Introduction aux tableaux numpy
Master IMSD UL

Bruno Pinçon (I.E.C.L.) bruno.pincon@univ-lorraine.fr

2021-2022

Les tableaux numpy I

principes

Contrairement aux listes, les tableaux numpy (numpy.ndarray) sont formés d'éléments de même type occupant tous la même place en mémoire (flottants sur 4 ou 8 octets, flottants complexes, entiers "classiques" sur 1, 2, 4 ou 8 octets). Ces tableaux sont caractérisés [a] par :

- le nombre de dimensions du tableau (tableau 1d, 2d, 3d, ...) ;
- le profil du tableau, c'est à dire le nombre d'éléments dans chacune des dimensions (voir exemples page suivante) ;
- le type des éléments du tableau.

---

a. Nous donnons ici les caractéristiques principales, les tableaux numpy sont des objets complexes...

Les tableaux numpy II

Sur profil des tableaux numpy (exemples)

- un tableau 1d de profil (n) comporte donc n éléments dans son unique dimension ; un tel tableau pourra être assimilé à un vecteur usuel à n composantes **mais pas à une matrice unicolonne ou uniligne** (cad ce qu'on appelle couramment un vecteur colonne et vecteur ligne) ;
- un tableau 2d de profil (m, n) pourra être assimilé à une matrice $m \times n$; dans ce cas (cad dans la visualisation classique que nous avons d'un tel tableau) , le nombre d'éléments dans la 1 ère dimension (m) est le nombre de lignes et le nombre d'éléments dans la 2 ème (n) est le nombre de colonnes, le nombre total d'éléments est bien sûr mn ;

Les tableaux numpy III

- un tableau 3d de profil (m, n, p) comporte m éléments dans la 1 ère dimension, n dans le 2 ème et p dans la troisième (et possède mnp éléments en tout). On peut éventuellement visualiser un tel tableau en considérant la 3 ème dimension comme autant d'étages (cad p ici) et avec cette image m et n peuvent être toujours considérés comme le nombre de lignes et de colonnes ;
- pour un tableau 4d de profil (m, n, p, q) le même vocabulaire s'applique (q est le nombre d'éléments dans la 4 ème dimension et le tableau comporte en tout $mnpq$ éléments) sauf qu'il est difficile de visualiser la 4 ème dimension, peut-être en la considérant comme le temps...
- et bien sûr le nombre de dimensions n'est pas limité, rien n'interdit de considérer des tableaux 5d, 6d, etc.

Les tableaux numpy IV

Quelques fonctions de création de tableaux numpy

- `array` : permet de transformer une liste en tableau. Le type des éléments du tableau est inféré à partir du type de chaque élément de la liste. Quelques exemples :

```
>>> import numpy as np
>>> T = np.array([1, 5, 8])
>>> T.dtype    # permet de connaître le type des éléments
dtype('int64')
```

La liste ne contenait que des entiers, il en résulte un tableau d'entiers sur 8 octets (64 bits...), ce qui est sans doute la norme sur une machine 64 bits.

Rmq : il est possible de préciser le type désiré avec l'argument optionnel nommé `dtype=` *(voir exemple page suivante)*.

Les tableaux numpy V

```
>>> T = np.array([1, 5, 8, np.pi], dtype='int8')
>>> T.dtype    # np.pi est devenu (np.int8(np.pi)
dtype('int8')
```

Voici un exemple avec des données de types différents et ici la liste (de listes qui ont toutes le même nombre d'éléments) est considérée comme la liste des lignes d'un tableau 2d :

```
>>> T = np.array([[1, 5, 8],[2.0, -1, 9]])
>>> T
array([[ 1.,  5.,  8.],
       [ 2., -1.,  9.]])
>>> T.dtype
dtype('float64')
>>> T.ndim  # attribut donnant le nb de dimensions
2
>>> T.shape # attribut donnant le profil
(2, 3)
```

