Numpy et les tableaux

INFORMATIQUE COMMUNE - TP n° 1.7 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ importer la bibliothèque Numpy
- ☞ utiliser un tableau Numpy mono et multidimensionnel dans un algorithme simple
- ☞ écrire des opérations élément par élément

A Tableaux numpy

En Python, les tableaux peuvent être implémentés par :

- une liste Python : il s'agit alors d'un tableau dynamique dont la taille change au gré de ajouts, des retraits et des modifications de la liste par `append` ou `pop`. La liste Python représente correctement des tableaux à une seule dimension.
- un type Numpy `array` : il s'agit alors d'un tableau statique, c'est-à-dire dont la taille ne peut pas changer. Ce type peut représenter des tableaux de toutes les dimensions.

Pour comprendre les différences entre les listes et les vecteurs, on propose de définir deux tableaux Numpy à partir de deux listes comme suit :

```
import numpy as np

L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = [L1, L2]

v1 = np.array(L1)
v2 = np.array(L2)
m = np.array([L1, L2])
```

A1. Exécuter puis commenter ce qui est renvoyé par les commandes suivantes :

```
print(L1 + L2)
print(v1 + v2)
print(2 * L1)
print(2 * v1)
print(1.5 * L1)
print(1.5 * v1)
print(L1 ** 2)
print(v1 ** 2)
L1.append(7)
v1.append(7)
print(np.sin(v1))
```

```
print(L3[1][2])
print(m[1, 2])
print(L3[1,2])
```

Solution :

Code 1 – Tableaux numpy

```
import numpy as np

L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = [L1, L2]

v1 = np.array(L1)
v2 = np.array(L2)
m = np.array([L1, L2])

print(L1 + L2)  # List concatetation [1, 2, 3, 4, 5, 6]
print(v1 + v2)  # element by element addition # [5 7 9]
print(2 * L1)  # List demultiplication [1, 2, 3, 1, 2, 3]
print(2 * v1)  # element by element multiplication [2 4 6]
# print(1.5 * L1) # TypeError: can't multiply sequence by non-int of type 'float
    '
print(1.5 * v1)  # element by element multiplication [1.5 3.  4.5]
# print(L1 ** 2) # TypeError: unsupported operand type(s) for ** or pow(): 'list
    ' and 'int'
print(v1 ** 2)  # element by element exponentiation [1 4 9]
L1.append(7)  # append an element to the list
# v1.append(7) # AttributeError: 'numpy.ndarray' object has no attribute 'append
    '
print(np.sin(v1))  # [0.84147098 0.90929743 0.14112001]
print(L3[1][2])  # nested list 6
print(m[1, 2])  # matrice 6
# print(L3[1, 2]) # TypeError: list indices must be integers or slices, not
    tuple
```

B Calcul élément par élément

B1. En informatique, on dispose de trois notes pour toute une promotion de n élèves.

(a) À l'aide d'une fonction de prototype `create_marks(n)`, créer des vecteurs `a`, `b`, `c` représentant ces notes allant de 0 à 20. On utilisera la fonction `np.random.rand` pour générer chaque vecteur de note.

Solution :

```
def create_marks(n):
    a = np.random.rand(n) *20
    b = np.random.rand(n) *20
    c = np.random.rand(n) *20
```

```
    return a,b,c
```

(b) À l'aide d'une fonction de prototype `with_loops(a,b,c)` et **en utilisant des boucles**, calculer le vecteur contenant la moyenne de chaque élève.

Solution :

```
def with_loops(a,b,c):
    assert len(a) == len(b)
    assert len(b) == len(c)
    n = len(a)
    m = np.zeros((n,))
    for i in range (n):
        m[i]= (a[i] + b[i] + c[i])/3
    return m
```

(c) À l'aide d'une fonction de prototype `without_loops(a,b,c)` et **en utilisant le calcul élément par élément**, calculer le vecteur contenant la moyenne de chaque élève.

Solution :

```
def without_loops(a,b,c):
    return (a+b+c)/3
```

B2. On dispose d'un vecteur `u` de dimension 100. Calculer la norme de u :

(a) en utilisant une boucle pour parcourir le vecteur.

Solution :

```
def norm_loop(u):
    acc = 0
    for i in range(len(u)):
        acc += math.sqrt(u[i]*u[i])
    return acc
```

(b) en utilisant le calcul élément par élément.

Solution :

```
def norm(u):
    return np.sum(np.sqrt(u*u))

u = np.random.rand(200)
assert np.allclose(norm(u), norm_loop(u))
print(norm(u), norm_loop(u))
```

C Attracteurs étranges

En automatique et dans le cadre de l'étude des systèmes dynamiques en général, un attracteur est un ensemble d'états vers lesquels un système dynamique évolue à partir d'une grande variété de conditions initiales. Selon le système considéré, un attracteur peut être un point, un ensemble fini de points, une courbe ou une variété.

L'attracteur de Clifford est un attracteur étrange. Il est défini par deux équations discrètes dans le temps qui déterminent les emplacements x et y sur le chemin d'une particule à travers un espace en deux dimensions. Étant donné un point de départ (x_0, y_0) et les valeurs de quatre paramètres a, b, c et d, on peut le modéliser comme suit :

$$\begin{aligned} x_{n+1} &= \sin\left(a y_n\right) + c\cos\left(a x_n\right) \\ y_{n+1} &= \sin\left(b x_n\right) + d\cos\left(b y_n\right) \end{aligned} \tag{1}$$

À chaque étape temporelle n, les équations permettent de définier l'emplacement des la particule pour l'instant $n+1$.

