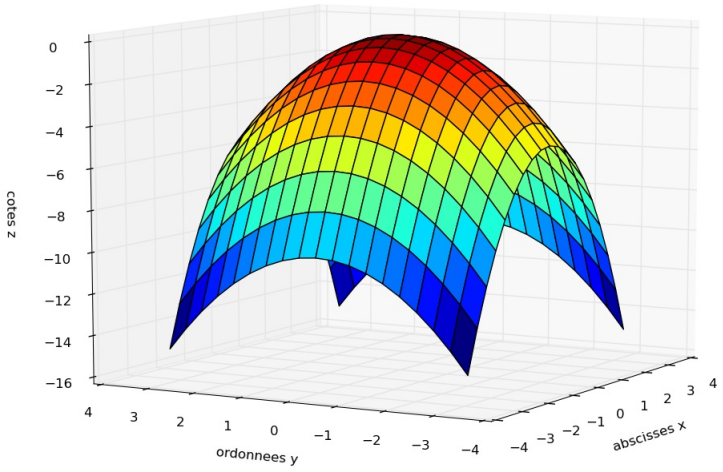
IPython, Numpy & Matplotlib



François Lefèvre

Université de Reims Champagne-Ardenne
Département de Mathématiques Mécanique et Informatique
Moulin de la Housse, 51062 REIMS Cedex 2
francois.lefevre@univ-reims.fr

MOIxxxx : Méthodes et Outils Informatiques
Licence de Mathématiques / Master Mathématiques & Applications

Table des matières

1 Généralités

Nous utiliserons sciemment `ipython3`[IPy] la version interactive de `python3`, accompagnée de la librairie `Matplotlib`[Mpl] qui donne un accès direct à de nombreuses fonctionnalités numériques et graphiques. Cet environnement permet un travail dans un esprit autodidacte similaire aux logiciels standarts tels que `Octave` et `Matlab`, mais aussi des facilités de déboguage grace à l'usage interactif.

1.1 Installation

* Sous LINUX, installer les paquets suivants dans un terminal en tapant la commande :

```
sudo apt-get install python3 ipython3 idle3 spyder3
sudo apt-get install python3-matplotlib python3-tk python3-scipy python3-sympy
```

* Sous WINDOWS/MacOS, on peut privilégier le logiciel scientifique `Anaconda`, libre de droits, bien qu'il ne fonctionne pas exactement comme dans l'environnement LINUX. Son éditeur de texte `Spyder` est très paramétrable et inclut une fenêtre de commande `ipython` ayant un comportement semblable à celui du terminal sous LINUX.
On peut aussi virtualiser une distribution de LINUX via VirtualBox, puis procéder comme plus-haut.

1.2 Lancement de IPython

Dans une fenêtre de Terminal LINUX, lancer la commande :

```
lefevre@Mac-Ubuntu:~$ ipython3 --pylab
```

Après affichage de l'écran d'accueil, on peut alors exécuter un script par la commande `run` suivie du nom du fichier script :

```
Python 3.4.3+ (default, Oct 14 2015, 16:09:02)
Type "copyright", "credits" or "license" for more information.
IPython 2.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
Using matplotlib backend: TkAgg

In [1]:
```

Le terminal devient alors réservé pour IPython. Le terminal redevient libre pour LINUX en tapant la commande IPython `quit`.

1.3 Modes de fonctionnement de IPython

IPython connaît deux modes de fonctionnement : le **mode en ligne** et le **mode programmation**.
* En mode "en ligne", les commandes sont écrites directement dans la fenêtre de commande après le prompt. Une commande en mode "en ligne" est exécutée après frappe de la touche "Entrée". On est dans un usage interactif entre l'utilisateur et le logiciel, ce qui permet une progression et un apprentissage pas à pas...
* En mode programmation, le programme doit d'abord être édité et enregistré via un bon éditeur de fichiers (idle ou spyder), puis il peut être exécuté sous IPython par la commande : `run`

`monscript.py`, sinon directement dans le Terminal par la commande `python3 monscript.py` (indépendamment donc de IPython).

```
In[1]: print("Hello world!") # :mode interactif
Hello world!
In[2]: run monscript.py      # :mode programmation
...
```

1.4 Positionnement dans un répertoire

Tout d'abord, on peut s'informer du répertoire où l'on est situé par la commande `pwd` (signifiant Print-Working-Directory). Pour changer de répertoire (nécessaire quand on écrit et range ses propres scripts et fonctions), on utilise la commande `cd` *nomdedossier* (`cd` signifiant Change Directory). La liste des fichiers du dossier courant s'obtient par la commande `ls` ou `ll`.
Toutes ces commandes étant issues d'Unix, on peut aussi bien les taper directement dans le Terminal, **avant** même d'avoir lancé la commande `ipython3 --pylab`.

```
In[3]: pwd
Out[3]: u'/home/lefevre'
In[4]: cd /home/toto/MOIxxxx
/home/toto/MOIxxxx
In[5]: ls
courbe1.py*  courbe3.py*  exo11.py*  exo5.py*  exo9.py*
courbe2.py*  exo10.py*    exo1.py*   exo8.py*  fctcarre.py*
```

commande Python et Linux	description
`pwd`	affiche le nom du répertoire courant
`cd` *directory*	changement de répertoire courant
`ls`	liste courte des fichiers du répertoire courant
`ll`	liste longue des fichiers du répertoire courant

Table 1 – Commandes relatives aux répertoires.

1.5 L'aide en ligne IPython

Pour obtenir des informations sur une fonction ou une instruction de votre connaissance, on a recours à la fonction `help()`. Par exemple pour l'instruction `for` :

```
In [6]: help("for")
```

L'aide s'ouvre alors à l'intérieur de la fenêtre, et on y navigue avec les flèches haut/bas du clavier ; taper la touche `q` pour quitter.

