Module C2

Manipulation de matrices et vecteurs avec Python 3

Timothée Duruisseau

Department de génie mécanique
École Polytechnique Montréal
Canada
6 septembre 2021

Objectifs du module

À la fin du module, il sera important de savoir manipuler les vecteurs et les matrices. À cette fin, il est important de savoir indexer adéquatement un objet `numpy`, transformer correctement et savoir faire de l'algèbre linéaire avec Python.

Table des matières

Chapitre 1

Chapitre 1 : Indexation d'un objet `numpy`

1.1 Introduction

Dans ce chapitre, l'indexation adéquate sera démontrée pour des objets `numpy`. L'indexation normal d'un objet Python, tel que les listes et les tuples, se fait de manière imbriquée. Cela signifie qu'il faut indiquer l'index entre crochets pour chaque niveau, séquenciellement. Pour les objets `numpy`, l'indexation se fait directement. Les prochaines sections expliqueront exactement ce que cela veut dire.

1.2 Indexation dans un vecteur

Commençons simplement avec un vecteur, qui est un objet possédant une seule dimension. La manière la plus simple d'indexer est d'écrire l'index entre crochets à côté du nom de l'objet. Le script 1 démontre cela.

Script 1
Exemple simple d'indexation d'un vecteur

```
1 import numpy as np
2
3 x = np.array([1,2,3,4,5,6])
4
5 print('Valeur de x à l'index 2 : %d' % x[2])
```

```
Valeur de x à l'index 2 : 3
```

L'indexation dans un vecteur possède les mêmes propriétés que l'indexation dans une liste ou un tuple. Ainsi, l'indexation négative implique un décompte à rebours. Il est possible d'indexer plusieurs valeurs à la fois en faisant des tranches, grâce à l'opérateur deux-points `:`. Son utilisation se fait ainsi : `depart:fin:pas`. Chaque partie est optionnelle, selon les besoins. La valeur par défaut de `depart` est 0. La valeur par défaut de `fin` est la fin du vecteur. La valeur par défaut de `pas` est 1. Si la valeur par

défaut est souhaité, il suffit de ne rien inscrire. Par exemple, si on souhaite prendre toutes les valeurs du vecteur en tranchant, il suffit d'écrire `x[:]` ou `x[::]`.

Pour prendre toutes les valeurs à rebours, le pas peut être mis négatif : `x[::-1]`. Le pas peut prendre n'importe quelle valeur, peu importe la grosseur du vecteur. La valeur de `depart` est toujours la première valeur à être sélectionnée. La valeur de `fin` est toujours exclue. Il faut donc prévoir la tranche pour qu'elle sélectionne bien toutes les valeurs désirées.

1.3 Indexation dans les matrices et tenseurs

L'indexation dans les matrices et les tenseurs se fait essentiellement de la même manière que pour les vecteurs. Les mêmes règles de tranche s'appliquent également. La différence provient des dimensions supplémentaires. En effet, un vecteur est une série de valeur contenue dans une seule dimension. Une matrice contient 2 dimensions dans lesquelles les valeurs sont inscrites. Un tenseur contient 3 ou plus dimensions dans lesquelles les valeurs sont stockées.

L'accès aux dimensions supérieures se fait en ajoutant les indices, ou tranches, séparés par des virgules. Ainsi, pour aller chercher la valeur se trouvant dans la 3e colonne de la 2e ligne, la commande serait `x[1,2]`. Il y a deux détails à bien faire attention. D'abord, l'indexation commence à 0 avec Python et donc, la 2e ligne possède l'indice 1. Ensuite, l'indexation des dimensions se fait selon l'ordre suivant : ligne, colonne. Chaque dimension supplémentaire s'inscrit à la suite de la précédente.