Les tableaux numpy VI

- `zeros` : permet de créer des tableaux de 0, par défaut (cad sans préciser le type attendu avec dtype=) on obtient un tableau de doubles (float64)). Le profil attendu du tableau est donné avec un tuple (un simple entier convient pour un tableau 1d) :

```
>>> T = np.zeros(4)  # tableau 1d à 4 éléments
>>> T.dtype
dtype('float64')
>>> U = np.zeros((4,3),dtype='int')  # tableau 2d (4,3)
>>> U.dtype
dtype('int64')
```

- `ones` : idem à la fonction précédente mais le tableau est initialisé avec des 1 (ici avec des "vrais" pour un tableau de booléens) :

```
>>> B = np.ones(6,dtype='bool')
>>> B
array([ True,  True,  True,  True,  True,  True])
```

Les tableaux numpy VII

- `linspace(a,b,n)` : permet de créer une subdivision régulière d'un intervalle $[a, b]$ avec n composantes en tout (soit $n-1$ intervalles, en notant $\delta = \frac{b-a}{n-1}$, la composante d'indice k est donc $a + k\,\delta$) :

```
>>> x = np.linspace(0,1,5)
>>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

- `logspace(a,b,n)` : pratique pour explorer l'infiniment petit ou l'infiniment grand, `logspace` crée une subdivision de 10^a à 10^b en utilisant n valeurs réparties de manière logarithmique :

```
>>> x = np.logspace(-3,3,7)
>>> x
array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03])
```

Les tableaux numpy VIII

- `arange(a,b,pas)` : celle-ci fonctionne de la même manière que la fonction `range` de python (qui ne crée plus de liste depuis la version 3.x mais un "itérateur") ou encore comme les "tranches" python `a:b:pas`, un tableau 1d est créé, ses composantes successives étant obtenues partant de *a* et incrémentées de *pas* jusqu'à ne pas "dépasser" *b* (*b* est exclu).

```
>>> x = np.arange(0,1,0.1)
>>> x
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> x = np.arange(10,0,-2)
>>> jj = np.arange(0,5)
>>> jj
array([0, 1, 2, 3, 4])
```

Les tableaux numpy IX

- `empty` : permet de créer un tableau non initialisé par défaut (sans préciser le type avec `dtype=` on obtient un tableau de doubles) :

```
>>> T = np.empty((2,2,2))   # tableau 3d à 8 éléments
>>> T.ndim
3
>>> T.dtype    # par défaut encore des doubles
dtype('float64')
>>> T = np.empty((2,2),dtype='int64')  # tableau 2d
>>> T.dtype
dtype('int64')
>>> T      # les bits (déjà) présents dans la zone mémoire
>>>        # sont donc interprétés comme des int64
array([[ 4607182418800017408,  4617315517961601024,
        [ 4611686018427387904, -4616189618054758400]])
```

Les tableaux numpy X

Attributs principaux et deux méthodes

- *T*.`ndim` : donne le nombre de dimensions du tableau *T*
- *T*.`shape` : donne le profil du tableau *T*
- *T*.`dtype` : donne le type des éléments du tableau *T*
- *T*.`size` : donne le nombre total d'éléments du tableau *T*
- *T*.`copy()` : copie le tableau *T*
- *T*.`reshape(profil)` : voir page suivante.

Il existe d'autres attributs et méthodes (voir qq unes ci-après). Une caractéristique de numpy est que souvent les attributs ou méthodes ont un équivalent fonction (entre autres les 4 attributs et 2 méthodes listés ci-dessus sont aussi disponibles en tant que fonctions (`np.ndim(T)`, ..., `np.reshape(T)`). Cette redondance doit permettre de contenter à la fois les utilisateurs qui préfèrent la notation fonctionnelle et ceux qui prèfèrent le paradigme objet.

Les tableaux numpy XI

La fonction ou méthode reshape

Elle permet d'obtenir un tableau de profil différent mais avec le même nombre d'éléments en tout :