Lorsqu'il s'agit d'une fonction d'une librairie (et non plus d'une instruction de base), on n'a plus besoin des double-cotes. Par exemple pour la fonction `det` (qui appartient à la librairie `numpy`) :

```
In [6]: help(det)
```

Lorsqu'on ne connait pas la commande désirée, on dispose du moteur de recherche intégré à IPython-Matplotlib, qui est `lookfor("mot-clef")`, il nécessite de lui fournir un mot-clé (en anglais) **dans une chaîne de caractères**, par exemple :

```
In [7]: lookfor("determinant")
Search results for 'determinant'
--------------------------------
numpy.linalg.det
    Compute the determinant of an array.
numpy.linalg.slogdet
    Compute the sign and logarithm of the determinant of an array.
numpy.vander
    Generate a Van der Monde matrix.
numpy.linalg.qr
    Compute the qr factorization of a matrix.
numpy.ma.vander
    Generate a Van der Monde matrix.
numpy.linalg._umath_linalg.slogdet
    slogdet on the last two dimensions and broadcast on the rest.
```

Faîtes alors votre choix !

1.6 Entrées/Sorties (clavier/écran)

Pour saisir une variable au clavier avec affichage d'un message, on utilise l'instruction `input()` qui renvoit par défaut une chaîne de caractères. Si on veut saisir un nombre, on devra convertir cette chaîne de caractères par `int()` ou `float()`.
Pour afficher simplement une variable, on utilise l'instruction `print()`.

```
In [12]: e=int(input("Entrez un entier ... "))
Entrez un entier ...  20
In [13]: print(e,type(e))
20 <class 'int'>  # :c'est bien un entier (int)!
In [14]: prenom=input("Comment t'appelles-tu ? "); print(prenom,type(prenom))
Comment t'appelles-tu ? Toto
Toto <class 'str'>  # :c'est bien une chaîne de caractères (str)!
```

ATTENTION : Les fonctions `input()` et `print()` sont présentées ici pour python3, elles ont des comportements sensiblement différents en python2. Par exemple, python2 permettait de saisir facilement une liste par un `input()`, et il semble que cela soit devenu impossible en python3...

2 Variables et espace de travail

- Les variables de base de "type" entier, réel, complexe, chaîne de caractères ainsi que tableaux de nombres
- Affectation des variables par le symbole "="
- Aucune déclaration de type n'est nécessaire, c'est automatique (typage dynamique) et cela dépend de ce que l'on met à droite du symbole "="
- **ATTENTION (cas des objets MUABLES : tableaux, listes...) :** pour **copier** une variable `x` dans une variable `y`, il faut écrire `y=x.copy()` mais pas `y=x` : la première solution est la vraie recopie de mémoire, la seconde n'est que la création d'une référence (=`x` et `y` sont deux variables synonymes pointant vers la même zone mémoire : la modification de l'une entraîne automatiquement celle de l'autre) !

2.1 Types de base

```
In [1]: a=2;      print(a)          # :un entier
2
In [2]: b=1.001;  print(b)          # :un réel
1.001
In [3]: c=2-3.5j; print(c)          # :un complexe
(2-3.5j)
In [4]: ch="Hello World"; print(ch) # :une chaîne de caractères
Hello World
```

symbole	valeur
`1j`	$\sqrt{-1}$
`pi`	$\pi \approx 3.1416$

TABLE 2 – Constantes prédéfinies.

La commande `whos` informe sur les variables stockées dans l'espace de travail, leur type et leur valeur :

```
In [5]: whos
Variable   Type       Data/Info
------------------------------
a          int        2
b          float      1.001
c          complex    (2-3.5j)
ch         str        Hello World
```

La commande `del` *var* permet de supprimer une variable *var* de l'espace de travail ; la commande `reset -s -f` supprime toutes les variables de l'espace de travail.

```
In [6]: del a; del b
In [7]: whos
Variable   Type       Data/Info
------------------------------
c          complex    (2-3.5j)
ch         str        Hello World
In [8]: %reset -s -f
In [9]: whos
Interactive namespace is empty.
```

2.2 Chaînes de caractères

Une chaîne de caractères est spécifiée entre double quotes `"..."` :

```
In [10]: chaine="Bonjour à tous !"
In [11]: print(chaine)
Bonjour à tous !
```

On peut concaténer des chaînes de caractères au moyen de l'opérateur `"+"`, et dupliquer par l'opérateur `"*"` ; par exemple :

```
In [12]: "Toto, " + "dis bonjour a la dame !"  # :concaténation
Out[12]: 'Toto, dis bonjour a la dame !'
In [13]: "Toto! "*3                           # :duplication
Out[13]: 'Toto! Toto! Toto! '
```

2.3 Formats d'affichage

Pour réaliser l'affichage de messages composés, on doit donc concaténer des chaînes de caractères au moyen de l'opérateur "+". Les fonctions `str()` et `format()` permettent des conversions numériques en chaînes de caractères. `str()` est la plus simple :

```
In [12]: n=5
In [13]: print("n="+str(n)+", pi="+str(pi)) # :format par défaut
n=5, pi=3.14159265359
```

`format()` nécessite de compléter l'information de comment afficher le nombre en spécifiant son *format*.

```
In [14]: print("pi=" + format(pi,"9.2e"))   # :format scientifique
pi= 3.14e+00
```

Les choix des formats possibles sont (similaires au langage C) :

format	description
d	l'argument est traité comme un entier
f	l'argument est traité comme un nombre à virgule flottante
e	l'argument est traité en notation scientifique
s	l'argument est traité comme une chaîne de caractères

TABLE 3 – Spécificateurs de formats selon les types.