SCRIPT 2
Exemple d'indexation et de tranches dans une matrice

```
1 import numpy as np
2
3 x = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20]])
4
5 print('4e ligne indexé, colonnes 2 et 3')
6 print(x[3,2:4])
7
8 print('Ligne 2 et 3, colonnes 2 et 3')
9 print(x[1:3,2:4])
```

```
[4e ligne indexé, colonnes 2 et 3
[18 19]
Ligne 2 et 3, colonnes 2 et 3
[[ 8  9]
 [13 14]]
```

Chapitre 2

Chapitre 2 : Transformation d'un objet `numpy`

2.1 Introduction

Les vecteurs et les matrices doivent pouvoir être manipulées. Cependant, les manipulations sont multiples. Les manipulations peuvent être de nature mathématiques ou géométriques. Les sections qui suivent vont expliquer comment faire les différents types de manipulations.

2.2 Opérations mathématiques élément par élément

Les opérations mathématiques élément par élément font l'opération demandée en ne considérant que 2 éléments à la fois. Par exemple, prenons 2 vecteurs, `x = np.array([1,2,3])` et `y = np.array([4,5,6])`. En effectuant la commande `x + y`, le résultat obtenu est `array([5, 7, 9])`, soit un vecteur avec la somme des éléments d'indice 0, d'indice 1 et d'indice 2, séparément. De la même manière, la commande `x*y` retournera le produit `array([4, 10, 18])`. Le script 3 démontre chaque opérateur entre vecteurs.

Afin que ces opérations puissent être exécutées, il faut que les dimensions soient diffusables, *broadcastable* en anglais. Cela veut dire que les 2 vecteurs doivent avoir le même nombre de valeurs. Les opérations peuvent également être faite entre deux matrices partageant les mêmes dimensions ou entre une matrice et un vecteur. Dans le cas d'opérations entre une matrice et un vecteur, le vecteur sera considéré comme étant un vecteur ligne par défaut. Cette affirmation n'est valide que pour les opérations élément par élément. Pour que le vecteur soit considéré comme une colonne, il faut explicitement spécifier l'opération souhaitée. Le script 4 démontre quelques exemples d'opérations entre une matrice et un vecteur, et le script 5 montre la sortie de cette démonstration.

SCRIPT 3
Démonstration de différents opérateurs mathématiques entre vecteurs

```python
import numpy as np

# Vecteurs
x = np.array([1,2,3])
y = np.array([4,5,6])

# Addition
print('Addition')
print(x+y)

# Soustraction
print('Soustraction')
print(x-y)

# Multiplication
print('Multiplication')
print(x*y)

# Division
print('Division')
print(x/y)

# Puissance
print('Puissance')
print(x**y)
```

```
Addition
[5 7 9]
Soustraction
[-3 -3 -3]
Multiplication
[ 4 10 18]
Division
[0.25 0.4 0.5 ]
Puissance
[  1  32 729]
```

SCRIPT 4
Démonstration de différent opérateurs mathématiques entre un vecteur et une matrice

```
import numpy as np

# Vecteur
x = np.array([1,2,3])
# Matrice
y = np.array([[ 4, 5, 6],
              [ 7, 8, 9],
              [10,11,12]])

# Addition
print('Addition en ligne')
print(x+y)
print('Addition en colonne')
# La méthode .T prend la transposée de la matrice
print((x+y.T).T)

# Soustraction
print('Soustraction en ligne')
print(x-y)
print('Soustraction en colonne')
print((x-y.T).T)

# Multiplication
print('Multiplication en ligne')
print(x*y)
print('Multiplication en colonne')
print((x*y.T).T)

# Division
print('Division en ligne, vecteur divisé par matrice')
print(x/y)
print('Division en colonne, matrice divisé par vecteur')
print((y.T/x).T)

# Puissance
print('Puissance en ligne, vecteur puissance matrice')
print(x**y)
print('Puissance en colonne, matrice puissance vecteur')
print((y.T**x).T)
```