```
>>> x = np.arange(1,13)  # tableau 1d à 12 éléments
>>> x
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> y = np.reshape(x,(3,4))  # ou y = x.reshape((3,4))
>>> y
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> y[1,2] = 72   # l'objet y partage ses valeurs avec x
>>> x
array([ 1,  2,  3,  4,  5,  6, 72,  8,  9, 10, 11, 12])
```

Les tableaux numpy XII

Cet exemple met en exergue deux spécificités des tableaux numpy :

- pour économiser la mémoire, certaines fonctions/méthodes vont créer, certes de nouveaux tableaux, mais les valeurs sont partagées avec celles du tableau initial ; en général ce n'est pas génant mais il faut absolument avoir en tête ce principe et, si besoin, utiliser la méthode copy si ce n'est pas l'effet escompté ;

```
>>> x = np.arange(1,13)  # on recrée le tableau x
>>> y = np.reshape(x.copy(),(3,4))
>>> # ou bien y = x.copy().reshape((3,4))
>>> y[1,2] = 72
>>> x  # cette fois plus de surprise
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Les tableaux numpy XIII

- le nouveau tableau est créé ligne par ligne : on “remplit” d'abord la première ligne puis la 2 ème et la troisième, c'est ce qu'on appelle l'ordre C (du langage C). Il est possible d'obtenir l'ordre fortran (on remplit la première colonne, puis la 2 ème, etc.) avec l'argument optionnel nommé `order='F'` :

```
>>> z = x.reshape((3,4), order='F')
>>> z
array([[ 1,  4,  7, 10],
       [ 2,  5,  8, 11],
       [ 3,  6,  9, 12]])
```

- pour en savoir plus sur un tableau numpy, on peut utiliser `.flags` :

Les tableaux numpy XIV

```
>>> z.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  # .... d'autres informations ...
```

Les tableaux numpy XV

Rmq : le champ `OWNDATA` n'est pas si explicite ! Il peut être positionné sur False alors que le tableau en question ne partage pas forcément ses valeurs avec un autre tableau et inversement, s'il est positionné sur True, il peut aussi partager ses valeurs avec un autre tableau. Le juge de paix (savoir si deux tableaux T_1 et T_2 partagent une partie de la mémoire) semble être la fonction :

$$\texttt{np.may\_share\_memory}(T_1, T_2).$$

Comme il a été dit précédemment les tableaux numpy sont des objets compliqués quant à leur fonctionnement interne. En pratique cependant ils ne posent généralement pas de problème.

Les tableaux numpy XVI

Référencer, adresser les éléments d'un tableau numpy

Prenons un tableau 2d T, référencer l'élément en position (i, j), soit $T_{i,j}$ s'obtient avec la syntaxe `T[i,j]`.

> *Avec les versions récentes de numpy il semble que la syntaxe `T[i][j]` fonctionne toujours (ce n'était pas le cas auparavent en particulier lorsque le tableau avait l'ordre Fortran) mais il est* **fortement recommandé d'utiliser toujours la première syntaxe pour des raisons d'efficacité** *(la deuxième crée un tableau temporaire cf `https://numpy.org/devdocs/user/basics.indexing.html`).*

Idem pour un tableau 3d, référencer $T_{i,j,k}$ s'obtient avec `T[i,j,k]` (`T[i][j][k]` doit fonctionner mais est de moins en moins efficace du fait de la création de 2 tableaux temporaires).

Les tableaux numpy XVII

Il est possible (et fortement conseillé pour vectoriser en cas de besoin) de référencer toutes sortes de sous-tableaux (voir exemples à la suite) :

- pour référencer toute la ligne i (ce qu'on note T_i dans le cours) `T[i,:]`, pour référencer toute la colonne j (noté T^j dans le cours) `T[:,j]` ;
- en utilisant les tranches python : par exemple `T[::2,1::2]` va désigner la sous-matrice formée des lignes d'indice pair $(0, 2, 4, ...)$ et des colonnes d'indice impair $(1, 3, ...)$. Autre exemple sur un tableau 3d, si on veut référencer les m_1 premières lignes, les n_1 premières colonnes et tous les étages d'indice multiple de 3 $(0, 3, 6, 9, ...)$, on utilisera : `T[:m1,:n1,::3]`
- on peut aussi utiliser des tableaux de booléens ;