Le premier chiffre (optionnel) au début du format est **la largeur de champ** : c'est à dire le nombre de caractères réservés à l'écran pour l'affichage du nombre. Le second chiffre (optionnel aussi) après le point est **le nombre de décimales** souhaitées pour les formats de réels "f" et "e".

```
In [1]: A=array([[1,2,3],[4,5,6]]); [m,n]=shape(A)
In [2]: print("La dimension du tableau est " +format(m,"1d")+\
  ....: " fois " +format(n,"1d")+ ".")
La dimension du tableau est 2 fois 3.

In [3]: x=sqrt(2)
In [4]: print("La valeur de x est: "+format(x,"8.4f"))
La valeur de x est:   1.4142
In [5]: print("La valeur de x est: "+format(x,"10.8f"))
La valeur de x est: 1.41421356
In [6]: print("La valeur de x est: "+format(x,"14.8e"))
La valeur de x est: 1.41421356e+00
```

Notez que pour une largeur de champ demandée, le nombre est systématiquement cadré à droite (l'affichage est complété de caractères espaces à gauche au besoin).

2.4 Listes

Une liste en python est une collection de valeurs mises entre crochets `[..,..,..]` et séparées par des virgules ; éventuellement les valeurs peuvent être de types hétérogènes :

```
In [1]: l1=[2, 4, -3, 7, 8, 9]  # :liste d'entiers (fournie à la main)
In [2]: l2=range(-1,9,1)        # :liste d'entiers (suite arithmétique)
In [3]: print(list(l2))
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
In [4]: l3=[1, 3, "toto", 3.14] # :liste non-homogène
In [5]: print(list(range(4,-4,-1))) # :liste d'entiers (suite arithmétique)
[4, 3, 2, 1, 0, -1, -2, -3]
In [6]: print(list(range(6)))       # :liste de 6 entiers (partant de 0)
[0, 1, 2, 3, 4, 5]
In [7]: whos
Variable   Type    Data/Info
---------------------------
l1         list    n=6
l2         range   range(-1, 9)
l3         list    n=4
```

Notez que la fonction `range(a,b,pas)` fournie une suite arithmétique d'entiers espacés d'un **pas entier** dans l'intervalle $[a, b[$ (si $a < b$ et $pas > 0$) ou bien dans l'intervalle $]b, a]$ (si $b < a$ et $pas < 0$) : **la borne b n'y est donc jamais !** Les variables `a` et `b` doivent aussi être **entières**. L'appel `range(a,b)` équivaut à `range(a,b,1)`, donc avec un `pas=1` par défaut. L'appel `range(b)` équivaut à `range(0,b,1)`, donc avec un `pas=1` et une borne `a=0` par défaut.
On peut récupérer la longueur d'une liste comme celle d'une chaîne par la fonction `len()` :

```
In [8]: len(l3)
Out[8]: 4
```

2.5 Tableaux numériques

En Python, `ndarray` est le type **tableau numérique** : il sert en particulier à représenter des vecteurs (=tableau monodimensionnel, à **un seul** indice !), et des matrices (=tableau bidimensionnel, à **deux** indices).

2.5.1 Vecteur : tableau de dimension 1

Un tableau monodimensionnel consiste à **convertir en tableau numérique la liste de ses coefficients**, par la fonction `array()` :

```
In [1]: v=array([7, -1, 2, 0]); print(v)
[ 7 -1  2  0]
In [2]: whos
Variable   Type       Data/Info
-------------------------------
v          ndarray    4: 4 elems, type ‘int64‘, 32 bytes
In [3]: type(v)      # :quel est le type de ma variable v ?
Out[3]: numpy.ndarray
```

Observer le type `ndarray` provenant de la librairie `numpy`.
On peut récupérer la dimension d'un vecteur par la fonction `size()` :

```
In [4]: size(v)
Out[4]: 4
```

Insistons : un vecteur ainsi représenté est bien **un tableau contenant une liste 1D de valeurs numériques : en aucun cas il n'a de forme ligne ou colonne**. Si l'on veut donner effectivement la forme ligne ou colonne à un "vecteur", alors il faudra le représenter comme une un tableau bidimensionnel, c'est à dire en faire en quelque sorte une "**matrice ligne**" ou bien une "**matrice colonne**".
Pour extraire un ou plusieurs coefficients d'un vecteur, on a recours aux crochets `[...]` ; **notez que les indices commencent à 0** ; en cas d'extraction multiple de coefficients, l'indice est remplacé par une liste d'indices :

```
In [1]: v=array([7, -1, 2, 0])
In [2]: v[0]
Out[2]: 7
In [3]: v[1]
Out[3]: -1
In [4]: v[2]
Out[4]: 2
In [5]: v[3]
Out[5]: 0
In [6]: v[[1,2]]
Out[6]: array([-1,  2])
```

Création d'un vecteur numérique dont les coefficients sont espacés d'un incrément constant par `arange(a,b,pas)` :

```
In [7]: arange(-1, 1, 0.25)
Out[7]: array([-1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75])
In [8]: arange(1, -1, -0.25)
Out[8]: array([ 1.    0.75  0.5   0.25  0.   -0.25 -0.5  -0.75])
In [9]: arange(-1, 1.001, 0.25)
Out[9]: array([-1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75  1.])
```

Contrairement à `range(a,b,pas)` qui nécessite des arguments **entiers**, notez que `arange(a,b,pas)` permet des arguments **réels**. Sinon, les mêmes remarques que pour `range` s'appliquent (valeurs par défaut, absence de la borne `b` du résultat produit).
Les vecteurs génériques de dimensions quelconques :

syntaxe Numpy	description
`zeros(n)`	vecteur de dimension n ne contenant que des 0
`ones(n)`	vecteur de dimension n ne contenant que des 1
`rand(n)`	vecteur aléatoire avec distribution uniforme sur $[0, 1]$
`randn(n)`	vecteur aléatoire avec distribution normale

TABLE 4 – Vecteurs génériques de dimensions quelconques.

```
In [10]: zeros(4)
Out[10]: array([ 0.,  0.,  0.,  0.])
```