Les constantes `a`, `b`, `c` et `d` sont des constantes globales.

```
a = -1.3
b = -1.3
c = -1.8
d = -1.9
```

C1. Écrire une fonction de prototype `clifford(x, y)` qui renvoie la position d'une particule à l'instant $n+1$ si `x` et `y` représente la position à l'instant n.

Solution :

```
def clifford(x, y):
    return sin(a * y) + c * cos(a * x), sin(b * x) + d * cos(b * y)
```

C2. Écrire une fonction de prototype `trajectoire(f, x0, y0, n)` qui renvoie la trajectoire `(X,Y)` d'une particule de position initiale `(x0, y0)` jusqu'à l'instant `n`. On représentera `X` et `Y` par un Numpy array à $n+1$ éléments.

Solution :

```
def trajectoire(f, x0, y0, n):
    x = np.zeros(n + 1)
    y = np.zeros(n + 1)
    x[0], y[0] = x0, y0
    for i in np.arange(n - 1):
        x[i + 1], y[i + 1] = f(x[i], y[i])
    return x, y
```

C3. Tracer l'attracteur dans la plan en utilisant les commandes suivantes :

```
plt.style.use('dark_background')
plt.figure()
```

```
plt.plot(X, Y, ',', color='white', alpha = 0.25, markersize = 0.25)
plt.title("Attracteur de Clifford")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

D Trier un tableau

Cette section permet de manipuler un tableau Numpy à une seule dimension et de réviser un peu.

D1. Créer un tableau `tab` de type Numpy `array` comportant 20 éléments choisis aléatoirement entre 0 et 100 exclu. Utiliser la fonction `np.random.randint`.

Solution :

```
import numpy as np
tab = np.random.randint(100, size=20)
print(t.shape)
```

D2. En implémentant le tri par insertion, trier le tableau créé précédemment.

Solution :

```
def insert_sort(t):
    # len(t) renvoie la taille de la première dimension du tableau
    # comme t.shape[0]
    for i in range(1, t.shape[0]):
        to_insert = t[i]
        j = i
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert
```

D3. En programmant récursivement, écrire une fonction de prototype qui tri par insertion le tableau.

Solution :

```
def rec_insert_sort(t, n):
    if n == 1 or n == 0:
        return t
    else:
        rec_insert_sort(t, n-1)
        to_insert = t[n - 1]
        j = n - 1
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert
```

E Calcul matriciel

Un système dynamique peut être décrit par des équations d'état. Il est souvent possible de la linéariser et on obtient un système sous la forme :

$$\begin{cases} \dot{X} = AX + Bu \\ Y = CX \end{cases} \tag{2}$$

où X le vecteur d'état, u l'entrée et Y la sortie du système sont des fonctions du temps. La matrice A est dite matrice d'évolution, B la matrice de commande et C la matrice d'observation.

On cherche à simuler l'évolution d'un pendule suivant : ce système possède une seule sortie et l'entrée est nulle (le système n'est pas soumis à une excitation). La représentation d'état du système est :

$$\begin{cases} \dot{X} = \begin{pmatrix} 0 & 1 \\ -3 & 0 \end{pmatrix} X \\ y = \begin{pmatrix} 1 & 0 \end{pmatrix} X \end{cases} \tag{3}$$

où X représente la position angulaire θ et la vitesse angulaire $\dot{\theta}$ du système à un instant t.

E1. Définir les constantes `A`, `C` du système.

Solution :

```
import numpy as np

A = np.array([[0, 1], [-3, 0]])
C = np.array([1, 0])
```

E2. Quelle est la dimension de X ?

Solution : Comme la matrice A est de dimension $(2,2)$ le vecteur X est de dimension $(2,1)$.

E3. Programmer le calcul de la sortie `Y` en fonction de `X`.

Solution :

```
def obs(X):
    return C @ X
```

E4. Programmer le calcul de la dérivée du vecteur d'état en fonction de `X` et `u`.

Solution :

```
def evol(X):
    return A @ X
```

E5. Avec la méthode d'Euler, simuler le système dynamique sur 50 secondes.

Solution :

```
def euler_step(x, dt, f):
    return  x + dt * f(x)

def euler_simulation(x0, dt, n, f):
    T= np.linspace(0, (n+1)*dt, n+1)
    X = np.zeros(((n+1),2))
    X[0] = x0
    for k in range(n):
        X[k+1] = euler_step(X[k], dt, f)
    return T, X

time, states = euler_simulation(np.array([0.2,0]), 0.0001, 500000, evol)
plt.figure()
plt.plot(time, states[:,0], 'r', label='position angulaire')
plt.plot(time, states[:,1], 'b', label='vitesse angulaire')
plt.xlabel("Time")
plt.legend()
plt.show()
```

E6. On ajoute un coefficient dans le modèle pour prendre compte le frottement du pendule avec l'air. La matrice d'évolution devient `A = np.array([[0, 1], [-3, -0.3]])`. Simulez de nouveau le système et commenter.

Solution : On observe l'amortissement du mouvement du pendule !