Informatique

11

Numpy - Array

Cours

1.I.1 Matrices - Array

1.I.1.a Contexte

Une matrice est généralement représentative d'un système linéaire et c'est pour les systèmes linéaires que je vais les aborder. En effet, il est très courant d'obtenir un système linéaire lorsque l'on résout des équations scientifiques sur un solide ou un fluide par exemple.

Comme vous le savez sans doute, on peut assez simplement passer d'un système linéaire à un système matriciel équivalent, exemple :

$$\begin{cases} a_1 x + b_1 y + c_1 z = d_1 \\ a_2 x + b_2 y + c_2 z = d_2 \\ a_3 x + b_3 y + c_3 z = d_3 \end{cases} \Leftrightarrow \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Avec a_i, b_i, c_i, d_i des coefficients réels constants.

On l'écrira souvent :

$$KU = F \quad ; \quad K = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \quad ; \quad U = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad ; \quad F = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

La solution de ce système s'obtient alors « très simplement » par inversion de la matrice K, si elle est inversible évidemment :

$$U = K^{-1}F$$

En effet :

$$KU = F \Leftrightarrow K^{-1}KU = K^{-1}F \Leftrightarrow U = K^{-1}F$$

Car $K^{-1}K = I$

Très simplement est entre guillemets, car selon les problèmes traités, son inversion numérique n'est pas toujours juste… Mais vous aurez l'occasion de voir ça dans vos études futures.

1.I.1.b Fonctions de Numpy

	`import numpy as np`	

Création d'un vecteur	$F = \begin{bmatrix}1\\2\\3\end{bmatrix}$; $H = \begin{bmatrix}0\\0\\0\end{bmatrix}$
	$G = \begin{bmatrix}0\\\vdots\\10\end{bmatrix}, 100\ termes$
		`F = np.array([1,2,3])`
		`F = np.empty([1,3]) ; V[0,:] = [1,2,3]`
		`F = np.empty([3,1]) ; V[:,0] = [1,2,3]`
		`G = np.linspace(0,10,101)`
		`G = np.arange(0,10.1,0.1)`
		`H = np.zeros([3]) ; H = np.zeros([3,1])`

Création d'une matrice	$K = \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$	`K = np.zeros([3,3])`
	$K = \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$	`K = np.eye(3)`
	$K = \begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	`K = np.ones([3,3])`
	$K = \begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}$	`K = np.array([[1,4,7],[2,5,8],[3,6,9]])`

Accès à un terme d'une matrice ou d'un vecteur	$K = \begin{bmatrix}1&4&7\\2&5&8\\3&\mathbf{6}&9\end{bmatrix}$
	$F = \begin{bmatrix}0\\\mathbf{0}\\0\end{bmatrix}$; $G = \begin{bmatrix}0\\\mathbf{0}\\0\end{bmatrix}$
		`K[2,1]`
		`F = np.zeros([3])` → `F[1]`
		`G = np.zeros([3,1])` → `G[1,0]` **ou** `G[1,:]`

Copie d'une matrice ou d'un vecteur		`B = np.copy(A)`
Transposition d'une matrice (2 dimensions !)		`K.T` ou `np.transpose(K)`
Produit matriciel Produit scalaire	$\begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}30\\36\\42\end{bmatrix}$ $UV = \begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = 14$	`np.dot(K,F)` ≠ `np.dot(F,K)` `np.dot(U,V) = np.dot(V,U)`
Récupération d'une portion d'array **ATTENTION :** **[,] OUI - ~~[][]~~ NON**	$K = \begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}$	Ligne : `K[1,:]` ou `K[1]` Et un exemple qui sert à rien : `K[1][:]`
		Colonne : Attention ! `L[:,1]` OUI - ~~`L[:][1]`~~ NON (ligne 2…)
		`K[0:2,0:2]`
Nb lignes colonnes d'un array de 2 dimensions		`l,c = np.shape(K)`
Dim d'un array à 3 dimensions (par exemple)		`l,c,n = np.shape(K)` `l,c = np.shape(K)` ne fonctionne pas
Nombre de termes d'un array		`np.size(K)`
Produit termes à termes	$U * V = \begin{bmatrix}a\\b\\c\end{bmatrix} * \begin{bmatrix}d\\e\\f\end{bmatrix} \Rightarrow \begin{bmatrix}ad\\be\\cf\end{bmatrix}$	`U*V`
Inversion d'une matrice		`np.linalg.inv(K)`
Résolution d'un système		`[x,y,z] = np.linalg.solve(K,F)`
Calcul d'un déterminant		`np.linalg.det(K)`

1.I.1.c Exemple de résolution

Il existe plusieurs modules permettant de traiter des matrices sous Python. J'ai choisi de vous parler de « Numpy ».

