

# Optimisation linéaire et convexité

## TP1

xx janvier 2016

Maxime CHUPIN, bureau 16-26-333, [chupin@ann.jussieu.fr](mailto:chupin@ann.jussieu.fr), le sujet et les corrections éventuelles pourront être trouvés sur le site <https://ljl11.math.upmc.fr/~chupin/>.

L'objectif de cette séance est de réviser les fonctions de bases de la librairie `numpy` de Python liées au calcul matriciel, de les utiliser dans le cadre de la résolution de systèmes linéaires :  $Ax = b$ , où  $A \in \mathcal{M}_{m,n}(\mathbf{R})$ ,  $b \in \mathbf{R}^n$  et  $x \in \mathbf{R}^m$  désigne l'inconnue et d'implémenter l'algorithme du pivot.

## 1 Calcul matriciel avec Python et numpy

Python est devenu un standard aussi bien dans le monde académique (recherche, enseignement, lycée, etc.) que dans le monde industriel. C'est un langage de programmation simple d'accès (au moins en surface) et d'une redoutable efficacité. Il est libre et s'utilise sur toutes les plateformes (Linux, Mac OSX, Windows).

De plus, pour le calcul scientifique, on dispose de la librairie `numpy` qui permet de rendre encore plus facile toutes les opérations de bases que l'on peut vouloir faire dans ce contexte (algèbre linéaire, optimisation, statistique, etc.).

Ici, il s'agit de se familiariser avec la manipulation de matrices et quelques opérations d'algèbre linéaire.

Comme tout apprentissage d'un langage de programmation, il *faut* utiliser la gigantesque source d'information qu'est internet. Le site officiel de `numpy` est très bien fait : <http://docs.scipy.org/doc/numpy/reference/index.html>. Je propose aussi l'introduction de Stéphane Gaïffas du CMAP : [http://www.cmap.polytechnique.fr/~gaiffas/intro\\_python.html](http://www.cmap.polytechnique.fr/~gaiffas/intro_python.html).

### 1.1 Écrire un script python

Python peut s'utiliser en *mode console*. L'interpréteur interactif permet d'écrire et d'exécuter du code Python à la volée, de faire des tests rapides, d'obtenir facilement des informations sur une fonction ou un module... Ça peut être un bon moyen de faire des testes. Cependant, en utilisant l'interpréteur interactif, nous ne pouvons garder l'historique de nos commande pour résoudre tel ou tel problème.

C'est pourquoi nous utiliserons des fichiers regroupant nos instructions python : les fichiers de *script*. Sous Linux, tous nos scripts (nommé par exemple *TP1.py*) commenceront par :

```
1 #!/usr/bin/env python
```

Ceci permet d'indiquer au système quel programme doit être utilisé pour exécuter le script. Dans un terminal, nous lancerons le script comme ceci :

```
> ./TP1.py
```

Par exemple, le script suivant

```
1 #!/usr/bin/env python
2 print("Hello World!!")
```

affichera Hello World!! dans le terminal. Nous utiliserons abondamment la librairie `numpy`, donc tous nos scripts devraient commencer par les lignes suivantes :

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import numpy as np
```

La deuxième ligne indique l'encodage de votre fichier source, c'est-à-dire le code utiliser pour coder les caractères. Cela permet par exemple de mettre des caractères accentués dans les commentaires. Mes fichiers sont en tous en `utf8`.

**Les commentaires.** Un commentaire est un texte ajouté au code source d'un programme servant à décrire le code source, facilitant sa compréhension par les humains. Il est donc séparé du reste du code grâce à une syntaxe particulière, ce qui fait qu'en général, le commentaire est ignoré par le compilateur ou l'interpréteur du langage concerné. **Pour pouvoir (re)comprendre rapidement son code, le réutiliser plus tard et le partager avec d'autres humains, il est nécessaire de commenter correctement son code.**

Ici, la syntaxe particulière est le `#` en début de ligne qui fera que l'interpréteur python ignorera ce qui se trouve derrière ce caractère.

## 1.2 Bibliothèques `numpy`

La librairie `numpy` contient des fonctions essentielles pour traiter les tableaux, les matrices et les opérations de type algèbre linéaire avec Python.

Dans cette section, je mets un très rapide listing des différentes fonctions que nous allons utiliser tout au long de ce TP. Il s'agit juste d'une petite aide, il est nécessaire de compléter cela avec les documents d'introduction et de documentation que j'ai indiqués plus haut dans ce document.

**Les tableaux.** Pour définir des matrices, ou tableaux, nous utiliserons `numpy.array()`, c'est-à-dire la fonction `array()` de la librairie `numpy`. Pour des questions pratiques, nous importons la librairie `numpy` avec la ligne suivante