Les tableaux numpy XVIII

- on peut aussi utiliser des listes d'indices ou des tableaux d'indices (cad dont les valeurs sont entières et sont comprises dans le profil du tableau référencé) ; il y a cependant quelques différences avec les tranches :
  - si `T` est un tableau 2d et `ii` et `jj` des listes ou des tableaux 1d numpy alors `T[ii,jj]` ne sera pas le tableau 2d obtenu par l'intersection des lignes `ii` et des colonnes `jj` (comme avec des tranches) mais le tableau 1d formé par les éléments `T[ii[0],jj[0]]`, `T[ii[1],jj[1]]`,... (dans ce cas `ii` et `jj` doivent avoir le nombre d'éléments) ;
  - on peut retrouver le comportement obtenu avec les tranches en transformant `ii` en tableau 2d de profil $(len(ii), 1)$;
  - autre exemple si `T` est un tableau 1d et `ij` un tableau 2d (m, n) d'entiers tous compris entre 0 et $T.size$ alors `U = T[ij]` est un tableau 2d de profil (m, n) tel que `U[i,j] = T[ij[i,j]]` ;

Les tableaux numpy XIX

- pour résumer : *le profil du tableau résultat dépend des profils du ou des tableaux d'indices !*

Voici maintenant les exemples :

```
>>> T = np.arange(0,1,0.05).reshape((5,4))
>>> T   # un tableau pour travailler dessus...
array([[0.  , 0.05, 0.1 , 0.15],
       [0.2 , 0.25, 0.3 , 0.35],
       [0.4 , 0.45, 0.5 , 0.55],
       [0.6 , 0.65, 0.7 , 0.75],
       [0.8 , 0.85, 0.9 , 0.95]])
>>> T[0,:]  # extraction de la première ligne
array([0.  , 0.05, 0.1 , 0.15])
>>> T[0,:] = np.ones(4)  # assignation
>>> T
array([[1.  , 1.  , 1.  , 1.  ],
       [0.2 , 0.25, 0.3 , 0.35],
       [0.4 , 0.45, 0.5 , 0.55],
```

Les tableaux numpy XX

```
       [0.6 , 0.65, 0.7 , 0.75],
       [0.8 , 0.85, 0.9 , 0.95]])
>>> T[0,:] = 2  # valable aussi
>>> T
array([[2.  , 2.  , 2.  , 2.  ],
       [0.2 , 0.25, 0.3 , 0.35],
       [0.4 , 0.45, 0.5 , 0.55],
       [0.6 , 0.65, 0.7 , 0.75],
       [0.8 , 0.85, 0.9 , 0.95]])
>>> T[:3,[0,2,3]] # lignes 0,1,2 et colonnes 0,2 et 3
array([[2.  , 2.  , 2.  ],
       [0.2 , 0.3 , 0.35],
       [0.4 , 0.5 , 0.55]])
>>> # lignes 0, 4 (booleen) et cols 0, 1, 2 (tranche)
>>> T[[True,False,False,False,True],:-1]
array([[2.  , 2.  , 2.  ],
       [0.8 , 0.85, 0.9 ]])
>>> # avec des listes ou tableaux d'indices
```

Les tableaux numpy XXI

```
>>> # un autre tableau de travail
>>> T = np.arange(20).reshape((5,4))
>>> T
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> T[[1,2,4],[0,1,3]] # résultat T[1,0],T[2,1],T[4,3]
array([ 4,  9, 19])
>>> # transformation de la liste en tableau (3,1) :
>>> ii = np.array([1,2,4]).reshape((-1,1))
>>> T[ii,[0,1,3]]  # tab. des lignes 1, 2, 4 et col. 0, 1, 3 de T
array([[ 4,  5,  7],
       [ 8,  9, 11],
       [16, 17, 19]])
```

