Oral

Python

MP, PC, PSI, TSI

Calcul matriciel

On travaille avec les modules `numpy` et `numpy.linalg`.

```
import numpy as np
import numpy.linalg as alg
```

Création de matrices

Pour définir une matrice, on utilise la fonction `array` du module `numpy`.

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

L'attribut `shape` donne la taille d'une matrice : nombre de lignes, nombre de colonnes. On peut redimensionner une matrice, sans modifier ses termes, à l'aide de la méthode `reshape`.

```
A.shape
(2, 3)
A = A.reshape((3, 2))
A
array([[1, 2],
       [3, 4],
       [5, 6]])
```

L'accès à un terme de la matrice `A` se fait à l'aide de l'opération d'indexage `A[i, j]` où `i` désigne la ligne et `j` la colonne. **Attention, les indices commencent à zéro !** À l'aide d'intervalles, on peut également récupérer une partie d'une matrice : ligne, colonne, sous-matrice. Rappel, `a:b` désigne l'intervalle ouvert à droite $[\![a, b[\![$, `:` désigne l'intervalle contenant tous les indices de la dimension considérée. Notez la différence entre l'indexation par un entier et par un intervalle réduit à un entier.

```
A[1, 0]     # terme de la deuxième ligne, première colonne
3
A[0, :]     # première ligne sous forme de tableau à 1 dimension
array([1, 2])
A[0, :].shape
(2,)
A[0:1, :]   # première ligne sous forme de matrice ligne
array([[1, 2]])
A[0:1, :].shape
(1, 2)
A[:, 1]     # deuxième colonne sous forme de tableau à 1 dimension
array([2, 4, 6])
A[:, 1:2]   # deuxième colonne sous forme de matrice colonne
array([[2],
       [4],
       [6]])
A[1:3, 0:2] # sous-matrice lignes 2 et 3, colonnes 1 et 2
array([[3, 4],
       [5, 6]])
```

Les fonctions `zeros` et `ones` permettent de créer des matrices remplies de 0 ou de 1. La fonction `eyes` permet de créer une matrice du type I_n où n est un entier. La fonction `diag` permet de créer une matrice diagonale.

```
np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
np.ones((3,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Enfin la fonction `concatenate` permet de créer des matrices par blocs.

```
A = np.ones((2,3))
B = np.zeros((2,3))
np.concatenate((A,B), axis=0)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
np.concatenate((A,B),axis=1)
array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.]])
```

Calcul matriciel

Les opérations d'ajout et de multiplication par un scalaire se font avec les opérateurs `+` et `*`.

```
A = np.array([[1,2], [3,4]])
B = np.eye(2)
A + 3*B
array([[ 4.,  2.],
       [ 3.,  7.]])
```

Pour effectuer un produit matriciel (lorsque que cela est possible), il faut employer la fonction `dot`.

```
A = np.array([[1,2], [3,4]])
B = np.array([[1,1,1], [2,2,2]])
np.dot(A, B)
array([[ 5,  5,  5],
       [11, 11, 11]])
```

On peut également utiliser la méthode `dot` qui est plus pratique pour calculer un produit de plusieurs matrices.

```
A.dot(B)
array([[ 5,  5,  5],
       [11, 11, 11]])
A.dot(B).dot(np.ones((3,2)))
array([[ 15.,  15.],
       [ 33.,  33.]])
```

La fonction `matrix_power` du module `numpy.linalg` permet de calculer des puissances de matrices.

```
alg.matrix_power(A,3)
```

```
array([[ 37,  54],
       [ 81, 118]])
```

La transposée s'obtient avec la fonction `transpose`. L'expression `A.T` renvoie aussi la transposée de `A`.

```
np.transpose(B)
array([[1, 2],
       [1, 2],
       [1, 2]])
B.T
array([[1, 2],
       [1, 2],
       [1, 2]])
```

Le déterminant, le rang et la trace d'une matrice s'obtiennent par les fonctions `det`, `matrix_rank` du module `numpy.linalg` et `trace` du module `numpy`. Enfin la fonction `inv` du module `numpy.linalg` renvoie l'inverse de la matrice s'il existe.

```
alg.det(A)
-2.0000000000000004
alg.matrix_rank(A)
2
np.trace(A)
5
alg.inv(A)
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Pour résoudre le système linéaire $Ax = b$ lorsque la matrice A est inversible, on peut employer la fonction `solve` du module `numpy.linalg`.

```
b = np.array([1,5])
alg.solve(A, b)
array([ 3., -1.])
```

La fonction `eigvals` du module `numpy.linalg` renvoie les valeurs propres de la matrice. Pour obtenir en plus les vecteurs propres associés, il faut employer la fonction `eig`.

```
alg.eigvals(A)
array([-0.37228132,  5.37228132])
L = alg.eig(A)
L
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
        [ 0.56576746, -0.90937671]]))
L[1].dot(np.diag(L[0])).dot(alg.inv(L[1]))
array([[ 1.,  2.],
       [ 3.,  4.]])
```

La fonction `vdot` permet de calculer le produit scalaire de deux vecteurs de $\mathbb{R}^n$.

```
u = np.array([1,2])
v = np.array([3,4])
np.vdot(u, v)
11
```

La fonction `cross` permet de calculer le produit vectoriel de deux vecteurs de $\mathbb{R}^3$.

```
u = np.array([1,0,0])
v = np.array([0,1,0])
np.cross(u, v)
array([0, 0, 1])
```