Essayons de résoudre le système suivant :

$$\begin{cases} x + 2z = 1 \\ 3y + 4z = 2 \\ 5y = 3 \end{cases}$$

Soit :

$$KU = F \quad ; \quad K = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 4 \\ 0 & 5 & 0 \end{bmatrix} \quad ; \quad U = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad ; \quad F = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

On peut écrire :

Import de Numpy	`import numpy as np`

Création de la matrice K	`K = np.zeros([3,3])`
	
	`K[0,0] = 1`
	`K[0,2] = 2`
	`K[1,1] = 3`
	`K[1,2] = 4`
	`K[2,1] = 5`

Création du vecteur F	`F = np.array([1,2,3])`
Résolution	`x,y,z = np.linalg.solve(K,F)`

1.I.1.d Utilisation de slices

1.I.1.d.i Principe

Il est possible de récupérer une portion d'un array en utilisant « : ».

Exemples :

```
>>> A = np.array([[1,2,3],[4,5,6],[7,8,9]])

>>> A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

>>> A[:,0]
array([1, 4, 7])

>>> A[0,:]
array([1, 2, 3])

>>> A[:2,:2]
array([[1, 2],
       [4, 5]])

>>> A[1:,1:]
array([[5, 6],
       [8, 9]])
```

```
>>> A
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

1.I.1.d.ii Avantage

Même si on attendra souvent de vous des doubles boucles for « pour faire de l'algorithmique », sachez que l'utilisation de slices améliore grandement les temps d'exécution des algorithmes.

Exemple :

```
N = 1000
import numpy as np
A = np.ones([N,N])

from time import perf_counter as tps
t1 = tps()
for l in range(N):
    for c in range(N):
        A[l,c] = 0

t2 = tps()
Temps_1 = t2-t1
print(Temps_1)

t1 = tps()
A[:,:] = 1
t2 = tps()
Temps_2 = t2-t1
print(Temps_2)

k = Temps_1/Temps_2
print(k)
```

```
0.4554673000000935
0.0017794000000032183
255.96678655264222
```

L'utilisation améliore grandement le temps d'exécution (facteur 255 dans cet exemple).

1.I.1.e Remarques

1.I.1.e.i Dimension supérieure à 2

On peut créer une matrice à l lignes, c colonnes, contenant des nuplets, en écrivant `Mat = np.zeros((l,c,n))`. C'est par exemple utilisé pour les images, qui sont sous python traduites en array contenant des triplets [R,G,B] ou R, G et B sont des entiers codés sur 8 bits (0 à 255).

1.I.1.e.ii Egalité d'array

Attention, l'égalité de deux array ne fonctionne pas. Il faut comparer chaque composante. Rappelons que la comparaison de liste et listes de listes fonctionne.

```
>>> a=np.array([1,2,3])

>>> b=np.array([1,2,3])

>>> a==b
array([ True,  True,  True], dtype=bool)

>>> a==b==True
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: The truth value of an array with more th
an one element is ambiguous. Use a.any() or a.all()
```

1.I.1.e.iii Transposition

Attention, pour que la transposition fonctionne, il est nécessaire de créer des « vecteurs » sous forme d'array à 2 dimensions. Exemples :

Fonctionne	Ne fonctionne pas

`A = np.zeros([3,1])`
`A[:,0] = [1,2,3]`
`AT = np.transpose(A)`
`print(A)`
`print(AT)`
`Prod = np.dot(A,AT)`
`print(Prod)`
	`A = np.array([1,2,3])`
	`AT = np.transpose(A)`
	`print(A)`
	`print(AT)`
	`Prod = np.dot(A,AT)`
	`print(Prod)`

`[[ 1.]`
` [ 2.]`
` [ 3.]]`
`[[ 1.  2.  3.]]`
`[[ 1.  2.  3.]`
` [ 2.  4.  6.]`
` [ 3.  6.  9.]]`
	`[1 2 3]`
	`[1 2 3]`
	`14`

Il faudra donc penser à l'initialiser avant de le remplir !

1.I.1.e.iv Append

Append est un outil des listes, il n'existe pas en array.