Les tableaux numpy XXII

L'indiçage avec des booléens qui sélectionne les indices "vrais" permettra de vectoriser des tests, exemple :

```
>>> x = np.arange(6)
>>> x
array([0, 1, 2, 3, 4, 5])
>>> x[x<3] = 0.1
>>> x    # voir explications ci-dessous
array([0, 0, 0, 3, 4, 5])
```

Nous verrons ultérieurement que l'opérateur < est une ufunc binaire et que le "broadcasting" élémentaire s'applique dans l'expression `x<3`. Noter au passage que, le tableau `x` étant un tableau d'entiers, l'assignation à 0.1 des 3 premières composantes opère d'abord un "cast" sur le "flottant" 0.1 (cad considère `int(0.1)`).

1 Les tableaux numpy

2 Exemples de vectorisation avec numpy

Vectorisation en numpy I

Elle consiste essentiellement à éviter les boucles python (ou en réduire fortement leur nombre) en utilisant un arsenal de fonctions/opérateurs/syntaxes qui vont agir directement sur les tableaux ou des sous-tableaux sans avoir besoin d'écrire explicitement une ou des boucles. Il n'y a pas de magie : les boucles sont bien exécutées mais à plus bas niveau sans faire appel à l'interprète python.

Quelques éléments (fonctions/opérateurs/syntaxes) :

- la généralisation des calculs sur les scalaires et entre deux scalaires à l'aide des *ufuncs unaires et binaires* et des raccourcis appelés *"broadcasting"* permettant d'effectuer des opérations utiles entre tableaux de tailles différentes ; dans ce registre il semble que numpy ait été plus loin que Matlab ; cette partie est détaillée ci-après ;

Vectorisation en numpy II

- des fonctions utiles comme `np.sum`, `np.prod`, `np.mean`, `np.std`, `np.cumsum`, `np.cumprod`, etc (nous en verrons unes à la fin des exemples de “broadcasting”) ;
- opérations d’algèbres linéaires (produit de matrices, produit scalaire, ...) ;
- et grâce aux diverses syntaxes d’adressage (cf pages précédentes), pouvoir faire ces mêmes opérations sur des sous-tableaux.

Vectorisation en numpy III

Fonctions “universelles” (ufuncs)

Les fonctions mathématiques classiques prenant un nombre et renvoyant un nombre (cos,sin,exp,..., moins “unaire”,...) ou prenant deux nombres et renvoyant un nombre (addition, soustraction, multiplication,...), sont définies par Numpy en tant que *ufuncs* ce qui permet de les utiliser sur les tableaux numpy.

```
>>> type(np.exp)
<class 'numpy.ufunc'>
>>> type(np.multiply)  # fct appelée pour l'opérateur *
<class 'numpy.ufunc'>
>>> x = np.arange(3)   # tableau [0,1,2]
>>> np.exp(x)  # fct appliquée élément par élément
array([1.        , 2.71828183, 7.3890561 ])
>>> x * x  # multiplication élément par élément
array([0, 1, 4])
```

Vectorisation en numpy IV

Comportement des ufuncs unaires sur un tableau numpy

Comme dans l'exemple précédent avec `exp` si *func* est une ufunc numpy et *x* un tableau numpy (numpy.ndarray) alors *func(x)* retourne un tableau de même profil que celui de `x` où la fonction a été appliquée "élément par élément" [a].

Bonus : les ufuncs s'appliquent aussi sur un simple scalaire : **inutile** d'importer math dès que vous utilisez numpy.

---

a. "element-wise" en anglais

Exemples :

```
>>> x = np.linspace(0,2*np.pi,80)
>>> y = np.sin(x)    # on a y[i] = sin(x[i]), i=0,..,79
>>> np.sin(np.pi/2) # fonctionne aussi sur un scalaire
1.0
```

Vectorisation en numpy V

```
>>> import math as m
>>> m.cos(m.pi)  # ça marche
-1.0
>>> np.cos(np.pi) # ça marche aussi
-1.0
>>> x = np.array([0.5, 8]) # un tableau
>>> np.tan(x)  # ça marche !
array([ 0.54630249, -6.79971146])
>>> m.tan(x)  # ça ne va pas marcher
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only size-1 arrays can be converted to Python scalars
```

Rmq (encore) : **inutile** d'utiliser le module math dès que vous devez importer numpy.

