La simulation numérique
Numpy et Matplotlib
Les modules incontournables du calcul scientifique

Anis SAIED
Institut Préparatoire aux Etudes d'Ingénieurs de Nabeul (IPEIN),
Université de Carthage,
Tunisie
anis.saieed@gmail.com

Mise à jour la plus récente :3 février 2017

Table des matières

Introduction

Ce cours débute par une introduction aux modules Numpy et Matplotlib de Python. Il a pour but de présenter l'essentiel grâce à des exemples à saisir dans la console (sous Idle) et se termine par quelques exercices de représentation graphique.

1 Le module Numpy : Présentation

Le calcul scientifique nécessite la manipulation de données en quantité souvent importante.
NumPy (**Num**erical **Py**thon) est une bibliothèque Python pour travailler avec les grands ensembles de données de manière efficace.
Le module numpy est la boîte à outils indispensable pour faire du calcul scientifique avec Python.
Pour modéliser les vecteurs, les matrices, et plus généralement les tableaux à n dimensions, numpy fournit le type ndarray (N dimensional array).
Il y a des différences majeures avec les listes (resp. les listes de listes) qui pourraient elles aussi nous servir à représenter des vecteurs (resp. des matrices) :

- Les tableaux numpy sont homogènes, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc.
- La taille des tableaux numpy est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme le ferait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ce module n'est pas fourni avec la distribution Python de base, il doit être installé à partir du site officel :
`http://docs.scipy.org/doc/numpy-1.10.1/user/install.html`
Le module numpy [1] propose plusieurs sous-modules intéressants :

- `numpy.linalg` : un module d'algèbre linéaire basique,
- `numpy.random` : un module pour les générateurs aléatoires,

Comment charger le module `numpy` ?
On charge traditionnellement l'ensemble du module numpy en le renommant `np` :

```
import numpy as np
```

Pour ne pas surcharger l'espace des noms et risquer la présence d'homonymes, on ne charge pas un module par l'instruction : `from module import *`

2 Les tableaux numpy

2.1 Création avec `np.array`

On définit souvent un tableau numpy à partir d'une liste (ou liste de listes), en utilisant la fonction **`array`** du module numpy.
La fonction **`array`** agit comme un mécanisme de conversion de type `'list'` vers `'numpy.ndarray'` qui est le type commun à tous les tableaux numpy.
Exemple :

1. http ://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

```
a = np.array([  [1, 2, 3],
                [2, 3, 4],
                [0, 1, 0]])

#[[1 2 3]
# [2 3 4]
# [0 1 0]]
```

Mais ce ne sont pas des listes :

```
print(type(a))

# <class 'numpy.ndarray'>
```

L'attribut **dtype** :

```
print(a.dtype)

# int64
```

⚠ Il ne faut pas confondre le type des tableaux numpy : 'numpy.ndarray' avec le résultat renvoyé par l'attribut **dtype** (abréviation de *data type*) de a, 'int64' qui indique le type commun à tous les éléments du tableau, ici des entiers codés sur 64 bits, c'est-à-dire quatre octets ou *bytes*.

Remarque : la méthode tolist d'un tableau numpy le transforme en la liste (éventuellement une liste de listes) de ses éléments. Attention, ce n'est pas une fonction du module numpy.
On n'écrira donc pas np.tolist(a) mais a.tolist().

Exemple :

```
a = np.array( [1, 2, 3] ) # créer un tableau à partir d'une liste
b =a.tolist() #créer un objet b de type list, copie de l'objet a de type ndarray

# [1 2 3]
```

2.2 Création avec des fonctions spéciales

2.2.1 Vecteurs spéciaux (tableaux uni-dimensionnels)

La fonction **arange** crée des vecteurs de *valeurs régulièrement espacées*.
La syntaxe est **arange(d,f,h,dtype=...)**, et génère les valeurs de l'intervalle [d, f[avec un pas de h.