2.5.2 Matrice : tableau de dimension 2

Déclaration de la matrice $A = \begin{pmatrix} 2 & 4 & 6 \\ 1 & 2 & 3 \end{pmatrix}$ et extraction de ses lignes/colonnes :

```
In [1]: A=array([[2,4,6],[1,2,3]]); print(A)
[[2 4 6]
 [1 2 3]]
In [2]: A[0,:]            # :ligne no 0 (la première ligne!)
Out[2]: array([2, 4, 6])
In [3]: A[1,:]            # :ligne no 1 (la seconde  ligne!)
Out[3]: array([1, 2, 3])
In [4]: A[:,0]            # :colonne no 0
Out[4]: array([2, 1])
In [5]: A[:,1]            # :colonne no 1
Out[5]: array([4, 2])
In [6]: A[:,2]            # :colonne no 2
Out[6]: array([6, 3])
```

L'extraction de blocs de manière plus générale est réalisée par l'opérateur deux-points " :" appelé **opérateur de slicing**. Notez que dans la séquence `d:f`, l'indice `f` n'est PAS dans la liste, retenir que :

`d:f` représente les indices `d,d+1,...,f-1`

```
In [7]: A[0:2,1:3]
Out[7]:
array([[4, 6],
       [2, 3]])
```

Déclaration d'une **matrice ligne** $L = \begin{pmatrix} 7 & -1 & 2 & 0 \end{pmatrix}$:

```
In [8]: L=array([[7, -1, 2, 0]]); print(L) # :notez les doubles [[ !
[[ 7 -1  2  0]]
```

Déclaration d'une **matrice colonne** $C = \begin{pmatrix} 3 \\ 5 \\ 8 \end{pmatrix}$:

```
In [9]: C=array([[3],[5],[8]]); print(C) # :notez les doubles [[ !
[[3]
 [5]
 [8]]
```

On peut récupérer les dimensions d'un tableau 2D soit par `shape()` soit par `size()` :

```
In [10]: [m,n]=shape(C)
In [11]: m
Out[11]: 3
In [12]: n
Out[12]: 1

In [13]: size(C,0)
Out[13]: 3
In [14]: size(C,1)
Out[14]: 1
```

Les matrices génériques de dimensions quelconques :

syntaxe Numpy	description
`zeros((m,n))`	matrice de dimension $(m \times n)$ ne contenant que des 0
`ones((m,n))`	matrice de dimension $(m \times n)$ ne contenant que des 1
`eye(m,n)`	matrice identité
`rand(m,n)`	matrice aléatoire avec distribution uniforme sur $[0, 1]$
`randn(m,n)`	matrice aléatoire avec distribution normale

TABLE 5 – Matrices génériques de dimensions quelconques.

syntaxe Numpy	description
`A.transpose()` ou `A.T`	matrice A^T transposée de A
`A.conj()`	matrice $\overline{A}$ conjuguée de A
`inv(A)`	matrice A^{-1} inverse de A

TABLE 6 – Opérations sur les matrices.

La fonction `diag()` a deux fonctionnements selon qu'elle s'applique à une matrice ou bien à un vecteur. `diag(A,k)` avec A une matrice et k un entier, extrait la $k$ème sur/sous-diagonale de A.

```
In [1]: A=array([[1,2,3],[4,5,6],[7,8,9]]); print(A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
In [2]: d0=diag(A);                          print(d0)
[1 5 9]
In [3]: d1=diag(A,1);                        print(d1)
[2 6]
In [4]: dm1=diag(A,-1);                      print(dm1)
[4 8]
```

Au contraire `diag(v,k)` avec v un vecteur et k un entier, produit la matrice carrée avec les coefficients de v sur sa diagonale numéro k.

```
In [1]: v=array([1,2,3]); print(v)
[1 2 3]
In [2]: D0=diag(v);       print(D0)
[[1 0 0]
 [0 2 0]
 [0 0 3]]
In [3]: Dm1=diag(v,-1);   print(Dm1)
[[0 0 0 0]
 [1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]]
```

3 Calculs numériques et opérateurs

3.1 Opérations

- Opérations élémentaires **entre deux scalaires** : `+,-,*,/,**`

```
In [1]: 2**3  # :élévation à la puissance
Out[1]: 8
In [2]: 5//2  # :division Euclidienne !
Out[2]: 2
In [3]: 5/2   # :division réelle, ATTENTION: c'est différent en Python2.7 !
Out[3]: 2.5
```

- **Opérations terme à terme** entre **un tableau et un scalaire** : `+,-,*,/,**`

```
In [1]: A=array([[1,2],[3,4]]); print(A)
[[1 2]
 [3 4]]
In [2]: A**2 # :composantes élevées au carré (terme à terme)
Out[2]:
array([[ 1,  4],
       [ 9, 16]])
```

- **Addition et soustraction** entre **2 tableaux de mêmes dimensions** (sinon message d'erreur) : `+,-`
- **Multiplication et division terme à terme** entre **2 tableaux de mêmes dimensions** : `*,/`

```
In [1]: A=array([[2, 4, 6],[1, 2, 3]]); print(A)
[[2 4 6]
 [1 2 3]]
In [2]: B=array([[10, 20, 30],[2, 4, 6]]); print(B)
[[10 20 30]
 [ 2  4  6]]
In [3]: A+B
Out[3]:
array([[12, 24, 36],
       [ 3,  6,  9]])
In [4]: A*B  # :terme à terme (ce N'EST PAS le produit matriciel!)
Out[4]:
array([[ 20,  80, 180],
       [  2,   8,  18]])
```

3.2 La librairie Numpy

- Matplotlib intègre automatiquement la librairie Numpy (entre autre) comprenant les fonctions mathématiques usuelles comme `sin`, `cos`, `exp` ... etc. Toutes ces fonctions calculent composante par composante sur des scalaires/vecteurs/matrices :

```
In [1]: x=pi/2; print(x)
1.57079632679
In [2]: y=sin(x); print(y)
1.0
In [3]: v=array([0, 0.5, 1]); print(v)
[ 0.   0.5  1. ]
In [4]: w=sin(v); print(w)
[ 0.          0.47942554  0.84147098]
```

```
In [5]: A=array([[1, 2, 3],[4, 5, 6]]); print(A)
[[1 2 3]
 [4 5 6]]
In [10]: B=sin(A)*cos(A); print(B)
[[ 0.45464871 -0.37840125 -0.13970775]
 [ 0.49467912 -0.27201056 -0.26828646]]
```

