 30.0, -40.0, 25.0, -6.0]
[1, -12.0, 45.0, -80.0, 75.0, -36.0, 7.0]
[1, -14.0, 63.0, -140.0, 175.0, -126.0, 49.0, -8.0]
[1, -16.0, 84.0, -224.0, 350.0, -336.0, 196.0, -64.0, 9.0]
[1, -18.0, 108.0, -336.0, 630.0, -756.0, 588.0, -288.0, 81.0, -10.0]
```

IV.2. Une amélioration : l'algorithme de Faddeev-Souriau (1948)

Théorème

Soit $A \in \mathcal{M}_n(\mathbb{K})$; on pose $A_1 = A$ puis : $\forall\, k \in [\![2\,;n]\!],\ A_k = A\left(A_{k-1} - \dfrac{1}{k-1}\operatorname{tr}(A_{k-1})I_n\right)$.

Alors :

$$\chi_A(X) = X^n - \sum_{k=1}^{n} \frac{1}{k}\operatorname{tr}(A_k)X^{n-k}\,.$$

☞ *Démonstration:*

On démontre par récurrence sur $k \in [\![1\,;n]\!]$ que

$$A_k = A^k - \sigma_1 A^{k-1} + \cdots + (-1)^{k-1}\sigma_{k-1}A \text{ et } \operatorname{tr}(A_k) = (-1)^k k\sigma_k$$

en utilisant les formules de Newton données ci-dessus.

2ème programme

1. Quelle est la complexité de cet algorithme pour calculer χ_A ?
2. Écrire le programme Python correspondant, en utilisant `numpy`. Comparer avec le programme précédent.

Corrigé :

```python
import numpy as np

def chi(A):
    n = A.shape[0]
    Ak = A.copy() # puissances de A
    I = np.identity(n)
    P = [1] # coeff dominant
    for k in range(1,n+1):
        c = -np.trace(Ak)/k
        P.append(c)
        Ak = np.dot(A, Ak + c*I )
    return P

for n in range (2,10):
    J = np.ones(n)
    I = np.identity(n)
    A = I + J
    print(chi(A))
    # il y a déjà la fonction dans numpy:
    # print(np.poly(A))
```

