

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
- ☞ effectuer un calcul élément par élément
- ☞ effectuer un calcul matriciel simple

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 et dont le type est fixé dès la création. Il est possible de 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 concatenation [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 Tracer expérimental et régression

Le code ci-dessous donne un exemple d'utilisation de Numpy pour établir :

- une régression linéaire,
- une régression non-linéaire

d'un jeu de données expérimentales.

B1. Tester ce code en le copiant numériquement.

```

import numpy as np
import matplotlib.pyplot as plt

```

```
# Données expérimentales
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = np.array([2, 3, 4, 3.5, 5, 6, 4.9, 5.5, 7.6, 8.8])

# Régression linéaire (moindres carrés)
a, b = np.polyfit(x, y, 1)
y_pred = a * x + b # calcul vectoriel

# Graphique
plt.figure(1)
plt.plot(x, y, '+r', label="Données expérimentales")
plt.plot(x, y_pred, label="Régression linéaire")
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Données + régression linéaire")
plt.show()

# Régression non linéaire (moindres carrés)
y = np.array([2, 3, 4, 3.5, 5, 4.5, 4.9, 3.5, 4.1, 2.7])
a, b, c = np.polyfit(x, y, 2)
y_pred = a * x**2 + b * x + c # calcul vectoriel

# Graphique
plt.figure(2)
plt.plot(x, y, '+r', label="Données expérimentales")
plt.plot(x, y_pred, label="Régression non linéaire")
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Données + régression linéaire")
plt.show()
```

C Calcul élément par élément

C1. 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 Numpy `a`, `b`, `c` de taille n représentant ces notes allant de 0 à 20. On utilisera la fonction `np.random.rand` pour générer chaque vecteur de note.

Solution :

```
import numpy as np
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 `m` contenant la moyenne des n élèves.

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
```

C2. 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 += u[i]*u[i]  
    return math.sqrt(acc)
```

- (b) en utilisant le calcul élément par élément et la fonction `np.sum` qui effectue la somme de tous les éléments d'un tableau Numpy.

Solution :

```
def norm(u):  
    return math.sqrt(np.sum(u*u))  
  
# TESTS  
u = np.random.rand(200)  
assert math.isclose(norm(u), norm_loop(u))  
print(norm(u), norm_loop(u))
```

D Trier un tableau

Cette section permet de manipuler un tableau Numpy à une seule dimension et de réviser un peu. On sait créer un tableau aléatoire d'entiers avec Numpy en utilisant le code suivant :

```
import numpy as np
rng = np.random.default_rng(42)
tab = rng.integers(low=0, high=100, size=3)
```

D1. En implémentant le tri par insertion, trier le tableau créé précédemment.

Solution : La syntaxe de cet algorithme peut être la même qu'avec les listes Python.

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

D2. 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 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. Un attracteur fournit un aperçu du comportement à long terme d'un système.

Lorsque le comportement du système est régulier, la trajectoire de l'attracteur dans l'espace des phases (paramètres) prend la forme d'un point ou d'un cycle limite. À termes, cela signifie que le système n'évolue plus ou évolue de manière périodique, c'est-à-dire il revient dans le même état à intervalles réguliers. Les systèmes régulés (chauffage, régulateur de vitesse d'une voiture, altitude d'un avion en vol) atteignent leur point de fonctionnement d'après leur consigne et se stabilise.

L'attracteur de Clifford est un attracteur étrange car sa trajectoire dans l'espace des phases est chaotique, c'est-à-dire complexe, apériodique et apparemment aléatoire. La structure même de l'attracteur

est fractale et un tel système est très sensible aux conditions initiales : de petits changements dans les conditions initiales conduisent à des résultats radicalement différents au fil du temps (effet papillon).

L'attracteur de Clifford 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 :

$$\begin{aligned}x_{n+1} &= \sin(Ay_n) + C \cos(Ax_n) \\ y_{n+1} &= \sin(Bx_n) + D \cos(By_n)\end{aligned}\tag{1}$$

À chaque étape temporelle n , les équations permettent de définir l'emplacement de 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
```

- E1. 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)
```

- E2. Écrire une fonction de prototype `trajectoire(f, x0, y0, n)` qui renvoie la trajectoire (X, Y) d'une particule de position initiale (x_0, y_0) 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):
        x[i + 1], y[i + 1] = f(x[i], y[i])
    return x, y
```