Vectorisation en numpy VI

comportement des ufuncs binaires sur les tableaux numpy

Le plus souvent il s'agit du comportement des tableaux numpy avec les opérateurs binaires arithmétiques usuels comme `+,-,*,/,**` et donc s'appliquant sur une expression du type *x op y* où au moins l'un des deux opérandes *x* ou *y* est un tableau numpy et *op* un opérateur usuel. De manière interne cela correspond à un appel du type *func_op(x,y)* où *func_op* est la fonction implémentant *op*.

Le comportement de base correspond au cas où *x* et *y* sont tous les deux des tableaux numpy (numpy.ndarray) de même profil [a] et alors *func_op(x,y)* retourne un tableau de même profil que celui de ses opérandes où l'opération a été appliquée "élément par élément".

---

a. même nombre de dimensions (1,2,3,..) et même nombre d'éléments dans chaque dimension.

Vectorisation en numpy VII

“broadcasting élémentaire”

Certains cas où les deux tableaux `x` et `y` n'ont pas le même profil sont licites, en particulier si l'un d'eux est un scalaire, tout fonctionne comme s'il était un tableau constant de même profil que l'autre.

Exemples :

```
>>> x = np.arange(4)
>>> x
array([0, 1, 2, 3])
>>> x - 2.5
array([-2.5, -1.5, -0.5,  0.5])
>>> x*2
array([0, 2, 4, 6])
>>> x / 3
array([0.        , 0.33333333, 0.66666667, 1.        ])
```

Vectorisation en numpy VIII

Autre exemple : la fonction de Runge

```
def runge(x) :
    return 1/(1 + x**2)
```

évaluée sur un tableau numpy contient 3 raccourcis de ce type :

1. l'opération `x**2` du type *tableau* ** *scalaire* (on note `temp` le tableau temporaire contenant le résultat) ;
2. puis `1 + temp` du type *scalaire* + *tableau* (on note `temp_bis` le tableau temporaire contenant le résultat) ;
3. et enfin `1/temp_bis` du type *scalaire* / *tableau*.

Rmq : on remarque (du fait de la création de tableaux temporaires) que la vectorisation, si elle permet d'accélérer fortement un code (voir page suivante), consomme de la mémoire.

Vectorisation en numpy IX

Autre exemple : la fonction $\varphi(x) = \frac{-2x}{(x-1)(x+1)}$.

```
def phi(x) :
    return -2*x/((x-1)*(x+1))
```

Accélération obtenue (cf script `timing_vec_fct.py` disponible sur Arche) :

n	éval scalaire	éval vec	accélération
10	$9.04\,10^{-6}$	$3.05\,10^{-6}$	2.96
100	$8.27\,10^{-5}$	$3.18e\,10^{-6}$	26
1000	0.00082	$6.18\,10^{-6}$	133
10000	0.00816	$4.36\,10^{-5}$	187
100000	0.0826	0.000837	98.6

Question : combiens de "raccourcis" (broadcasting) comporte l'évaluation de `phi(x)` si `x` est un tableau ?

Vectorisation en numpy X

“Broadcasting” règle générale

Considérons 2 tableaux numpy (ndarray) T et U de profils différents :

1. si T et U n'ont pas le même nombre de dimensions, le profil du tableau en ayant le moins est complété par des 1 à gauche ;
2. pour les 2 tableaux, pour chaque dimension on regarde le nombre d'éléments : si c'est le même c'est bon, s'ils sont différents mais que l'un est égal à 1, c'est bon aussi, dans le cas contraire il n'y a pas de broadcasting et une erreur est levée ;
3. les dimensions de chaque tableau qui, en nombre d'éléments, sont passées de 1 à n (cad le nombre d'éléments de la dimension correspondante de l'autre tableau) sont complétées par n copies de l'élément correspondant (rmq : il ne doit pas y avoir création d'un nouveau tableau mais “tout se passe comme si”.).