```
[1, -4.0, 3.0]
[1, -6.0, 9.0, -4.0]
[1, -8.0, 18.0, -16.0, 5.0]
[1, -10.0, 30.0, -40.0, 25.0, -6.0]
[1, -12.0, 45.0, -80.0, 75.0, -36.0, 7.0]
[1, -14.0, 63.0, -140.0, 175.0, -126.0, 49.0, -8.0]
[1, -16.0, 84.0, -224.0, 350.0, -336.0, 196.0, -64.0, 9.0]
[1, -18.0, 108.0, -336.0, 630.0, -756.0, 588.0, -288.0, 81.0, -10.0]
```

V. Recherche des éléments propres

Il est souvent nécessaire de savoir calculer numériquement les valeurs et les vecteurs propres dans des situations concrètes impliquant des grands systèmes où les méthodes théoriques sont pratiquement inutilisables.
La stratégie générale pour déterminer les valeurs propres et les vecteurs propres consiste à construire une suite de transformations en des matrices semblables jusqu'à l'obtention d'une matrice diagonale, ou plus simplement jusqu'à l'obtention d'une matrice triangulaire, à partir de laquelle il est possible de déterminer assez facilement les vecteurs propres.
Pour réaliser ces transformations, deux grandes classes de méthodes sont disponibles : les méthodes directes et les méthodes itératives.
Les premières consistent à effectuer une suite de transformations conservant la similitude ; elles ne s'appliquent que sur des matrices de taille modeste, car le temps de calcul croît comme le cube de la taille de la matrice.
Pour les matrices de grande taille et généralement creuses, les méthodes itératives sont plus adaptées.
De plus, il existe des méthodes spécifiques pour certaines catégories de matrices (les matrices symétriques, tridiagonales etc...).

V.1. Méthode de la puissance itérée

Cette méthode donne une très bonne approximation de la plus grande valeur propre en module.

Théorème

Soit $A \in \mathcal{M}_n(\mathbb{R})$, dont les valeurs propres complexes λ_i sont telles que $|\lambda_1| > |\lambda_2| \geqslant \cdots \geqslant |\lambda_n|$. On suppose aussi A diagonalisable dans $\mathcal{M}_n(\mathbb{C})$.
On munit $\mathcal{M}_{n,1}(\mathbb{R})$ de la norme $\|\cdot\|_\infty$.

On construit alors une suite $X^{(k)} = \begin{pmatrix} x_1^{(k)} \\ \vdots \\ x_n^{(k)} \end{pmatrix}$ de vecteurs de $\mathcal{M}_{n,1}(\mathbb{R})$ de la façon suivante :

$$X^{(0)} \text{ est donné} \quad \text{et} \quad \forall\, k \in \mathbb{N},\ X^{(k+1)} = \frac{AX^{(k)}}{M_k},$$

où M_k est l'élément de $AX^{(k)}$ le plus grand en valeur absolue ($M_k = \pm \left\|AX^{(k)}\right\|_\infty$) de sorte que la plus grande coordonnée de $X^{(k)}$ est toujours égale à 1.
Si $X^{(0)}$ n'est pas orthogonal au sous-espace propre de ${}^tA$ pour la valeur propre λ_1 alors :

- La suite $\left(X^{(k)}\right)_k$ converge vers un vecteur propre associé à λ_1 lorsque $k \to +\infty$;
- La suite (M_k) converge vers λ_1.

☞ *Démonstration:*

• On notera U_i des vecteurs propres associés aux λ_i, les U_i formant une base de $\mathbb{C}^n$ puisque A est supposée diagonalisable.
Déjà on peut noter que λ_1 est un réel, car sinon $\overline{\lambda_1}$ serait une autre valeur propre de A mais elle est de même module, c'est exclu.
Un vecteur propre U_1 associé à λ_1 sera donc à coefficients réels, et quitte à le diviser par $\pm\|U_1\|_\infty$, on peut supposer que sa plus grande coordonnée est égale à 1.
• Soit V_1 un vecteur propre de ${}^tA$ associé à λ_1 : ${}^tAV_1 = \lambda_1 V_1$, et $X^{(0)}$ non orthogonal à V_1.
On écrit : $X^{(0)} = \sum_{i=1}^{n} \alpha_i U_i$. Montrons que $\alpha_1 \neq 0$. En effet, on a :

$$\left\langle X^{(0)} \,\middle|\, V_1 \right\rangle = \sum_{i=1}^{n} \alpha_i \langle U_i \,|\, V_1\rangle = \alpha_1 \langle U_1 \,|\, V_1\rangle \neq 0$$

En effet, si $i \geqslant 2$,

$$\lambda_i \langle U_i \,|\, V_1\rangle = \langle AU_i \,|\, V_1\rangle = \left\langle U_i, {}^tAV_1 \,\middle|\, = \right\rangle \lambda_1 \langle U_i \,|\, V_1\rangle$$

d'où $\langle U_1 \,|\, V_1\rangle = 0$ puisque $\lambda_i \neq \lambda_1$.
• On a, en itérant :

$$X^{(k)} = \frac{AX^{(k-1)}}{M_{k-1}} = \frac{1}{M_{k-1}} \frac{1}{M_{k-2}} A^2 X^{(k-2)} = \cdots = \frac{A^k X^{(0)}}{C_k}$$

où $C_k = \prod_{i=0}^{k-1} M_i$.
D'autre part :

$$A^k X^{(0)} = \sum_{i=1}^{n} \alpha_i A^k U_i = \sum_{i=1}^{n} \alpha_i \lambda_i^k U_i = \lambda_1^k \left(\alpha_1 U_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k U_i \right) = \lambda_1^k Y^{(k)}$$

en ayant posé $Y^{(k)} = \alpha_1 U_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k U_i$.

Ainsi $X^{(k)} = \dfrac{\lambda_1^k Y^{(k)}}{C_k}$.

Or compte tenu des hypothèses sur les λ_i on a $\lim\limits_{k\to+\infty} Y^{(k)} = \alpha_1 U_1$ donc, puisque la plus grande coordonnée de $X^{(k)}$ et celle de U_1 sont égales à 1, on a $\lim\limits_{n\to+\infty} \dfrac{\lambda_1^k}{C_k} = \dfrac{1}{\alpha_1}$. On en tire $\lim\limits_{k\to+\infty} X^{(k)} = U_1$, puis $M_k = \dfrac{C_{k+1}}{C_k} \underset{k\to+\infty}{\longrightarrow} \lambda_1$.

Remarque : le résultat reste valable si λ_1 est une valeur propre multiple de module strictement supérieur aux modules de toutes les autres (adapter la démonstration ci-dessus pour décomposer $X^{(0)} = \alpha_1 U_1 + \sum_{i=p+1}^{n} \alpha_i U_i$, où U_1 est la composante sur le sous-espace propre propre de dimension p associé à λ_1), mais ne l'est plus lorsque $\lambda_1 \neq \lambda_2$ et $|\lambda_1| = |\lambda_2|$.

3ème programme

Écrire un programme qui détermine par cette méthode la plus grande valeur propre d'une matrice (en valeur absolue) ainsi qu'un vecteur propre associé.

Corrigé :