- E3. Tracer l'attracteur dans la plan en utilisant les commandes suivantes :

```
#plt.style.use('dark_background')
plt.figure()
plt.plot(X, Y, ',', color='black', alpha = 0.25, markersize = 0.25)
plt.title("Attracteur de Clifford")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

Votre résultat devrait ressembler à la figure 1.

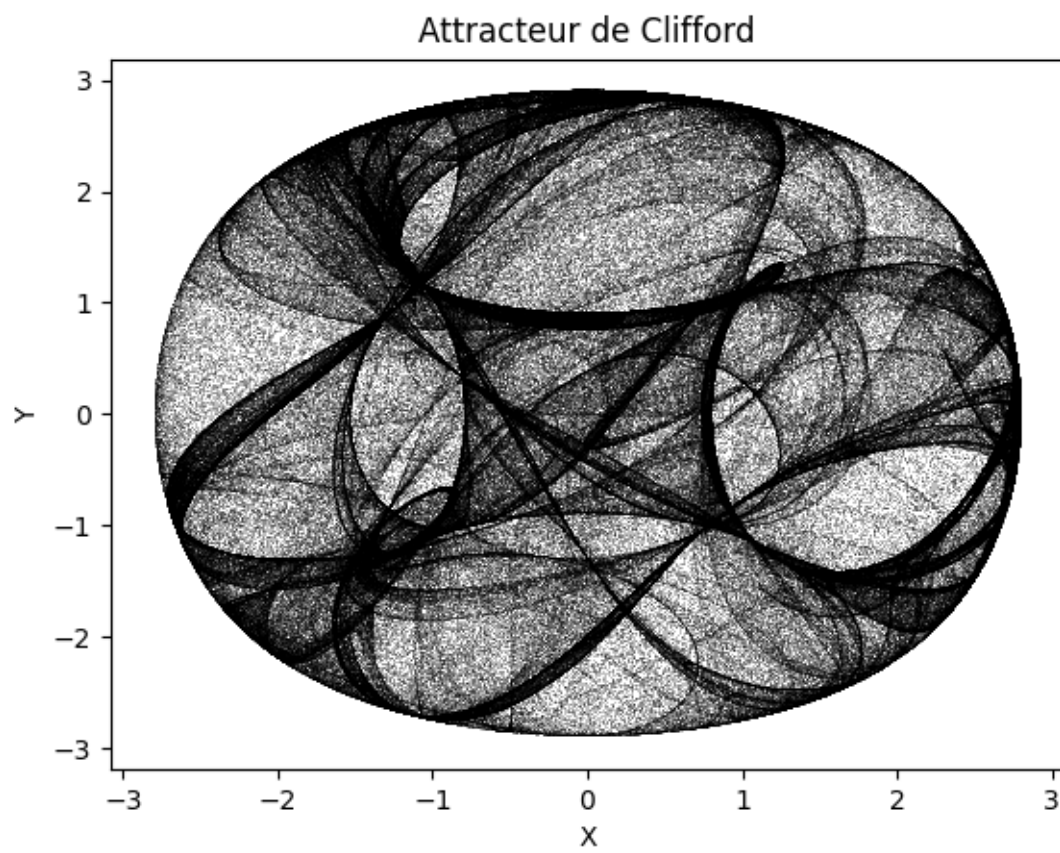


FIGURE 1 – Attracteur de Clifford

F Calcul matriciel

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

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

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 simple. Ce système linéarisé possède une seule sortie : on observe sa position angulaire, l'angle qu'il fait avec la verticale. L'entrée est nulle, c'est-à-dire que le système n'est pas soumis à une excitation, il oscille librement depuis son point de départ. La représentation d'état du système est :

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

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

F1. Définir les constantes A , C du système.

Solution :

```
import numpy as np

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

F2. Quelle est la dimension de X ?

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

F3. Programmer le calcul de la sortie Y en fonction de x .

Solution :

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

F4. Programmer le calcul de la dérivée du vecteur d'état en fonction de x et u .

Solution :

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

F5. 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()
```

F6. On ajoute un coefficient dans le modèle pour prendre compte le frottement du pendule avec l'air. La matrice d'évolution devient $A = \text{np.array}([0, 1], [-3, -0.3])$. Simuler de nouveau le système et commenter.

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