```
1 import numpy as np
```

qui nous permet de faire référence à `numpy` par simplement `np`.

Si nous voulons définir un tableau d'une ligne contenant les valeurs (1, 2, 3, 4), on écrira alors :

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import numpy as np
5
6 A = np.array([1, 2, 3, 4])
```

Pour une matrice de deux lignes par exemple on fera :

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import numpy as np
5
6 A = np.array([[1,2,3,4],[5,6,7,8]])
```

Voici quelques opérations de bases sur les `array` :

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import numpy as np
5
6 # tableaux
7 A = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
8 # copie de tableau
9 B = A.copy()
10 # sans le .copy(), si on modifie la variable C, on modifie aussi A
11 C = A
12 # accès à un élément
13 print("A[0,3] = ", A[0,3])
14 # accès à une ligne entière
15 D = A[1,:]
16 # accès à seulement quelques éléments d'une colonne
17 # indice départ : indice arrivé : step
18 E = A[1:4:2, 2]
19
20 # affichage
21 print("A = ", A)
```

**Listes des fonctions utiles.** Voici une listes des fonctions qui peuvent vous être utiles pour ce TP (liste non exhaustive bien entendu).

```
1 type()
2
3 np.vstack()
4 np.hstack()
5 np.concatenate()
6 np.dot()
7 np.sum()
8 np.diag()
9 np.eye()
10 np.arange()
11 np.ones()
12 np.shape()
13 np.ndarray
14
15 np.linalg.eig()
16 np.linalg.inv()
17 np.linalg.matrix_rank()
18 np.linalg.solve()
```

### 1.3 Structure de contrôle, boucles et fonctions

**L'indentation est fondamentale en Python.** Les blocs sont identifiés par l'indentation, au lieu d'accolades comme en C ou C++; ou de `begin ... end` comme en Pascal ou en Ruby. Une augmentation de l'indentation marque le début d'un bloc, et une réduction de l'indentation marque la fin du bloc courant.

**Boucle `for`.** La syntaxe générale pour une boucle `for` est la suivante :

```
1 for element in objet_iterable:
2     traiter(element)
```

Par exemple, on peut écrire :

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 import numpy as np
5
6 # boucle: i variant de 0 à 9
7 for i in np.arange(10):
8     print("i = ", i)
```

**Structure `if/else/elif`.** Pour la structure de test, là encore, ce sont les tabulations qui définissent les blocs.

```
1 if mois == 'Décembre':
2     print 'Joyeux Noel'
3 elif mois == 'Janvier':
4     print 'Bonne année'
5 else :
6     print 'rien ce mois'
```

**Définition de fonction.** On peut définir des fonctions avec python. Voici la syntaxe :

```
1 def nom_fonction(liste de paramètres):
2     bloc d'instructions
```

Un exemple :