```
import numpy as np
import random

def PuissanceIteree(A, eps):

    def norme(v):
        return max(abs(v))

    def indicemax(v):
        norme_v = norme(v)
        n = len(v)
        for i in range(n):
            if abs(v[i]) == norme_v:
                return i

    nmax = 100000 # nombre max d'itérations, si plus grand, méthode diverge
    n = A.shape[0]
    X = np.array([random.random() for _ in range(n)])
    erreur = eps + 1
    niter = 0
    j = indicemax(X)
    X = X/X[j]
    while erreur > eps and niter < nmax:
        Y = np.dot(A, X)
        j = indicemax(Y)
        mu = Y[j] # tend vers valeur propre
        if abs(mu) < eps:
            return 0, X
        Y = Y/mu
        erreur = norme(X - Y)
        X = Y
        niter += 1
    if niter == nmax:
        return None
    return mu, Y

if __name__ == "__main__":
    eps = 1e-16
    for n in range (2,15):
        J = np.ones(n)
        I = np.identity(n)
        A = I + J
        print(PuissanceIteree(A, eps))
```

```
(3.0, array([ 1.,  1.]))
(4.0, array([ 1.,  1.,  1.]))
(5.0, array([ 1.,  1.,  1.,  1.]))
(6.0, array([ 1.,  1.,  1.,  1.,  1.]))
(7.0, array([ 1.,  1.,  1.,  1.,  1.,  1.]))
(8.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(9.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(10.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(11.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(12.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(13.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(14.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
(15.0, array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]))
```

V.2. Méthode de déflation

Prop:

Soit A une matrice vérifiant les hypothèses du théorème précédent, X_1 un vecteur propre associé à λ_1 et Y_1 un vecteur propre de ${}^tA$ associé à la même valeur propre λ_1.
Alors la matrice

$$B = A - \frac{\lambda_1}{{}^tY_1X_1} X_1 {}^tY_1$$

(appelée *déflatée de* A) a pour valeurs propres $0, \lambda_2, \ldots, \lambda_n$.

☞ *Démonstration:*

Il est en effet facile de vérifier que $BX_1 = 0$.
Ensuite, soit $i \geqslant 2$. On a

$${}^tY_1AX_i = \lambda_i {}^tY_1X_i$$

mais aussi puisque ${}^tY_1AX_i$ est un réel :

$${}^tY_1AX_i = {}^t({}^tY_1AX_i) = {}^tX_i{}^tAY_1 = \lambda_1 {}^tX_iY_1\,,$$

donc $\lambda_i {}^tY_1X_i = \lambda_1 {}^tX_iY_1 = \lambda_1 {}^tY_1X_i$, et puisque $\lambda_i \neq \lambda_1$ on en déduit ${}^tY_1X_i = 0$ d'où $BX_i = AX_i = \lambda_iX_i$, et λ_i est valeur propre de B.

4ème programme

Écrire un programme qui détermine par cette méthode toutes les valeurs propres d'une matrice ainsi que des vecteurs propres associés.

Corrigé :

```python
import numpy as np

from PuissanceIteree import PuissanceIteree

def deflation(A, eps):
    n = A.shape[0]
    liste_valp = []
    liste_vecp = []
    for i in range(n):
        res = PuissanceIteree(A, eps)
        if res == None:
            return('Méthode inapplicable')
        else:
            mu, X = res
        res = PuissanceIteree(A.transpose(), eps)
        if res == None:
            return('Méthode inapplicable')
        else:
```

```
            nu, Y = res
        # mu = nu !
        liste_valp.append(mu)
        liste_vecp.append(X)
        A = A - mu * np.outer(X, Y)/ np.dot(X, Y)
    return liste_valp, liste_vecp

eps = 1e-12

def arrondi(v):
    return([int(1000*x)/1000 for x in v])

for n in range (2,15):
    #A = -2*np.identity(n) + np.eye(n, k=-1) + np.eye(n, k=1)
    A = np.identity(n) + np.ones((n,n))
    #on n'affiche que les valeurs propres pas les vecteurs
    print(arrondi(deflation(A, eps)[0]))
```

