TD d'informatique

MPSI/PCSI **Pr. Youssef Ouassit**

Simulation Numérique
« Le module numpy»

Instruction	Description
import numpy as np	Importer le module numpy et lui donner l'alias np.
np.pi	La constante π
T1 = np.array([1 , 0 , 4, 9, 0]) T2 = np.array([2 , 3 , -4 , 4, 3])	Créer un tableau numpy à partir d'une liste
>>> T1[0] 1	Accès à un élément du tableau
A = np.array([[1, 2, 3], [4, 5, 4]])	Créer une matrice numpy
A[1, 2] >> 4	Accéder à une composante de la matrice
A.size >> 6	Renvoie la taille d'un tableau
A.shape >> (2, 3)	Renvoie la forme d'un tableau
A[0 : 2, 0 : 2] >> array([[1, 2], [4, 5]])	Extraire une sous-matrice
np.zeros(6) >> array([0., 0., 0., 0., 0., 0.]) np.zeros((2,3)) >> array([[0., 0., 0.],[0., 0., 0.]])	Création d'un tableau ou matrice nulle
np.arange(3, 15, 2) >> array([3, 5, 7, 9, 11, 13])	Créer un tableau qui contient les valeurs de 3 à 14 avec le pas 2
np.ones(4) >> array([1., 1., 1., 1.]) np.ones((2,4)) >> array([[1., 1., 1., 1.], [1., 1., 1., 1.]])	Création d'un tableau ou matrice remplie des 1
np.eye(3) >> array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])	Créer une matrice identité d'ordre 3
np.dot(A, B)	Produit matriciel de A et B
A=np.array([2,3 ,1]) B=np.array([0,2 ,4]) A+B >> np.array([2,5,5]) A*B >> np.array([0,6,4]) A+2 >> np.array([4,5,3])	Opérations sur les tableaux
np.linspace(3, 9, 10)	np.linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments équidistant.

np.copy(T)	Copie d'un tableau
C = A[: , [1]] >> array([[2],[5]])	Extraction sous matrice
T=np.array([0, np.pi/2, 2]) np.sin(T) >> array([0. , 1. , 0.90929743])	Fonctions sur vercteurs
A = array([[3,5,2],[1,8,4]]) A >=5 >> array([[False,True,False],[False,True,False]])	Numpy compare les paires d'éléments correspondants. Le résultat est une matrice de constantes booléennes, de valeurs False ou True. Ceci peut permettre d'exprimer une expression logique dans une instruction conditionnelle de façon condensée.
A = np.array([3,5,2,1,8,4]) B=A.reshape((2,3)) >> array([[3,5,2],[1,8,4]])	Redimensionner un tableau
np.transpose(M)	Transposé d'une matrice M
np.diag(v)	Créer une matrice de diagonale v
np.vdot(v1, v2)	Produit scalaire
np.concatanate(A, B)	Concaténer deux tableaux de n'importe quelle dimension.
np.linalg.inv(A)	Inverse d'une matrice
np.linalg.solve(A,Y)	Résoudre les systèmes linéaires de forme AX=Y
np.linalg.det(A)	Déterminent de la matrice A
np.linalg.eigvals(a)	La liste des valeurs propres
np.linalg.eig(a)	La liste des valeurs et vecteurs propres
np.linalg.norm(A)	La norme d'un vecteur ou d'une matrice
np.linalg.cholesky(a)	Retourne the la décomposition de Cholesky
np.all(A)	Test whether all array elements along a given axis evaluate to True

Exercice 1 :

A l'aide de numpy, résoudre les systèmes d'équations linaires suivants d'inconnues x; y; z :

(1)

$$\begin{cases} x+y+z=1 \\ x-2y+z=0 \\ 2x-y+z=2 \end{cases}$$

(2)

$$\begin{cases} x+y+z=a \\ x-2y+z=b \\ 2x-y+z=c \end{cases} \quad \text{avec} \quad \begin{cases} 3a+b-c=3 \\ a+b+c=3 \\ a-2b+2c=1 \end{cases}$$

De 2 façons différentes, avec les fonctions solve() ou inv() du sous module linalg de numpy. Que constatez-vous ?

Exercice 2 :

Déterminer les polynômes de degré au plus 2 prenant pour valeur 1,4 et 3 respectivement en 0, 1 et 2.

Exercice 3 :

A l'aide de maatplotlib.pyplot tracer :

1. La courbe paramétrée x(t) = t .cos(t); y(t) = t.sin(t) pour t ∈ [0 , 10].
2. Une ellipse de fonction paramétrique :

x = u + a.cos(t) ;
y = v + b.sin(t)

u #x-position of the center
v #y-position of the center
a #radius on the x-axis
b #radius on the y-axis

Exercice 4 :

Les fonctions demandées peuvent ne faire qu'une ligne de code !
On pourra utiliser les fonctions suivantes :

- np.triu(A) Renvoie le tableau A dont les éléments au-dessous de la diagonale ont été annulés.
- np.tril(A) Renvoie le tableau A dont les éléments au-dessus de la diagonale ont été annulés.