SCRIPT 5
Résultats du script 4

```
Addition en ligne
[[ 5  7  9]
 [ 8 10 12]
 [11 13 15]]
Addition en colonne
[[ 5  6  7]
 [ 9 10 11]
 [13 14 15]]
Soustraction en ligne
[[-3 -3 -3]
 [-6 -6 -6]
 [-9 -9 -9]]
Soustraction en colonne
[[-3 -4 -5]
 [-5 -6 -7]
 [-7 -8 -9]]
Multiplication en ligne
[[ 4 10 18]
 [ 7 16 27]
 [10 22 36]]
Multiplication en colonne
[[ 4  5  6]
 [14 16 18]
 [30 33 36]]
Division en ligne, vecteur divisé par matrice
[[0.25      0.4       0.5      ]
 [0.14285714 0.25     0.33333333]
 [0.1       0.18181818 0.25    ]]
Division en colonne, matrice divisé par vecteur
[[4.        5.        6.       ]
 [3.5       4.        4.5      ]
 [3.33333333 3.66666667 4.     ]]
Puissance en ligne, vecteur puissance matrice
[[    1     32    729]
 [    1    256  19683]
 [    1   2048 531441]]
Puissance en colonne, matrice puissance vecteur
[[   4    5    6]
 [  49   64   81]
 [1000 1331 1728]]
```

2.3 Manipulations et fonctions d'algèbre linéaire

Afin de bien pouvoir faire les opérations d'algèbre linéaire, il est important de savoir comment appeler les bonnes fonctions et comment faire les modifications nécessaires. Le script 6 démontre les fonctions et les résultats obtenus sont au script 7. Les fonctions et méthodes utilisées sont celles venant avec le module `numpy`.

Il est à remarquer que l'ordre des arguments à une importance. Par exemple, la fonction de produit scalaire `np.dot()` ne retourne pas la même chose selon que la matrice soit en premier ou en deuxième. Cela suit évidemment la logique de l'algèbre linéaire. De plus, lorsque des opérations d'algèbres linéaires sont faites, les vecteurs sont considérés de la forme logique pour l'opération. Cela veut dire que le vecteur est considéré à la fois comme un vecteur ligne et un vecteur colonne.

Il est possible de forcer une interprétation spécifique d'un vecteur. Pour ce faire, il suffit de créer une matrice ne possédant qu'une seule ligne ou une seule colonne, selon ce qui est désiré. Cela impliquera que le vecteur sera considéré comme possédant deux dimensions, 2 *axis* en anglais.

Il est également possible d'interchanger les lignes et colonnes en effectuant un produit scalaire avec une matrice identité possédant la commutation souhaitée. Le script 8 démontre la commutation de lignes et de colonnes en utilisant la même matrice de modification. Les dimensions des matrices doivent êtres diffusables.

SCRIPT 6
Déomnstration de différentes fonctions d'algèbre linéaire

```
import numpy as np

# Vecteur
x = np.array([1,2,3])
y = np.array([4,5,6])
# Matrice
A = np.array([[13,14,15],
              [ 7, 8, 9],
              [10,11,12]])
B = np.array([[1,0,5],
              [2,1,6],
              [3,4,0]])

# Produit scalaire
print("Produit scalaire A.x")
print(np.dot(A,x))
print("Produit scalaire x.A")
print(np.dot(x,A))

# Produit vectoriel
print("Produit vectoriel")
print(np.cross(A,y))

# Valeurs et vecteurs propres
z = np.linalg.eig(A) # La fonction retourne un tuple, il faut donc indexer
print("Valeurs propres de A")
print(z[0])
print("Vecteurs propres de A")
print(z[1])

# Normes
print("Norme de A")
print(np.linalg.norm(A))

# Déterminant
print("Déterminant de A")
print(np.linalg.det(A))

# Trace, la somme sur la diagonale
print("La trace sur la diagonale principale de A")
print(np.trace(A))
print("La trace sur la diagonale supérieure de A")
print(np.trace(A,1))
print("La trace sur la diagonale inférieure de A")
print(np.trace(A,-1))

# Les matrices inverses et pseudo-inverses
print("La pseudo-inverse de A")
print(np.linalg.pinv(A))
print("L'inverse de B")
print(np.linalg.inv(B))
print("Multiplication de B par son inverse")
print(np.dot(B,np.linalg.inv(B)))
```