```
1 def sommeEntiers(stop):
2     i = 0
3     somme = 0
4     while i < stop:
5         i = i + 1
6         somme = somme + i
7     return somme
8
9 compteur(4)
10 compteur(2)
```

## 2 Exercices

**Exercice 1:** On considère la matrice  $A \in \mathcal{M}_{3,4}(\mathbf{R})$  définie par :

$$\begin{pmatrix} 4 & 6 & -2 & 3 \\ 2 & -1 & 0 & 1 \\ -7 & 0 & 1 & 12 \end{pmatrix}.$$

1. Définir la matrice  $A$  comme un `np.array()`.
2. Modifier la matrice  $A$  pour que ses deux premières lignes soient multipliées par 2 et que sa dernière colonne soit divisée par 3.
3. Créer une nouvelle matrice  $B$  définie par

$$\begin{pmatrix} 4 & 5 & 6 \\ 5 & 10 & 15 \\ 1 & 1 & 1 \end{pmatrix},$$

en utilisant le fait que les lignes 1 et 2 sont composées des éléments successifs de deux suites arithmétiques (voir fonction `np.arange` et `np.ones()`).

4. Créer la matrice  $C \in \mathcal{M}_{3,3}(\mathbf{R})$  extraite de  $A$  telle que pour  $1 \leq i, j \leq 3$ ,  $c_{ij} = a_{ij}$ .
5. Différents produits matriciels

→ Réaliser le produit matriciel  $D$  de  $B$  et  $A$  (`np.dot()`).

→ Réaliser le produit d'HADAMARD  $E$  de  $B$  et de  $C$ .

Pour mémoire, le produit d'HADAMARD  $E \in \mathcal{M}_{3,3}(\mathbf{R})$  des matrices  $B \in \mathcal{M}_{3,3}(\mathbf{R})$  et  $C \in \mathcal{M}_{3,3}(\mathbf{R})$  est défini par

$$\forall 1 \leq i, j \leq 3, \quad e_{ij} = c_{ij}b_{ij}.$$

6. Calculer la somme des éléments de la matrice  $E$  et le vecteur colonne  $Y \in \mathbf{R}^3$  tel que pour  $1 \leq i \leq 3$ ,  $y_i = \sum_{j=1}^4 d_{ij}$  (`np.sum()`).

**Exercice 2:** Nous allons ici explorer quelques commandes très utiles en algèbre linéaire.

On considère la matrice  $A \in \mathcal{M}_{4,4}(\mathbf{R})$  suivante

$$\begin{pmatrix} 4 & 5 & 6 & -1 \\ 5 & 10 & 15 & 2 \\ 6 & 15 & 1 & 4 \\ -1 & 2 & 4 & -2 \end{pmatrix}.$$

1. Pourquoi  $A$  est-elle diagonalisable? À l'aide de la fonction `np.linalg.eig()`, calculer avec Python ses valeurs propres et donner une base de vecteurs propres.
2. Calculer de deux manières l'inverse de  $A$  en utilisant le résultat précédent et la fonction `np.linalg.inv`. Comparer les résultats obtenus (vous pourrez regarder la librairie `matplotlib` pour faire des tracés avec Python).
3. On considère maintenant la matrice  $A \in \mathcal{M}_{n,n}(\mathbf{R})$

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix}$$

Examiner, comme à la question précédente, les cas  $n = 5, 10, 50$  (on utilisera  $n$  comme une variable, et on devra construire la matrice avec `np.eyes()` en fonction de  $n$ ). Comparer les valeurs propres obtenues avec les valeurs définies pour  $1 \leq k \leq n$  par

$$\lambda_k = 4 \sin^2 \left( \frac{k\pi}{2(n+1)} \right).$$

(On pourra utiliser la fonction `np.sort()`)

**Exercice 3:** On s'intéresse à la matrice  $A \in \mathcal{M}_{3,5}(\mathbf{R})$

$$\begin{pmatrix} 1 & -1 & 2 & 1 & 2 \\ -1 & 2 & 3 & -4 & 1 \\ 0 & -1 & 1 & 0 & 0 \end{pmatrix}.$$