- Le **produit matriciel** (de l'algèbre linéaire) est réalisé sur les tableaux de type `ndarray` par la fonction `dot(A,B)`. Le produit matriciel comme on le sait n'est **pas commutatif**, l'ordre des arguments est donc important !

```
In [1]: A=array([[2, 3],[4, 5]]); print(A)
[[2 3]
 [4 5]]
In [2]: B=2*eye(2,2); print(B)
[[ 2.  0.]
 [ 0.  2.]]
In [3]: dot(A,B)  # :on ferait A*B en MATLAB (produit matriciel)
Out[3]:
array([[ 4.,  6.],
       [ 8., 10.]])
```

```
In [4]: x=array([2, 1]); print(x)
[2 1]
In [5]: y=ones(2); print(y)
[ 1.  1.]
In [6]: dot(A,x)  # :on ferait A*x   en MATLAB (matrice-vecteur)
Out[6]: array([ 7, 13])
In [7]: dot(y,A)  # :on ferait y.'*A en MATLAB (vecteur-matrice)
Out[7]: array([ 6.,  8.])
In [8]: dot(y,x)  # :on ferait y.'*x en MATLAB (produit scalaire)
Out[8]: 3.0
```

- Résolution d'un système linéaire $x?/Ax = b$: la fonction `x=solve(A,b)`.

```
In [1]: A=array([[2, 3],[4, 5]]); print(A)
[[2 3]
 [4 5]]
In [2]: b=array([-1, 1]); print(b)
[-1  1]
In [3]: A1=inv(A); print(A1)
[[-2.5  1.5]
 [ 2.  -1. ]]
In [4]: x=dot(A1,b); print(x)
[ 4. -3.]
In [5]: residu=dot(A,x)-b; print(residu)
[ 0.  0.]
In [6]: x=solve(A,b); print(x) # :même résultat que dot(inv(A),b)
[ 4. -3.]
```

3.3 Opérateurs et logique booléenne

Les **opérateurs relationnels** permettent de **comparer** des nombres. Les **opérateurs logiques** construisent toute la **logique booléenne**. Les constantes booléennes sont `True` et `False`.

syntaxe Python	description
$<$, $>$	strictement inférieur, strictement supérieur
$<=$, $>=$	inférieur ou égal, supérieur ou égal
$==$	égal à
$!=$	différent de
`and`	**et** logique
`or`	**ou** logique
`not(..)`	**non** logique

TABLE 7 – Opérateurs relationnels (comparaisons) et logiques (booléens).

```
In [1]: bool1=(1>2); print(bool1)
False
In [2]: bool2=(1>2 or 2<5); print(bool2)
True
```

4 Structures de contrôle

Python connaît toutes les structures de contrôle d'un langage de programmation classique. Ces structures sont surtout utilisées en mode programmation. Observez la présence obligatoire du caractère deux-points `":"` sur la ligne des mot-clés. Tout `else` d'un `if` est facultatif.

4.1 Test conditionnel (`if`)

```
if (expr. booléenne):
    instruction_alors_1
    instruction_alors_2
    ...
```

```
if (expr. booléenne):
    instruction_alors_1
    instruction_alors_2
    ...
else:
    instruction_sinon_1
    instruction_sinon_2
    ...
```

```
if (expr. booléenne1):
    instruction_11
    instruction_12
    ...
elif (expr. booléenne2):
    instruction_21
    instruction_22
    ...
elif (expr. booléenne3):
    instruction_31
    instruction_32
    ...
else:
    instruction_sinon_1
    instruction_sinon_2
    ...
```

4.2 Structures de boucle (`for, while`)

```
for k in range(10):
    instruction_1
    instruction_2
    ...
```

```
while (expression booléenne):
    instruction_1
    instruction_2
    ...
```

Attention aux boucles infinies avec le `while` (si l'expression booléenne reste fausse)...
Pour sortir prématurément d'une boucle `for` ou `while`, il est possible d'utiliser l'instruction `break`.
Remarquez qu'il n'y a ni caractère matérialisant les blocs d'instructions, ni mot clé de terminaison des structures de contrôle. Comment fait-on alors pour savoir si une instruction est dans ou hors d'un `if` ?

C'est l'**IN-DEN-TA-TION** qui le fait !

Une vertue de Python est d'obliger l'utilisateur à apporter du soin à la présentation de ses programmes : alignez donc bien vos instructions.

L'INDENTATION DU CODE EST FONDAMENTALE, ELLE CONDITIONNE TOUTE LA LOGIQUE ALGORITHMIQUE DE PYTHON !

5 Scripts et sous-programmes

En mode programmation, Python fonctionne par l'écriture de scripts (programmes principaux) complétés éventuellement de fonctions/procédures.

5.1 Les scripts

Un script est un fichier, saisi au moyen d'un éditeur de texte, contenant une suite d'instructions Python. Le nom du fichier est une chaîne de caractères composée de lettres, chiffres et/ou de caractères spéciaux, **mais surtout sans espace ni accent**. Le symbole "underscore" (_) est autorisé dans un nom pour remplacer l'espace. Les variables utilisées dans les instructions du script restent dans l'espace de travail après son exécution, elles peuvent être alors utilisées en mode interactif. Exemple de script le plus simple qui soit :

```
# -*- coding: latin-1 -*-
print("Hello World!")
```

Si ce script est sauvegardé dans le fichier de nom `hello.py`, on l'exécutera directement dans le Terminal par la commande :

```
lefevre@Mac-Ubuntu:~$ python3 hello.py
Hello World!
```

Sous IPython qu'on lance ainsi :

```
lefevre@Mac-Ubuntu:~$ ipython3 --pylab
```

on peut exécuter le script par la commande `run` suivie du nom du fichier script :

```
Python 3.4.3+ (default, Oct 14 2015, 16:09:02)
Type "copyright", "credits" or "license" for more information.
IPython 2.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
Using matplotlib backend: TkAgg

In [1]: run hello.py
Hello World!
In [2]:
```

Pour permettre des inclusions de scripts entre-eux sans risquer de conflit entre programmes principaux, il est recommandé de déclarer le programme principal comme `__main__` plutôt ainsi :

```
# -*- coding: latin-1 -*-
if __name__ == '__main__':  # :c'est bien lui le programme principal...
  print("Hello World!")
```

Le programme principal du script exécuté prévaut alors sur tous les autres.