```
[2.999, 1.0]
[3.999, 1.0, 1.0]
[4.999, 0.999, 1.0, 1.0]
[5.999, 1.0, 1.0, 1.0, 1.0]
[6.999, 1.0, 1.0, 1.0, 0.999, 1.0]
[7.999, 1.0, 0.999, 1.0, 1.0, 0.999, 0.999]
[8.999, 1.0, 1.0, 0.999, 1.0, 0.999, 1.0, 1.0]
[9.999, 0.999, 1.0, 0.999, 0.999, 0.999, 0.999, 1.0, 1.0]
[10.999, 0.999, 0.999, 1.0, 0.999, 1.0, 1.0, 1.0, 1.0, 1.0]
[11.999, 1.0, 1.0, 1.0, 0.999, 0.999, 1.0, 1.0, 0.999, 1.0, 0.999]
[12.999, 0.999, 0.999, 0.999, 0.999, 1.0, 0.999, 1.0, 0.999, 1.0, 1.0, 1.0]
[13.999, 0.999, 1.0, 0.999, 1.0, 0.999, 1.0, 1.0, 1.0, 1.0, 0.999, 0.999, 1.0]
[14.999, 1.0, 0.999, 1.0, 1.0, 0.999, 1.0, 0.999, 1.0, 0.999, 0.999, 1.0, 0.999, 0.999]
```

V.3. Méthode de Jacobi

Cette méthode concerne la réduction des matrices symétriques réelles.
Si A est une matrice symétrique réelle, on construit une suite de matrices $A^{(k)}$ toutes orthogonalement semblables à la matrice de départ (c'est-à-dire de la forme $P^{-1}AP$ avec $P \in \mathcal{O}_n(\mathbb{R})$, $P^{-1} = {}^tP$), qui converge vers une matrice diagonale D contenant les valeurs propres de A.
Pour cela, on utilise des *matrices de Givens* $G(i_0, j_0, \theta)$ qui sont des matrices de la forme ci-après.
Une telle matrice est une matrice de rotation d'angle $-\theta$ dans le plan $\text{Vect}(e_{i_0}, e_{j_0})$, elle est donc orthogonale.

$$G(i_0,j_0,\theta)=\begin{pmatrix}
1 & & & & & & & & & & \\
& 1 & & & & & 0 & & & & \\
& & \ddots & & & & & & & & \\
& & & 1 & & & & & & & \\
& & & & \cos\theta & 0 & \dots & 0 & \sin\theta & 0 & \dots & 0 & \to i_0\text{ème ligne} \\
& & & & 0 & 1 & & \vdots & 0 & & & & \\
& & & & & 0 & \ddots & 0 & & & & & \\
& & & & 0 & \vdots & & 1 & 0 & & & & \\
& & & & -\sin\theta & 0 & \dots & 0 & \cos\theta & 0 & \dots & 0 & \to j_0\text{ème ligne} \\
& & 0 & & 0 & & & & 0 & 1 & & & \\
& & & & \vdots & & & & \vdots & & \ddots & & \\
& & & & & & & & & & & 1 & \\
& & & & 0 & & & & 0 & & & 1 & \\
& & & & \downarrow & & & & \downarrow & & & &
\end{pmatrix}$$

$i_0$ème colonne — $j_0$ème colonne

Lemme:

Soit $A\in\mathcal{S}_n(\mathbb{R})$, et (i_0,j_0) un couple d'indices distincts. Soit $\theta=\frac{1}{2}\operatorname{Arc\,tan}\left(\frac{2a_{i_0j_0}}{a_{j_0j_0}-a_{i_0i_0}}\right)$ ($\theta=\frac{\pi}{2}$ si $a_{i_0i_0}=a_{j_0j_0}$), et soit $P=G(i_0,j_0,\theta)$.
Alors la matrice $B={}^tPAP$ (orthogonalement semblable à A) vérifie :

$$b_{ii}=a_{ii}\ si\ i\neq i_0\ et\ i\neq j_0\ ,\ b_{i_0j_0}=b_{j_0i_0}=0\ ,\ b_{i_0i_0}^2+b_{j_0j_0}^2=a_{i_0i_0}^2+a_{j_0j_0}^2+2a_{i_0j_0}^2\ ,$$
$$b_{i_0i_0}-a_{i_0i_0}=-\tan(\theta)a_{i_0j_0}\quad,\quad b_{j_0j_0}-a_{j_0j_0}=\tan(\theta)a_{i_0j_0}.$$

☞ *Démonstration:*

Puisque P est la matrice d'une rotation dans le plan $\operatorname{Vect}(e_{i_0},e_{j_0})$, il est clair que seuls les 4 éléments situés aux intersections des lignes/colonnes de numéros i_0 et j_0 sont modifiés (on peut aussi considérer un produit par blocs). Et l'on a :