Vectorisation en numpy XI

exemple de conformité de deux tableaux : si le profil de T est $(2, 3, 5)$ et celui de U est $(3, 5)$ alors :

1. le profil de U devient $(1, 3, 5)$ (on ajoute une dimension à U "devant" ses dimensions initiales) ;
2. le scan sur les dimensions successives nous dit que le "broadcasting" est valable (seul le nombre d'éléments dans la première dimension est différent mais celui de U complété est égal à 1) ;
3. pour l'opération entre les deux tableaux (qu'on a pas spécifiée ici mais cela pourrait être n'importe quelle ufunc binaire `T < U`, `T + U`, ...) on considère que le tableau U complété (noté ci-dessous par $\tilde{U}$) est obtenu avec le tableau U initial selon :

$$\tilde{U}[0, i, j] = \tilde{U}[1, i, j] = U[i, j], \forall (i, j) \in [\![0, 2]\!] \times \in [\![0, 4]\!]$$

Vectorisation en numpy XII

Un exemple classique : soit T un tableau 2d de profil (m, n) et c un tableau 1d de profil (n). Que fait `T * c` (ou `c * T`) ? La règle de broadcasting nous dit que l'opération est valide et correspond à l'opération `T * cc` où :

$$cc[i, j] = c[j], \forall (i, j) \in [\![0, m-1]\!] \times \in [\![0, n-1]\!]$$

Le tableau `cc` correspond à un tableau à m lignes identiques, ces lignes étant constituées des n éléments de `c`.

Résultat : pour $j = 0, \ldots, n-1$, la colonne j de T est multipliée par $c[j]$.

Rmq : `T/c`, `T-c`, `T**c`, `T<c`, etc. sont toutes des opérations valables. Chacune retourne un tableau de même profil que T dont l'élément en position (i, j) est respectivement égal à $T[i, j]/c[j]$, $T[i, j] - c[j]$, $T[i, j]^{c[j]}$, $T[i, j] < c[j]$ (un booléen).

Vectorisation en numpy XIII

Autre exemple classique : reprenons notre tableau T de profil (m, n) mais on voudrait maintenant faire des opérations sur les lignes de T (multiplier ou diviser ou additionner ou soustraire..., chaque ligne de T par un scalaire différent, ces m scalaires étant dans un tableau c donc de profil (m), comment si prendre ?

Réponse : il faut transformer c en tableau 2d de profil $(m, 1)$. On peut utiliser `reshape` avec `c.reshape((m,1))` ou `np.reshape(c,(m,1))`.

Exercice : appliquer les règles de broadcasting et vérifier que cela fonctionne (voir aussi les exemples qui suivent) !

Vectorisation en numpy XIV

Exemple : multiplication de chaque ligne d'une matrice par un scalaire différent. À tester directement à la console !

```
>>> A = np.arange(1,13).reshape((3,4))
```

$$A \text{ est le tableau 2d : } \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

```
>>> v = np.array([0.5, 2, 3]).reshape((3,1))
>>> v*A
array([[ 0.5,  1. ,  1.5,  2. ],
       [10. , 12. , 14. , 16. ],
       [27. , 30. , 33. , 36. ]])
```

La 1 ère ligne de A est multipliée par 0.5, la 2 ème par 2 et la 3 ème par 3.

Vectorisation en numpy XV

Exemple : additionner chaque ligne d'un tableau par un scalaire différent.

On reprend les mêmes tableaux que dans l'exemple précédent :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad v = \begin{bmatrix} 0.5 \\ 2 \\ 3 \end{bmatrix} \quad \text{de profil } (3, 1)$$

```
>>> v + A
array([[ 1.5,  2.5,  3.5,  4.5],
       [ 7. ,  8. ,  9. , 10. ],
       [12. , 13. , 14. , 15. ]])
```

On a bien ajouté 0.5 à la première ligne, 2 à la deuxième et 3 à la troisième !