SCRIPT 7
Résultats du script 6

```
Produit scalaire A.x
[86 50 68]
Produit scalaire x.A
[57 63 69]
Produit vectoriel
[[  9 -18   9]
 [  3  -6   3]
 [  6 -12   6]]
Valeurs propres de A
[3.27249807e+01 2.75019260e-01 1.17819126e-16]
Vecteurs propres de A
[[-0.71824706 -0.72820453 0.40824829]
 [-0.40839797 0.68501966 -0.81649658]
 [-0.56332252 -0.02159244 0.40824829]]
Norme de A
33.896902513356586
Déterminant de A
0.0
La trace sur la diagonale principale de A
33
La trace sur la diagonale supérieure de A
23
La trace sur la diagonale inférieure de A
18
La pseudo-inverse de A
[[ 8.05555556e-01 -1.13888889e+00 -1.66666667e-01]
 [ 5.55555556e-02 -5.55555556e-02 3.68450671e-16]
 [-6.94444444e-01 1.02777778e+00 1.66666667e-01]]
L'inverse de B
[[-24. 20.  -5.]
 [ 18. -15.  4.]
 [  5.  -4.  1.]]
Multiplication de B par son inverse
[[ 1.00000000e+00 0.00000000e+00 0.00000000e+00]
 [-3.55271368e-15 1.00000000e+00 -8.88178420e-16]
 [ 0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

SCRIPT 8
Exemple de commutation de lignes et de colonnes