Exemples :

```
t0 = np.arange(0, 10, 2) # début=0, fin(exclue)=10 , pas=2
t1 = np.arange(3) # tableau de 3 entiers
t2 = np.arange(3.) # tableau de 3 réels
t3 = np.array([2]*5); #concaténation de 5 exemplaires de la liste [2]
print(t0,t1,t2,t3)

# [0 2 4 6 8] [0 1 2] [ 0. 1. 2.] [2 2 2 2 2]
```

La fonction **linspace(a,b,n)** retourne un vecteur de n valeurs régulièrement échelonnées du segment [a, b] (donc ici les extrémités sont incluses). Par défaut, n = 50. Si b < a, les valeurs sont obtenues dans l'ordre décroissant.
Ces fonctions sont en particulier utilisées pour le tracé des fonctions.

```
b = np.linspace(0, 10, 5); b # début=0, fin(inclue)=10 , n=5
#[ 0. 2.5 5. 7.5 10.]
```

2.2.2 Matrices spéciales (tableaux bi-dimensionnels)

Découvrons quelques fonctions spéciales : `np.ones`, `np.zeros`, `np.eye`, `np.diag`,...

```
a = np.ones((3, 5)); a # 3 lignes , 5 colonnes, dtype = float (par défaut)
b = np.ones((3,5),np.int); b
c = np.zeros((3, 5)); c
d = np.zeros((3, 5),dtype=np.bool); d
e = np.eye(3); e #Créer une matrice identité d'ordre n
g = np.eye(4,4); g
h = np.diag([1,2,3]); h #Créer une matrice diagonale
i = np.diag([1,2,3],1); i # essayer : -1, 2 (décalage de la diagonale)

"""
np.concatenate : permet de créer des matrices par blocs
"""
A=np.ones((2,3)); A
B=np.zeros((2,3)); B
AB0 = np.concatenate((A,B),axis=0); AB0 #B sous A
AB1 = np.concatenate((A,B),axis=1); AB1 #B à droite de A

"""
np.column_stack : construire la matrice dont les colonnes sont les vecteurs
passés en arguments.
"""
v1 =np.array([1,2,3]); v1
v2 =np.array([12,22,32]); v2
v3 =np.array([13,23,33]); v3
v = np.column_stack((v1,v2,v3)); v
```

2.2.3 Tableaux répondant à une formule donnée

La fonction **`fromfunction`** permet de construire un tableau dont le terme général obéit à une formule donnée.

```
def f(i,j): return 10*i+j
t1 = np.fromfunction(f,(4,5)); t1 # t1[i,j] = f(i,j)
t2 = np.fromfunction(f,(4,5),dtype=int); t2 #On exige un tableau de type int
"""
Le même tableau peut être obtenu en convertissant une liste
(ou une liste de listes) en compréhension vers un tableau Numpy.
"""
t3 = np.array([[f(i,j) for j in range(5)] for i in range(4)]); t3
```

Exemple :
Matrice d'Hilbert, de terme général : $H[i,j] = \frac{1}{(i+j+1)}$ avec $i, j \geq 0$

```
t4 = np.fromfunction(lambda i,j: 1/(i+j+1),(4,4)); t4
```

2.3 Lecture dans un tableau

2.3.1 Lecture de valeurs dans un vecteur, « slicing »

La lecture d'éléments d'un tableau numpy procède par coupes (« slices » en anglais).

M[indice_début(inclu) : indice_fin(exclu):incrément].