5.2 Les fonctions et procédures

En programmation structurée, un sous-programme permet d'externaliser des tâches récurrentes. Vocabulaire : paramètre formel (=déclaratif) / effectif (=d'appel) ; paramètre d'entrée (=donnée) / de sortie (=résultat). Il existe 2 types de sous-programmes au sens algorithmique : les fonctions qui affectent une/des variable(s) retournée(s), elles font donc des calculs / les procédures qui ne retournent rien mais qui réalisent le plus souvent des affichages.... **Les paramètres d'entrée ne doivent absolument pas être saisis dans le sous-programme, mais bien à l'extérieur de celui-ci : là où il est appelé** : le faire constiturait une mauvaise compréhension

du rôle des paramètres. **Les paramètres d'entrée sont passés par références (=variables synonymes)**. Le mot-clé `def` permet la déclaration des sous-programmes.

*** Déclaration d'une fonction** :

```
def nom_fonction(e1, e2, ..., eM):
  # commentaires : bla, bla, bla
  instruction1
  instruction2
  ...
  return [s1, s2, ..., sN]
```

avec

- *e1, e2, ..., eM* : les M arguments d'entrée
- *s1, s2, ..., sN* : les N arguments de sortie

Les instructions de la fonction ont vocation à affecter toutes les variables de sortie ! Celles-ci sont alors retournées à l'espace de travail par le mot-clé `return`. Parmi les instructions de la fonction, on peut utiliser **d'autres variables** qui deviendront de facto **locales à cette fonction**, donc inconnues de l'espace de travail après exécution de celle-ci. Exemple :

```
# -*- coding: latin-1 -*-
from pylab import *
def pythagore(a,b):
  """ Fonction h = pythagore(a,b)
      calcule la longueur de l'hypoténuse d'un triangle rectangle
      variables d'entrée: a,b (=longueurs des côtés du triangle rectangle)
      variable de sortie: h   (=longueur  de  l'hypoténuse) """
  return sqrt(a**2 + b**2);
```

Pour exécuter cette fonction, on réalise un appel de la fonction en fournissant autant de paramètres effectifs (=d'appel) **que de paramètres formels** (=de déclaration) :

```
In [1]: run pythagore.py
In [2]: hyp=pythagore(3,4); print hyp
5.0
In [3]: X=array([1, 2, 3])
In [4]: Y=array([3, 2, 1])
In [5]: C=pythagore(X,Y); print C # :ça marche bien terme à terme !
[ 3.16227766  2.82842712  3.16227766]
```

Notons que les commentaires (= lignes entre `""" blabla """`) qui suivent l'entête de la fonction seront affichés si l'on demande l'aide de celle-ci par la commande `help()` :

```
In [6]: help(pythagore)

Help on function pythagore in module __main__:
pythagore(a, b)
    Fonction h = pythagore(a,b)
    calcule la longueur de l'hypoténuse d'un triangle rectangle
    variables d'entrée: a,b (=longueurs des côtés du triangle rectangle)
    variable de sortie: h   (=longueur  de  l'hypoténuse)
```

*** Déclaration d'une procédure** :

```
def nom_procedure(e1, e2, ..., eM):
  # commentaires : bla, bla, bla
  instruction1
  instruction2
  ...
```

Notez qu'il n'y a évidemment pas de `return` ici. Une **procédure** sert typiquement à **afficher** ou éventuellement à **faire un graphique**. Exemple :

```
# -*- coding: latin-1 -*-
def encadrer(chaine):
  """ Procédure encadrer(chaine)
      affiche la variable chaine (de caractères !)
      avec un très joli cadre autour ... """
  print("  * " + "="*len(chaine) + " +")
  print("  | " + chaine + " |")
  print("  + " + "="*len(chaine) + " *")
```

```
In [1]: run encadrer.py
In [2]: encadrer("Bonjour Tout Le Monde !")
  * ======================= +
  | Bonjour Tout Le Monde ! |
  + ======================= *
```

6 Graphiques

Les commandes graphiques sont apportées par la librairie Matplotlib ; elles sont presque similaires à celles de Matlab/Octave. On les présente dans ce paragraphe soit en mode interactif sous ipython, soit en scripts : ne pas oublier alors l'importation de cette librairie par la commande :

```
from pylab import *
```

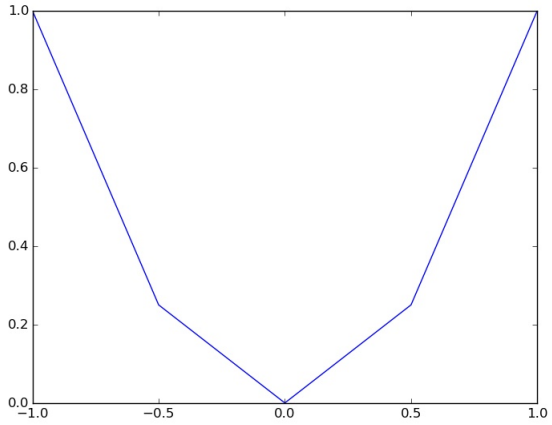
6.1 Graphiques 2D

L'appel à la commande `plot()` ouvre une fenêtre graphique, appelée figure, dans laquelle seront affichés les résultats graphiques. Le principe de la commande `plot()` est de relier deux à deux des bi-points du plan par des segments de droites. Syntaxe :

```
plot(tabx,taby)
```