1. Écrire une fonction est_triangulaire_sup(M) qui renvoient True si la matrice M est triangulaire supérieure, False sinon.

2. Écrire une fonction est_diagonale(M) qui renvoient True si la matrice M est diagonale, False sinon.

3. Écrire une fonction est_inversible(M) qui renvoient True si la matrice M est inversible, False sinon

Exercice 5 : Pivot de Gauss pour la résolution du système AX=b

On résoudra cet exercice à l'aide la librairie numpy.

1. Programmer une fonction remontee(T,b) qui résout le système T.X = b, où T est une matrice triangulaire supérieure. Si le système est de Cramer, remontee renvoie l'unique solution, sinon elle renvoie False.

Le pivot de Gauss est une des méthodes les plus stable pour la résolution des systèmes linéaires, la première phase de cette méthode est de rendre la matrice A triangulaire supérieure en appliquant des transvection sur les lignes, puis une étape de remontée est appliquée pour calculer la solution :

2. Programmer une fonction echanger_ligne(A, i, j) qui permet de permuter les lignes d'indices i et j de la matrice A.
3. Programmer un fonction pivot_partiel(A, ind) qui retourne l'indice de la ligne contenant le ind ème pivot maximal de la matrice A.
4. Programmer un fonction transvection(A, i, j, mu) qui calcule A[i] = A[i] + mu*A[j].
5. Programmer une fonction solution_systeme(A0,b0) qui résout le system A0.X = b0 avec l'algorithme du pivot de Gauss.

Exercice 6 : Méthode de Strassen

L'algorithme de Strassen permet de calculer un produit matriciel en effectuant moins de multiplications, car la méthode "classique" n'est pas optimale. Cet algorithme ne s'applique que sur les matrices dont la taille est une puissance de 2. Ce n'est pas vraiment une limitation car n'importe quelle matrice peut devenir de cette forme en complétant les lignes et les colonnes par des 0.

L'algorithme de Strassen est récursif : à chaque étape la matrice est divisée en quatre sous-matrices, l'amélioration consistant à effectuer des opérations plus simples entre celles-ci par rapport à la méthode dite classique. Le cas d'arrêt de la récusivité est celui où les matrices sont de taille 1x1.

NB : Dans le cas où une des dimensions est impair, on se ramène au cas précédent en ajoutant une ligne ou une colonne en bas ou à droite des matrices. On pourra utiliser la fonction **concatenate** de la librairie numpy.

Soient A et B deux matrices carrées de taille n, dont on souhaite calculer le produit C=A×B. On suppose, pour se simplifier la vie, que n est de la forme une puissance de 2 : si tel n'est pas le cas, il suffit de compléter les matrices avec des 0.

Algorithme :

Les trois matrices A, B et C sont divisées en matrices par blocs de taille égale :

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Dans la méthode classique on calcule le produit par :

$C_{1,1}=A_{1,1}\times B_{1,1}+A_{1,2}\times B_{2,1}$
$C_{1,2}=A_{1,1}\times B_{1,2}+A_{1,2}\times B_{2,2}$
$C_{2,1}=A_{2,1}\times B_{1,1}+A_{2,2}\times B_{2,1}$
$C_{2,2}=A_{2,1}\times B_{1,2}+A_{2,2}\times B_{2,2}$

On constate que cette méthode nécessite 8 multiplications de matrices pour calculer les Ci,j, et ce comme dans le produit classique.

L'idée de Strassen est de réussir à obtenir ces Ci,j avec seulement 7 multiplications, en introduisant les matrices suivantes :

$M1=(A_{1,1}+A_{2,2})\times(B_{1,1}+B_{2,2})$
$M2=(A_{2,1}+A_{2,2})\times B_{1,1}$
$M3=A_{1,1}\times(B_{1,2}-B_{2,2})$
$M4=A_{2,2}\times(B_{2,1}-B_{1,1})$
$M5=(A_{1,1}+A_{1,2})\times B_{2,2}$
$M6=(A_{2,1}-A_{1,1})\times(B_{1,1}+B_{1,2})$
$M7=(A_{1,2}-A_{2,2})\times(B_{2,1}+B_{2,2})$
Les Ci,j sont alors exprimées comme

$C_{1,1}=M1+M4-M5+M7$
$C_{1,2}=M3+M5$
$C_{2,1}=M2+M4$
$C_{2,2}=M1-M2+M3+M6$

On fait plus d'additions et de soustractions matricielles (qui ont une complexité linéaire) que dans la méthode naïve, mais une multiplication de moins : le gain est énorme.

Le procédé est répété jusqu'à ce que les matrices A et B soient tailles raisonnables.

Implémenter en python une fonction récursive **produit_strassen(A, B)** qui calcule le produit matricielle A*B. **Pour simplifier on suppose que A et B sont des matrices carrées**.