```
import numpy as np

# Matrice
B = np.array([[1,0,5],
              [2,1,6],
              [3,4,0]])
print("La matrice B de départ")
print(B)

# Matrice de changement
w = np.array([[1,0,0],
              [0,0,1],
              [0,1,0]])

# Interchanger les colonnes 2 et 3
print("La matrice B suite à la commutation des colonnes 2 et 3")
print(np.dot(B,w))

# Interchanger les lignes 2 et 3
print("La matrice B suite à la commutation des lignes 2 et 3")
print(np.dot(w,B))
```

```
La matrice B de départ
[[1 0 5]
 [2 1 6]
 [3 4 0]]
La matrice B suite à la commutation des colonnes 2 et 3
[[1 5 0]
 [2 6 1]
 [3 0 4]]
La matrice B suite à la commutation des lignes 2 et 3
[[1 0 5]
 [3 4 0]
 [2 1 6]]
```

2.4 Manipulations géométriques

Parfois, il est nécessaire de faire des manipulations géométriques sur les matrices et vecteurs. Cela signifie de changer les dimensions et l'orientation de la matrice ou du vecteur. La première transformation géométrique à savoir est la transposée.

Chapitre 3

Chapitre 3 : Algèbre linéaire

3.1 Introduction

Il est important de savoir faire de l'algèbre linéaire pour faire de la résolution de problème de manière efficace. À ce titre, `numpy` possède un sous-module tout indiqué, `linalg`. Dans ce chapitre, nous verrons comment bâtir des matrices, obtenir leur inverse et faire des multiplications matricielles. Ensuite, ces notions seront utilisées afin de faire des rotations et des translations, ainsi que des mises à l'échelle.

3.2 Bâtir une matrice

Depuis le début de cet ouvrage, les matrices ont été traité comme étant des objets `ndarray`. Il existe un objet `numpy` spécifiquement pour les matrices en 2 dimensions, appelé `matrix`. Ce type ne sera pas abordé car il est plus contraignant et n'a pas un comportement aussi régulier. Ainsi, toutes les matrices discutées seront de type `ndarray` et les opérations effectuées seront en fonction de cela.

Ainsi, il existe plusieurs manières de construire une matrice. Le plus simple est de déclarer un `array` possédant 2 dimensions : `a = np.array([[1,2],[3,4]])`. Chaque ligne est contenue entre 2 crochets et l'ensemble des lignes est contenu entre 2 crochets. Chaque valeur doit être entrée manuellement en utilisant cette manière. Bien que très pratique, cette manière n'est pas idéale dans quelques situations. La création de matrices spéciales, tel que la matrice identitée par exemple, peut se faire grâce à des fonctions spécifiques.

La matrice identitée se fait grâce à la fonction `np.eye(N,M,k)`. Par défaut, si un seul argument est donné en entrée, la matrice identitée retournée sera une matrice carrée avec la diagonale centrale composée de 1. Si la matrice souhaitée est rectangulaire, il faut alors spécifié le nombre de lignes, `N`, et le nombre de colonnes, `M`. De plus, si la diagonale de 1 souhaitée n'est pas centrée, c'est-à-dire qu'elle commence à (0,0), il faut alors spécifié l'argument `k`. Une valeur positive déplacera la diagonale vers le haut, une valeur négative déplacera la diagonale vers le bas.

Il est possible de faire des matrices remplies uniquement de 1 ou de 0. Une matrice de 1 se fait avec la fonction `np.ones([N,M])`. Le nombre de lignes est indiqué par `N` et le nombre de colonnes par `M`. Ces deux valeurs doivent être donnés en tant que liste ou tuple. Si un seul entier est donné en argument, un vecteur sera retourné contenant autant de 1 que spécifié. Une matrice de 0 se fait avec la fonction `np.zeros([N,M])`. Elle agit de la même manière que la fonction `np.ones()`.

Il existe également une fonction permettant de créer une matrice vide. Cela est souvent utilisé lorsque tous les éléments vont être changé par la suite. Cela permet d'accélérer le code tout en réservant l'espace mémoire nécessaire. La fonction est `np.empty([N,M])`. Il faut remarquer cependant que cette fonction retournera une matrice contenant tout de même des valeurs. Ces valeurs représentent ce qui

était déjà présent dans l'espace mémoire avant qu'il ne soit réservé. Toutes les valeurs devront donc être changées pour que l'utilisation de cette matrice aie du sens.

3.3 Les matrices inverses

Pour la résolution de systèmes matriciels, il est important de savoir comment obtenir les matrices inverses. Cela dit, toute matrice n'a pas nécessairement d'inverse clairement défini. Ainsi, selon le cas auquel un codeur fait face, il peut être appelé à générer une matrice inverse avec `np.linalg.inv()` ou une matrice pseudo inverse avec `np.linalg.pinv()`. Le concept de pseudo inverse est une généralisation de la matrice inverse, applicable à toutes les matrices, carrées ou non. Cependant, les propriétés de l'inverse ne sont pas nécessairement préservé. Cela peut servir pour faire des approximations de courbes, par exemple avec la méthode de Vandermonde qui sera vu plus loin.

Script 9
Démonstration d'une matrice inverse

```
1 import numpy as np
2
3 a = np.array([[2,-5,3],[0,7,-2],[-1,4,1]])
4
5 a_inv = np.linalg.inv(a)
6
7 print(np.dot(a,a_inv))
```

```
array([[ 1.00000000e+00, -1.38777878e-17, -1.11022302e-16],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [-2.77555756e-17, 1.38777878e-17, 1.00000000e+00]])
```

SCRIPT 10
Dmonstration d'une matrice pseudo inverse

```
import numpy as np

a = np.array([[0,2,-1],[3,-2,1],[3,2,-1]])

a_pinv = np.linalg.pinv(a)

print('Matrice pseudo inverse de a')
print(a_pinv)
print('Résultat du produit matriciel de a et sa pseudo inverse')
print(np.dot(a,a_pinv))
```

```
Matrice pseudo inverse de a
[[ 0.         0.16666667 0.16666667]
 [ 0.13333333 -0.13333333 0.13333333]
 [-0.06666667 0.06666667 -0.06666667]]
Résultat du produit matriciel de a et sa pseudo inverse
[[ 0.33333333 -0.33333333 0.33333333]
 [-0.33333333 0.83333333 0.16666667]
 [ 0.33333333 0.16666667 0.83333333]]
```

3.4 Multiplication matricielle

Chapitre 4

Chapitre 4 : Résolution d'un système matriciel