$$\begin{pmatrix}b_{i_0i_0}&b_{i_0j_0}\\b_{j_0i_0}&b_{j_0j_0}\end{pmatrix}={}^t\begin{pmatrix}\cos\theta&\sin\theta\\-\sin\theta&\cos\theta\end{pmatrix}\times\begin{pmatrix}a_{i_0i_0}&a_{i_0j_0}\\a_{j_0i_0}&a_{j_0j_0}\end{pmatrix}\times\begin{pmatrix}\cos\theta&\sin\theta\\-\sin\theta&\cos\theta\end{pmatrix}$$

avec $a_{i_0j_0}=a_{j_0i_0}$. Il « suffit » alors de calculer, en utilisant des formules de trigonométrie bien connues : $\cos\operatorname{Arc\,tan}\alpha=\frac{1}{\sqrt{1+\alpha^2}}$ et $\sin\operatorname{Arc\,tan}\alpha=\frac{\alpha}{\sqrt{1+\alpha^2}}$. Par exemple :

$$\begin{aligned}b_{j_0i_0}&=(a_{i_0i_0}-a_{j_0j_0})\sin\theta\cos\theta+a_{i_0j_0}(\cos^2\theta-\sin^2\theta)\\&=\frac{1}{2}(a_{i_0i_0}-a_{j_0j_0})\sin(2\theta)+a_{i_0j_0}\cos(2\theta)\\&=\left(\frac{1}{2}(a_{i_0i_0}-a_{j_0j_0})\left(\frac{2a_{i_0j_0}}{a_{j_0j_0}-a_{i_0i_0}}\right)+a_{i_0j_0}\right)\times\begin{pmatrix}\dots&\dots\end{pmatrix}=0.\end{aligned}$$

La dernière égalité provient juste, elle, de la conservation de la norme (au carré).

Théorème : Méthode de Jacobi

Soit $A \in \mathcal{S}_n(\mathbb{R})$. On construit par récurrence une suite de matrices (B_k) de la façon suivante :

$$B_0 = A \quad et \quad \forall k \in \mathbb{N}^*,\ B_k = {}^tP_{k-1}B_{k-1}P_{k-1}$$

où P_{k-1} est la matrice du lemme précédent avec i_0 et j_0 distincts tels que, en notant $B_k = (b_{ij}^{(k)})$:

$$\left|b_{i_0j_0}^{(k-1)}\right| = \max_{\ell \neq m} \left|b_{\ell m}^{(k-1)}\right| .$$

Alors la suite (B_k) converge vers une matrice diagonale D dont les éléments diagonaux sont les valeurs propres de A lorsque $k \to +\infty$.

De plus, dans le cas où les valeurs propres de A sont distinctes, si l'on note $Q_k = P_0P_1 \dots P_k$ alors :

- la suite (Q_k) a une limite $\Omega \in \mathcal{O}_n(\mathbb{R})$ lorsque $k \to +\infty$;
- $D = {}^t\Omega D \Omega$;
- les colonnes de Ω sont des vecteurs propres de A.

☞ *Démonstration:*

On note $B_k = (b_{ij}^{(k)})_{1\leqslant i,j\leqslant n}$. D'après les résultats du lemme on a :

$$b_{i_0j_0}^{(k)} = b_{j_0i_0}^{(k)} = 0 \quad \text{et si } \ell \neq i_0,\ \ell \neq j_0, \quad b_{\ell\ell}^{(k)} = b_{\ell\ell}^{(k-1)}.$$

• Montrons d'abord que :

$$\sum_{i\neq j} (b_{ij}^{(k)})^2 \leqslant \left(1 - \frac{2}{n(n-1)}\right)^k \sum_{i\neq j} a_{ij}^2 \qquad (*)$$

On sait que $\operatorname{tr}({}^tB_kB_k) = \sum_{i,j} (b_{ij}^{(k)})^2$ (cf. produit scalaire canonique dans $\mathcal{M}_n(\mathbb{R})$). De plus,

$$\operatorname{tr}({}^tB_kB_k) = \operatorname{tr}({}^tP_{k-1}{}^tB_{k-1}P_{k-1}{}^tP_{k-1}B_{k-1}P_{k-1}) = \operatorname{tr}({}^tB_{k-1}B_{k-1})$$

puisque la matrice P_{k-1} est orthogonale. On aura donc :

$$\begin{aligned}
\sum_{i\neq j} (b_{ij}^{(k)})^2 &= \operatorname{tr}({}^tB_kB_k) - \sum_{\ell=1}^{n} (b_{\ell\ell}^{(k)})^2 = \operatorname{tr}({}^tB_{k-1}B_{k-1}) - \sum_{\ell=1}^{n} (b_{\ell\ell}^{(k)})^2 \\
&= \operatorname{tr}({}^tB_{k-1}B_{k-1}) - (b_{i_0i_0}^{(k)})^2 - (b_{j_0j_0}^{(k)})^2 - \sum_{\substack{\ell\neq i_0\\ \ell\neq j_0}} (b_{\ell\ell}^{(k)})^2 \\
&= \operatorname{tr}({}^tB_{k-1}B_{k-1}) - \sum_{\ell=1}^{n} (b_{\ell\ell}^{(k-1)})^2 - 2(b_{i_0j_0}^{(k-1}))^2 \qquad \text{(cf. lemme)} \\