```
>>> A = np.array([1,2,3])

>>> A.append(2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has
no attribute 'append'
```

1.I.1.f Avantages des array

1.I.1.f.i Rapidité de calcul et mémoire

La bibliothèque Numpy de Python présente les gros avantages, par rapport à des listes, de :

- Permettre des calculs plus rapides (complexité en temps)
- Prendre moins de mémoire (complexité en espace)

1.I.1.f.ii Opérations

Lorsque l'on manipule de vrais vecteurs, on souhaite généralement pouvoir les sommer ou en faire la différence. On peut aussi vouloir réaliser une opération sur tous les termes d'une matrice ou d'un vecteur. Ces opérations sur les listes ne fonctionnement pas. Il est donc très intéressant de convertir les listes en array pour réaliser ces opérations.

```
>>> A = [1,1]

>>> B = [2,2]

>>> A+B
[1, 1, 2, 2]

>>> 2*A
[1, 1, 1, 1]
```

```
>>> import numpy as np

>>> A = np.array(A)

>>> B = np.array(B)

>>> A+B
array([3, 3])

>>> 2*A
array([2, 2])
```

On pourra toujours retranscrire un array en liste même si cela est souvent inutile.

```
>>> A = [t for t in A]

>>> A
[1, 1]

>>> B
array([2, 2])
```

On peut même aller plus loin en réalisant des opérations :

```
M = np.array([[1,2,3],[4,5,6],[7,8,9]])
Mexp = np.exp(M)
Mcos = np.cos(M)
Minv = 1/M
```

```
>>> Mexp
array([[  2.71828183e+00,   7.38905610e+00,   2.00855369e+01],
       [  5.45981500e+01,   1.48413159e+02,   4.03428793e+02],
       [  1.09663316e+03,   2.98095799e+03,   8.10308393e+03]])

>>> Mcos
array([[ 0.54030231, -0.41614684, -0.9899925 ],
       [-0.65364362,  0.28366219,  0.96017029],
       [ 0.75390225, -0.14550003, -0.91113026]])

>>> Minv
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667],
       [ 0.14285714,  0.125     ,  0.11111111]])
```

1.I.1.f.iii Récupération dans des listes

Pour récupérer, par exemple, une colonne dans une liste, deux solutions :

```
L = [[1,2,3],[4,5,6],[7,8,9]]
Col1 = [L[i][0] for i in range(len(L))]
```

```
>>> Col1
[1, 4, 7]
```

```
La = np.array(L)
Col1 = La[:,0]
```

```
>>> Col1
array([1, 4, 7])
```

La différence réside dans le résultat, qui dans un cas est une liste, et dans l'autre un array. Mais on pourra reconvertir un array en liste en faisant par exemple :

```
>>> A = np.array([1,2,3])

>>> L = [A[i] for i in range(len(A))]

>>> A
array([1, 2, 3])

>>> L
[1, 2, 3]
```

1.I.1.g Types des nombres d'un array

Lorsque l'on crée un array, le type des nombres qui le composent est défini par défaut :

```
>>> A = np.array([1,2,3])

>>> type(A[0])
<class 'numpy.int32'>
```

```
>>> A = np.array([1.0,2.0,3.0])

>>> type(A[0])
<class 'numpy.float64'>
```

On peut sois même préciser le type des nombres qui le composent :

```
>>> A = np.array([1,2,3],dtype='uint8')

>>> type(A[0])
<class 'numpy.uint8'>
```

Dès lors qu'un élément est d'un type donné, il faut faire attention à sa manipulation. En effet, lorsqu'on le manipule dans l'array, le résultat des opérations effectuées sur lui est du même type que les nombres de l'array (ici int8), alors qu'en dehors, les choses se passent autrement… (à la première ligne, il est converti en int32…) :

```
>>> A[0] + 255
256

>>> A[0] = A[0] + 255

>>> A
array([0, 2, 3], dtype=uint8)
```

On pourra relire le chapitre sur le codage des nombres pour comprendre que le type à un fort intérêt pour éviter l'overflow…

```
>>> D = np.array([400000000])

>>> D[0]**2
__main__:1: RuntimeWarning: overflow encountered in long_s
calars
-66060288

>>> D = np.array([400000000.0])

>>> D[0]**2
1.6e+17

>>> D = np.array([400000000],dtype='float64')

>>> D[0]**2
1.6e+17
```