1. Quel est le rang de la matrice  $A$ ? Utiliser la fonction `np.linalg.matrix_rank` afin de retrouver ce résultat.
2. En définissant :

$$A = \begin{pmatrix} 1 & -1 & 2 \\ -1 & 2 & 3 \\ 0 & -1 & 1 \end{pmatrix}, \text{ et } b = \begin{pmatrix} 3 \\ -7 \\ 1 \end{pmatrix},$$

résoudre à l'aide de la fonction `np.linalg.solve()` le système  $Ax = b$ .

### 3 Systèmes linéaire et pivot de GAUSS-JORDAN

Bien qu'il existe, comme on vient de le voir, des outils génériques pour traiter la résolution de problèmes linéaires avec Python, on va s'intéresser à l'implémentation dans cet environnement de la méthode du pivot. Il s'agit d'éviter de recourir aux boîtes noires que constituent les fonctions préprogrammées et surtout de prendre en main Python et d'apprécier la simplicité de l'implémentation des opérations associées à la méthode.

**Exercice 4:** On s'intéresse au système de 5 équations linéaires à 7 inconnues  $(x_i)_{1 \leq i \leq 7}$

$$\begin{cases} 4x_1 + 2x_2 + x_3 + 4x_4 + 5x_5 + 6x_6 + 7x_7 = 1 \\ x_1 + 4x_2 - 6x_3 + 2x_4 + 2x_5 - 2x_7 = 2 \\ x_1 - 2x_2 + 3x_3 + 3x_4 - 5x_5 + 6x_6 + x_7 = -10 \\ x_1 - x_2 + x_3 - x_4 + 2x_5 - 3x_6 - x_7 = -2 \\ x_1 + x_2 + 8x_4 + 2x_5 + 3x_6 + 4x_7 = 3 \end{cases}$$

1. Reformuler le problème sous la forme  $Ax = b$ .
2. Définir la matrice  $M \in \mathcal{M}_{5,8}(\mathbf{R})$  obtenue par adjonction du vecteur colonne  $b$  à la matrice  $A$ , c'est-à-dire pour  $1 \leq j \leq 7$  et  $1 \leq i \leq 5$ ,  $M_{ij} = A_{ij}$  et pour  $1 \leq i \leq 5$ ,  $M_{i8} = b_i$ .

C'est à la matrice  $M$  que l'on va appliquer la méthode du pivot de GAUSS. On adopte alors la notation suivante :

$$M = (L_1, \dots, L_5)^T,$$

où, pour  $1 \leq i \leq 5$ ,  $L_i$  désigne le vecteur ligne associé à la  $i$ -ème ligne de  $M$ .

### Description de l'algorithme.

- (a) *Première itération du pivot.* Si  $a_{11} \neq 0$ ,
  - $L_1 \leftarrow L_1/a_{11}$ ;
  - $L_i \leftarrow a_{i1}L_1$  pour  $2 \leq i \leq 5$ .
- (b) *Deuxième itération du pivot.* Si  $a_{22} \neq 0$ ,
  - $L_2 \leftarrow L_2/a_{22}$ ;
  - $L_i \leftarrow a_{i2}L_2$  pour  $3 \leq i \leq 5$ .
- (c) etc.

3. **Définition d'une fonction pivot.** Écrire une fonction Python qui généralise l'algorithme décrit. L'algorithme prend pour argument une matrice  $M \in \mathcal{M}_{n,m}(\mathbf{R})$  et renvoie une matrice  $N \in \mathcal{M}_{n,m}(\mathbf{R})$  triangulaire supérieure. Voici le prototype possible d'une telle fonction

```
1 def pivot(M):
2     ... serie d'instructions ...
3     ...
4     ...
5     return ...
```

- 4. Appliquer cette fonction à la matrice  $M$  définie précédemment.
- 5. Générer sous Python via la fonction `np.random.rand()` une matrice  $Q \in \mathcal{M}_{20,30}(\mathbf{R})$ , dont les valeurs sont toutes comprises entre 0 et 1. Appliquer la méthode du pivot au système  $Qx = d$  où  $d \in \mathbf{R}^{20}$  vérifie  $d_i = i$  pour  $1 \leq i \leq 20$  (on pourra utiliser les fonctions `np.arange()` et `np.reshape()`).