&= \sum_{i\neq j} (b_{ij}^{(k-1)})^2 - 2(b_{i_0j_0}^{(k-1}))^2.
\end{aligned}$$

Or d'après le choix de (i_0, j_0), $(b_{i_0j_0}^{(k-1}))^2 \geqslant \dfrac{1}{n(n-1)} \sum_{i\neq j} (b_{ij}^{(k-1)})^2$ (il y a $n^2 - n$ termes dans la somme) donc finalement :

$$\sum_{i\neq j} (b_{ij}^{(k)})^2 \leqslant \left(1 - \frac{2}{n(n-1)}\right) \sum_{i\neq j} (b_{ij}^{(k-1)})^2,$$

et cela démontre le résultat voulu $(*)$ par récurrence sur k (puisque $B_0 = A$.

• Puisque $\left(1 - \dfrac{2}{n(n-1)}\right) < 1$, il résulte de cette inégalité que tous les termes non diagonaux de B_k tendent vers 0 lorsque $k \to +\infty$.

• Il reste à examiner ce qui se passe pour les termes diagonaux.

D'après les résultats du lemme on a pur $\ell \neq i_0$ et $\ell \neq j_0$, $b_{\ell\ell}^{(k)} = b_{\ell\ell}^{(k-1)}$.

Et pour $\ell = i_0$, $b_{i_0i_0}^{(k)} - b_{i_0i_0}^{(k-1)} = -\tan(\theta_k) b_{i_0j_0}^{(k-1)}$, d'où l'on tire, puisque $|\theta_k| < \frac{\pi}{4}$: $\left|b_{i_0i_0}^{(k)} - b_{i_0i_0}^{(k-1)}\right| \leqslant \left|b_{i_0j_0}^{(k-1)}\right|$. On démontre de même que $\left|b_{j_0j_0}^{(k)} - b_{j_0j_0}^{(k-1)}\right| \leqslant \left|b_{i_0j_0}^{(k-1)}\right|$.

L'inégalité $\left|b_{\ell\ell}^{(k)} - b_{\ell\ell}^{(k-1)}\right| \leqslant \left|b_{i_0j_0}^{(k-1)}\right|$ est donc vraie pour tout $\ell \in [\![1\,;n]\!]$. Maintenant, en utilisant la relation $(*)$:

$$\left|b_{\ell\ell}^{(k)} - b_{\ell\ell}^{(k-1)}\right| \leqslant \left|b_{i_0j_0}^{(k-1)}\right| \leqslant \sqrt{\sum_{i\neq j} (b_{ij}^{(k-1)})^2} \leqslant \left(1 - \frac{2}{n(n-1)}\right)^{\frac{k-1}{2}} \sqrt{\sum_{i\neq j} a_{ij}^2},$$

ce qui montre, par comparaison à une série géométrique, que la série $\sum_k \left|b_{\ell\ell}^{(k)} - b_{\ell\ell}^{(k-1)}\right|$ est convergente.

La série $\sum_k \left(b_{\ell\ell}^{(k)} - b_{\ell\ell}^{(k-1)}\right)$ étant absolument convergente, elle est convergente, ce qui signifie que la suite $(b_{\ell\ell}^{(k)})$ converge lorsque $k \to +\infty$.
Cela étant valable pour tout ℓ, on conclut que la suite (B_k) converge vers une matrice diagonale D.
Puisque à chaque étape on a des matrices semblables, on aura :

$$\forall\, k \in \mathbb{N},\ \chi_A(X) = \det(XI_n - A) = \chi_{B_k}(X) = \det(XI_n - B_k)$$

et le déterminant étant une fonction continue des coefficients, on en déduit par passage à la limite :

$$\chi_A(X) = \det(XI_n - D) = \chi_D(X)\,,$$

ce qui implique que les valeurs propres de D, c'est-à-dire ses éléments diagonaux, sont les valeurs propres de A.

• Supposons maintenant les valeurs propres de A distinctes, donc les coefficients diagonaux de D distincts.
Puisque ces coefficients diagonaux sont les limites des $b_{\ell\ell}^{(k)}$, il existe un réel $\eta > 0$ tel que, pour k assez grand, l'on ait $\left|b_{ii}^{(k)} - b_{jj}^{(k)}\right| > \eta$ pour tous couples (i,j) d'indices distincts.
Notons θ_k l'angle qui apparaît dans chaque matrice P_k, de sorte que :

$$\theta_k = \frac{1}{2}\operatorname{Arc\,tan}\left(\frac{2b_{i_0j_0}^{(k)}}{a_{j_0j_0}^{(k)} - a_{i_0i_0}^{(k)}}\right)$$