Vectorisation en numpy XVI

Exemple : diviser chaque colonne d'un tableau par un scalaire différent.

On reprend le tableau A précédent (4 colonnes) :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

et on va diviser la première colonne par 1, la deuxième par 2, la troisième par 3 et la quatrième par 4 :

```
>>> w = np.arange(1,5)    # création du tableau 1d (1,2,3,4)
>>> A / w
array([[1.        , 1.        , 1.        , 1.        ],
       [5.        , 3.        , 2.33333333, 2.        ],
       [9.        , 5.        , 3.66666667, 3.        ]])
```

Vectorisation en numpy XVII

Exercice

On dispose d'un tableau numérique X de taille $n \times p$, n est le nombre d'individus p le nombre de caractéristiques (numériques) par individus. Comment créer le tableau centré et réduit avec numpy ?

Aide :

- Pour calculer la moyenne et l'écart type de chaque colonne, on pourra utiliser les fonctions numpy `mean` et `std`.
- Dans ces fonctions (tout comme dans les fonctions `sum`, `prod`, `cumsum`, `cumprod`) il faudra utiliser l'argument optionnel `axis=0` pour que l'opération se fasse dans la direction de la première dimension (0 dans les "langages à indice 0").

Vectorisation en numpy XVIII

Une solution :

`X` doit être un tableau 2d !

```
mu = np.mean(X, axis=0)
sigma = np.std(X, axis=0, ddof=1)
Xcr = (X - mu)/sigma
```

Oui, c'est compact et lisible ! Ce n'est pas toujours le cas... L'argument ddof=1 dans la fonction std permet de diviser par $n-1$ plutôt que par n.

Vectorisation en numpy XIX

Un exemple un peu plus difficile (mais pas tant que ça) :
vectoriser Lagrange (la basique en $O(n^2)$) !

```
def lagrange_bis(t,x,y):
    p = np.zeros_like(t)
    n = len(x)
    for i in range(n):
        # calcul du i eme monôme L_i(t)
        Li = np.ones_like(t)
        for j in range(n):
            if j != i:
                Li = Li*((t-x[j])/(x[i]-x[j]))
        p = p + y[i]*Li
    return p
```

Explications données oralement... Plus essais pour voir l'apport en vitesse d'éxécution.

Vectorisation en numpy XX

Vectorisation des tests

Vectorisation d'une fonction définie par morceaux.

Exemple : une fonction "créneau" $f(x) = 2$ pour $x \in [-1, 0.5]$ et $f(x) = 0$ sinon.

```
def creneau_vect(x) :
    y = np.zeros_like(x)
    inside = np.logical_and(-1 <= x, x <= 0.5)
    y[inside] = 2
    return y
```

Elle utilise :

- l'indiçage par tableau de booléens ;
- encore des raccourcis de broadcasting.

Rmq : les opérateurs `and`, `or`, `not` n'ont pas été promus comme *ufuncs*, il faut utiliser `logical_and`, `logical_or`, `logical_not`.

Vectorisation en numpy XXI

Multiplication matricielle : on l'obtient avec l'opérateur `@`. Si x et z sont des tableaux 1d de profil (n), y un tableau 1d de profil (m) et A un tableau 2d de profil (m, n) alors :

- `x@z` effectue le produit scalaire $< x, z >= (x|z) = \sum_k x_k z_k$ (on peut aussi utiliser `np.sum(x*z)` mais cela doit être un peu moins rapide) ;
- `A@x` effectue le produit matriciel Ax (x considéré comme un vecteur colonne dans l'écriture mathématique) ;
- `y@A` effectue le produit matriciel $y^\top A$ (idem y considéré comme un vecteur colonne dans l'écriture mathématique).

Rmq : si on veut créer la matrice $B := y^\top x$ de profil (m, n), $(b_{i,j} = y_i x_j)$, plutôt que d'utiliser `y.reshape((m,1)) @ x.reshape((1,n))`, on peut utiliser `np.outer(y,x)`.