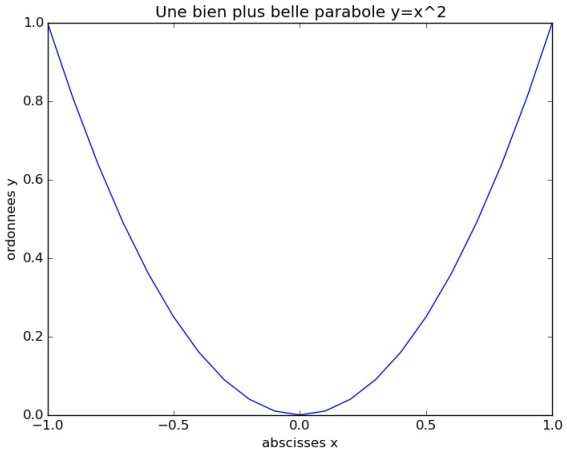
où `tabx` (pour les abscisses) et `taby` (pour les ordonnées) sont **des tableaux 1D de même dimension**. Souvent `tabx` est une discrétisation donnée d'un intervalle et `taby` est l'image des valeurs de `tabx` (terme à terme) exprimée par des fonctions mathématiques et/ou les opérateurs `+,-,*,/,**`. Exemple :

```
In [1]: x=array([-1, -0.5, 0, 0.5, 1]); y=x**2;
In [2]: print x; print y  # ... x et y ont bien autant de composantes:
[-1.  -0.5   0.    0.5   1. ]
[ 1.   0.25  0.    0.25  1. ]
In [3]: plot(x,y)
```



Pour "habiller" un graphique, on lui rajoute un titre par `title()` et une désignation des axes par `xlabel()` et `ylabel()` :

```
In [1]: x=arange(-1, 1.01, 0.1); y=x**2; plot(x,y)
In [2]: title("Une bien plus belle parabole y=x^2")
In [3]: xlabel("abscisses x"); ylabel("ordonnees y")
```



S'il y a plusieurs courbes, une légende produite par la commande `legend()` vient pour les identifier : la première chaîne de caractères s'associe au 1er `plot()` et la seconde chaîne au 2nd `plot()`... etc. Observer :

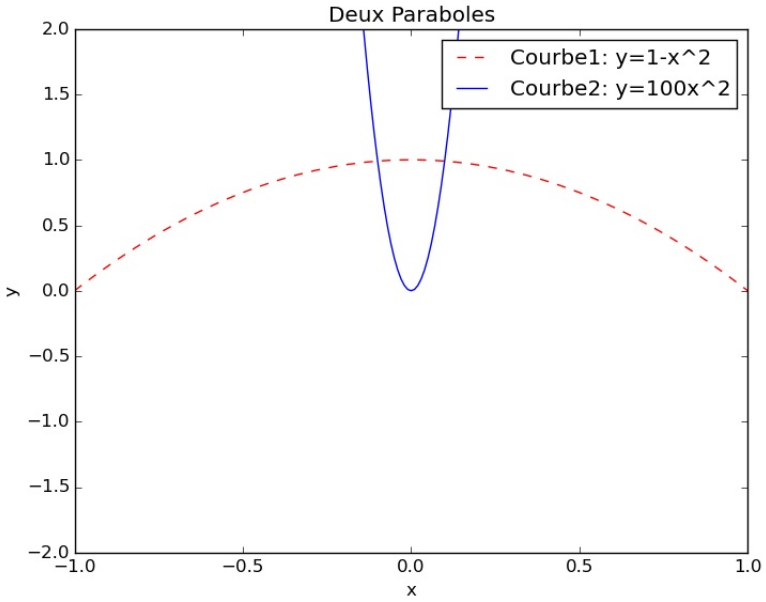
```
# -*- coding: latin-1 -*-
from pylab import *

x=arange(-1,1.001,0.01); y=1-x**2; z=100*x**2

plot(x,y,"r--");
plot(x,z,"b-") ; axis([-1, 1, -2, 2])

title("Deux Paraboles"); xlabel("x"); ylabel("y")
legend(["Courbe1: y=1-x^2", "Courbe2: y=100x^2"])

show() # --> please always remember this: the show() must go on...
```

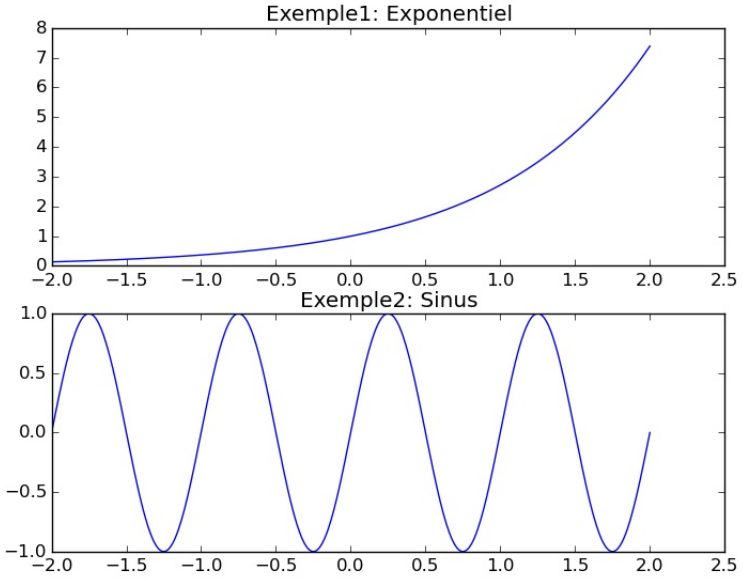


La fenêtre graphique peut être subdivisée pour tracer plusieurs graphiques mis en regard (pour comparaison de comportements par exemple...). Il faut utiliser alors la commande `subplot()`. Exemple :

```
# -*- coding: latin-1 -*-
from pylab import *

x=arange(-2, 2.001, 0.01);  y=exp(x);  z=sin(2*pi*x)

figure(1)
subplot(211); plot(x,y); title('Exemple1: Exponentiel');
subplot(212); plot(x,z); title('Exemple2: Sinus');
show()
```