En vertu de l'inégalité $|\operatorname{Arc\,tan} x| \leqslant |x|$ on aura, à partir d'un certain rang :

$$|\theta_k| \leqslant \frac{\left|b_{i_0j_0}^{(k)}\right|}{\eta} \leqslant \frac{1}{\eta}\leqslant \left(1 - \frac{2}{n(n-1)}\right)^{\frac{k}{2}} \sqrt{\sum_{i\neq j} a_{ij}^2} \quad (**)$$

en utilisant encore la relation $(*)$.
Notons que les matrices $Q_k = P_0P_1\dots P_k$ sont orthogonales, comme produit de telles matrices.
Munissons $\mathcal{M}_n(\mathbb{R})$ de la norme associé au produit scalaire canonique : $\|M\|^2 = \operatorname{tr}({}^tMM)$. Pour cette norme, si Ω est une matrice orthogonale on a :

$$\|\Omega M\|^2 = \operatorname{tr}({}^tM{}^t\Omega\Omega M) = \operatorname{tr}({}^tMM) = \|M\|^2$$

donc

$$\begin{aligned}\|Q_{k+1} - Q_k\|^2 = \|Q_kP_k - Q_k\|^2 &= \|Q_k(P_k - I_n)\|^2\\ &= \|P_k - I_n\|^2 = 2(\cos\theta_k - 1)^2 + 2\sin^2\theta_k = 8\sin^2\left(\frac{\theta_k}{2}\right) \leqslant 2\,|\theta_k|^2\,,\end{aligned}$$

et l'inégalité $(**)$ permet de dire que la série $\sum_k \|Q_{k+1} - Q_k\|$ converge.

La série $\sum_k (Q_{k+1} - Q_k)$ est donc absolument coinvergente dans un espace vectoriel de dimension finie, donc elle est convergente, ce qui implique que la suite (P_k) converge, vers une certaine matrice Ω.

• La relation ${}^tQ_kQ_k$ donne par passage à la limite (justifié par la continuité de l'applicatiojn bilinéaire en dimension finie $(A,B) \mapsto {}^tAB$), ${}^t\Omega\Omega = I_n$, donc Ω est une matrice orthogonale.
De même, la relation $B_k = {}^tQ_{k-1}AQ_{k-1}$ donne par passage à la limite (justifié de la même façon que ci-dessus), $D = {}^t\Omega A\Omega$, ce qui montre bien que Ω est la matrice de passage de la base initiale à une base orthonormale de vecteurs propres.

Mise en oeuvre

Dans la pratique on ne cherche pas à calculer le nombre θ_k pour déterminer les coefficients des lignes et des colonnes i_0, j_0 de P_k ; on peut les obtenir par les relations algébriques suivantes : soit

$$\frac{2a_{i_0j_0}}{a_{j_0j_0} - a_{i_0i_0}} = \tan(2\theta) = \frac{1}{r} \quad \text{et} \quad t = \tan\theta.$$

On a les relations trigonométriques suivantes :

$$c = \cos\theta = \frac{1}{\sqrt{1+t^2}} \qquad s = \sin\theta = \frac{t}{\sqrt{1+t^2}} \quad \text{et} \quad \tan(2\theta) = \frac{2t}{1-t^2} = \frac{1}{r}\,.$$

D'où, t est égal à l'unique solution du trinôme

$$t^2 + 2rt - 1 = 0,\ \text{avec } t \in\ ]-1\,;0[\,\cup\,]0\,;1[\,.$$

Les formules donnant les éléments de la matrice B du lemme se réécrivent alors (calcul à vérifier !) :

$$\begin{aligned} b_{i_0j} &= c * a_{i_0j} - s * a_{j_0j} \quad \text{si } j \neq i_0, j \neq j_0\\ b_{j_0j} &= c * a_{j_0j} - s * a_{i_0j} \quad \text{si } j \neq i_0, j \neq j_0\\ b_{i_0i_0} = a_{i_0i_0} - t * a_{i_0j_0} \quad &, \quad b_{j_0j_0} = a_{j_0j_0} + t * a_{i_0j_0} \quad , \quad b_{i0j_0} = b_{j0i_0} = 0.\end{aligned}$$

5ème programme

Écrire un programme qui diagonalise par cette méthode une matrice symétrique réelle.

Corrigé :