Exemples :

```
v = np.array(range(0,160,10)); v
# le 1er élément
v[0] #compte à partir du début
v[-v.size] #on compte à partir de la fin
# le dernier élément
v[-1]
v[v.size-1]
```

Quelques coupes de valeurs consécutives dans l'ordre des indices croissants :

```
v[4:11] # de v[4] à v[10], donc 11-4 = 7 éléments consécutifs
v[:] #la totalité du vecteur v
v[::-1] #la totalité du vecteur v mis à l'envers (incrément = -1)
v[:6:2] # (incrément = 2)
```

Accès aux données dans un ordre quelconque :
Si v est un tableau, et si *I* est un tableau d'indices (éventuellement une liste ou un tuple), alors *v[I]* renvoie le tableau (de même format que le tableau *I*) formé des éléments *v[i]*, où *i* est dans *I*.

```
v = np.array(range(0,160,10)); v
a = v[[6,2,1,3]]; a
```

Remarque : le tableau a formé ci-dessus (et qui n'est pas obtenu par « slicing ») n'est pas une simple « vue » sur une partie de v. Il est donc indépendant de v (modifier a n'affecte pas v et réciproquement).

2.3.2 Lecture de valeurs dans une matrice

Il est possible de sélectionner une sous matrice, en ne gardant que quelques lignes consécutives et/ou certaines colonnes consécutives.
La spécification la plus générale d'une coupe d'un tableau M est :

```
M[numero_ligne : numero_colonne].
M[ligne_début(inclue) : ligne_fin(exclue),colonne_début(inclue) :
colonne_fin(exclue)].
```

Si M est une matrice d'ordre $a * b$:

- `M[a,b]` : élément en position (a,b)
- `M[a] ou M[a,:]` : Vecteur ligne en position a
- `M[:,b]` : Vecteur colonne en position b
- `M[a:b,c:d]` : lignes de a à b-1, colonnes de c à d-1
- `M[:,:]` : une copie intégrale de M avec nouvelle référence
- `M[:a,:b]` : sous matrice les lignes du début jusqu'à *a* exclu ; les colonnes de début jusqu'à *b* exclu (Les a premières lignes, les b premières colonnes).

- `M[::-1]` : On inverse l'ordre des lignes
- `M[:,::-1]` : On inverse l'ordre des colonnes
- `M[::2,::2]` : lignes et colonnes paires
- `M[1::2,1::2]` : lignes et colonnes impaires

2.4 Écriture dans un tableau

Les tableaux numpy sont **mutables**, on peut donc modifier les valeurs qu'ils contiennent sans redéfinir l'objet lui-même.

2.4.1 Écriture de valeurs dans un vecteur

Si a est un vecteur numpy l'instruction « `a[n] = x` » écrit la valeur x en position n.
La valeur x écrite en position n du vecteur a doit a priori être compatible avec le « `data type` » de a.

```
a = np.arange(10); a # le vecteur des entiers de 0 à 9
a[7] = 21 # écrit la valeur 21 en position 7
a # affiche le nouveau contenu de a
```

```
#array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
#array([ 0, 1, 2, 3, 4, 5, 6, 21, 8, 9])
```

On peut écrire plusieurs valeurs simultanément dans un vecteur, en utilisant une syntaxe du genre `a[coupe ]=...`.
Il faut simplement veiller à ce que le membre de droite (la source) soit « `array-like` » (une liste, un tuple, un tableau) et que la coupe spécifiée *a* (donc la cible) soient de même taille. Ainsi, les instructions :
`a[4:7] = [111,222,333]`,
`a[4:7] = (111,222,333)` ou encore
`a[4:7] = np.array([111,222,333])` ont le même effet.

On peut écrire plusieurs valeurs simultanément dans un vecteur dans un ordre quelconque, en utilisant une syntaxe du genre `a[I]=V`, avec I : tableau d'indices et V : tableau de valeurs à inserer dans le tableau *a*.

```
a[[7, 2, 5, 3]] = (7777, 2222, 5555, 3333); a # on modifie a[7], a[2], a[5] et a[3]
```

2.4.2 Écriture de valeurs dans une matrice

```
a = np.array(range(10))
a.shape = (2, 5)
print(a)
```

```
#[[0 1 2 3 4]
#[5 6 7 8 9]]
```

```
a[0,0] = 7
print(a)
```

```
#[[7 1 2 3 4]
#[5 6 7 8 9]]
```

```
a[0,:] = [8, 7, 6, 5, 4]
```

```
#[[8 7 6 5 4]
#[5 6 7 8 9]]
```