```
from math import sqrt
import numpy as np

def Givens(A, i, j):
    # calcule la matrice de rotation comme dans le lemme
    if A[i,j] == 0:
        c, s = 1, 0
    else:
        r = (A[j,j] - A[i,i])/(2*A[i,j])
        t = -r + sqrt(r*r + 1)
        c = 1/sqrt(1+t*t)
        s = c*t
    return c, s

def norme(A):
    # il s'agit de la somme des carrés des élts non diagonaux
    return np.linalg.norm(A - np.diag(np.diag(A)))

def Jacobi(A, eps):
    nmax = 10000 # nombre max d'itérations
    erreur = eps + 1
    niter = 0
    n = A.shape[0]
    P = np.identity(n) # future matrice de passage
    while erreur > eps and niter < nmax:
        ind = np.argmax(abs(A - np.diag(np.diag(A))))
        i0, j0 = np.unravel_index(ind, (n,n))
        c, s = Givens(A, i0, j0)

        G = np.identity(n)
        G[i0,i0] = c
        G[i0,j0] = s
        G[j0,i0] = -s
        G[j0,j0] = c
        # on pourrait se contenter de faire :
        # A = np.dot(np.transpose(G), np.dot(A,G))
        # mais beaucoup de produits pour rien!
        t = s/c
        L = A[i0,:].copy()
        x, y, z = A[i0,i0], A[i0,j0], A[j0,j0]
        for j in range(n):
            if j != j0:
                A[i0,j] = c*A[i0,j]-s*A[j0,j]
                A[j, i0] = A[i0, j]
        for j in range(n):
            if j != i0:
                A[j0,j] = c*A[j0,j] + s*L[j]
                A[j, j0] = A[j0, j]
        A[i0,j0] = 0
        A[j0,i0] = 0
        A[i0,i0] = x - t*y
        A[j0,j0] = z + t*y
```

```
            P = np.dot(P,G) # matrice de passage

            erreur = norme(A)
            niter += 1
        if niter == nmax:
            return None
        return np.diag(A), P

    eps = 1e-10
    for n in range (2,10):
        #A = -2*np.identity(n) + np.eye(n, k=-1) + np.eye(n, k=1)
        A = np.identity(n) + np.ones((n,n))
        # on se contente d'afficher les valeurs propres
        print(Jacobi(A,eps)[0])
```

```
[ 1.  3.]
[ 1.  1.  4.]
[ 1.  1.  1.  5.]
[ 1.  1.  1.  1.  6.]
[ 1.  1.  1.  1.  1.  7.]
[ 1.  1.  1.  1.  1.  1.  8.]
[ 1.  1.  1.  1.  1.  1.  1.  9.]
[  1.   1.   1.   1.   1.   1.   1.   1.  10.]
```