3 Traçage des courbes : Module Matplotlib

Le module `Matplotlib` est chargé de tracer les courbes :

```
>>> import matplotlib.pyplot as plt
```

Dans cette section on donne un bref aperçu des possibilités graphiques de `Matplotlib`. Il est possible de générer des graphiques à deux et à trois dimensions, et d'agir facilement sur les attributs du graphe (couleurs, type de ligne, axes, annotations...).
Les commandes présentées ci-dessous sont utiles notamment pour être introduites dans des scripts et automatiser la production de figure.

la fonction `plot()` :
D'une manière générale les fonctions **`plt.plot`** attendent des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe.
La syntaxe de base de la fonction **`plot()`** est la suivante :
`plot(x,y)` permet de tracer une courbe reliant les points dont les **abscisses** sont données dans le vecteur **x** et les **ordonnées** dans le vecteur **y**.
Commençons par la fonction sinus :

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi

# échantillonner la variable x
x= np.arange(0,2*pi,pi/16) # ou x= np.linspace(0,2*pi,100)

plt.plot(x,np.sin(x),label="sin") # on utilise la fonction sinus de Numpy

plt.axis("equal") # Imposer une échelle identique sur les deux axes de coordonnées.

plt.xlim(-2,10) # Fixer l'intervalle de visualisation sur l'axe des abscisses.

plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")

plt.title('Fonction sinus')

# sauvegarder la figure (en .png ou .pdf), pour la voir ou la conserver
plt.savefig('ma_figure.png')

plt.show() #Afficher l'image

plt.clf()
```

Si tout se passe bien, une fenêtre doit s'ouvrir avec la figure ci-dessus.

Exercice :
Tracer la courbe représentative de $x \longmapsto cos(x)$ sur l'intervalle [-5, 5] à l'aide de la fonction `plot()`.
Pour utiliser la fonction `plot()` de `matplotlib.pyplot`, on pourra exécuter les instructions :

```
{ X = arange(-5,5,0.1)
{ Y = cos(X)
```

3.1 Manipuler plusieurs graphiques

On peut utiliser plusieurs fenêtres graphiques en même temps, à condition de stoker les figures dans des variables :

```
t = np.linspace(0,2*pi,50)
fig1 = plt.figure()
plt.plot(t,np.cos(t),label="cos")
fig2 = plt.figure()
plt.plot(t,np.sin(t),label="sin")
plt.show() #Afficher toutes les figures
```

Une même fenêtre peut intégrer plusieurs graphes non superposés grâce à la commande `subplot` : La commande `subplot(n,m,k)` permet de subdiviser la fenêtre en $n * m$ cases, n lignes et m colonnes (comme pour les matrices).
Ces cases sont numérotées de gauche à droite et de haut en bas. La valeur de k permet de spécifier dans quelle case on désire faire un graphique.

```
plt.subplot(2,1,1)
plt.plot(t,np.cos(t))
plt.subplot(2,1,2)
plt.plot(t,np.sin(t))
plt.show() #Afficher toutes les figures
```

Il est possible de superposer deux courbes sur le même graphique :

```
plt.plot(t,np.cos(t),t,np.sin(t))
plt.show() #Afficher toutes les figures
```

3.2 Graphiques à 3 dimensions : Les courbes

Il est possible de tracer des courbes dans l'espace avec la librairie Axes3D. Par exemple une hélice :

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.plot(np.cos(t),np.sin(t),t,label="helice")
plt.show()
```

Exemple 1 :

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
p1=plt.plot(x,np.sin(x),marker='o')
p2=plt.plot(x,np.cos(x),marker='v')
plt.title("Fonctions trigonometriques") # Problemes avec accents (plot_directive) !
plt.legend([p1, p2], ["Sinus", "Cosinus"])
plt.show()
```

Exemple 2 :

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```