commande Pylab	description
`plot(`*tabx,taby*`)`	courbe dans le plan (Oxy)
`title("`*text*`")`	ajoute un titre au graphique
`xlabel("`*text*`"); ylabel("`*text*`")`	ajoute une désignation à l'axe des abscisses et des ordonnées
`legend([`*"graph1", "graph2",...*`])`	ajoute une légende des courbes
`grid("on")`	affiche la grille de cadrillage du graphique
`grid("off")`	supprime l'affichage de la grille
`axis([`*xmin, xmax, ymin, ymax*`])`	permet de choisir l'échelle d'affichage
`clf()`	efface le dernier graphique
`close("all")`	ferme toutes les fenêtres graphiques
`figure()`	ouvre une nouvelle fenêtre graphique
`figure(n)`	rend active la fenêtre de numéro n ou crée une fenêtre graphique portant le numéro n
`subplot(`*mnp*`)`	divise la fenêtre graphique en $m \times n$ sous-fenêtres et selectionne la $p$ème sous-fenêtre (n^o en lignes)
`show()`	force l'affichage par vidage du buffer du graphique

TABLE 8 – Commandes graphiques.

6.2 Graphiques 2.5D

Nous appellerons **graphique 2.5D** un **graphique qui montre en 2D une 3ème variable** (définie en fonction d'abscisses et d'ordonnées) : cette variable peut être soit un champ de scalaires (représenté par échelle de couleurs ou bien des lignes de niveaux), soit un champ de vecteurs (représenté par "des flèches").

*** Préliminaire : discrétiser le plan 2D par une grille** :
Nous avons déjà l'habitude par `arange(a,b,pas)` de discrétiser une direction d'abscisses ou bien une direction d'ordonnées. La fonction `[X,Y]=meshgrid(x,y)` va permettre de construire deux matrices `X,Y` de même taille par réplication des discrétisations unidimensionnelles `x` et `y`. De la sorte, chaque point M_{ij} de la grille sera repéré par ses coordonnées `(X(i,j),Y(i,j))` dans le plan $\mathbb{R}^2$. Exemple :

```
In [1]: x=arange(-1, 1.001, 0.5)
In [2]: x
Out[2]: array([-1. , -0.5,  0. ,  0.5,  1. ])
In [3]: y=arange(0, 1.001, 0.5)
In [4]: y
Out[4]: array([ 0. ,  0.5,  1. ])
In [5]: [X,Y]=meshgrid(x,y)
In [6]: X
Out[6]:
array([[-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ]])
In [7]: Y
Out[7]:
array([[ 0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0.5,  0.5,  0.5],  # Exercice: dessinez cette grille de
       [ 1. ,  1. ,  1. ,  1. ,  1. ]]) # points sur votre feuille à carreaux...
```

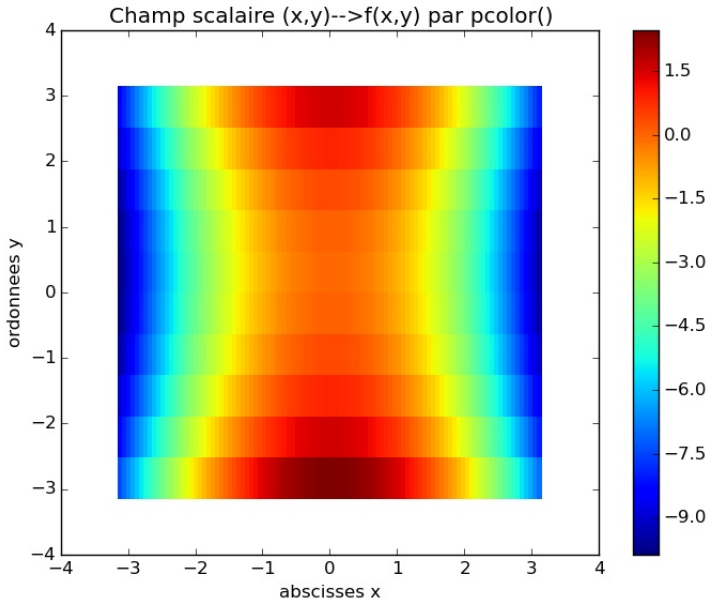
Noter que :
- `X` est une matrice ne contenant que des abscisses : ceux de `x` répliqués en direction des ordonnées.
- `Y` est une matrice ne contenant que des ordonnées : ceux de `y` répliqués en direction des abscisses.

*** Champ scalaire par échelle de couleurs** :

Nous souhaitons représenter la fonction $f(x,y) = -x^2 + \frac{y^2}{4}$ sur $[-\pi,\pi] \times [-\pi,\pi]$ en échelle de couleurs. Nous construisons la grille par `[X,Y]=meshgrid(x,y)` (vu ci-dessus), puis nous représentons le graphique 2.5D par la fonction `C=pcolor(X,Y,Z)` ; la commande `colorbar(C)` vient alors placer l'échelle de couleur en légende.

```
# -*- coding: latin-1 -*-
from pylab import *

figure(1)
x=arange(-pi, pi*1.001, 2*pi/100)
y=arange(-pi, pi*1.001, 2*pi/10 )
[X,Y]=meshgrid(x,y)
Z = -X**2 + Y**2/4
C=pcolor(X,Y,Z)     # :tracé graphique en niveau de couleurs
colorbar(C)         # :échelle de couleurs en légende
xlabel("abscisses x"); ylabel("ordonnees y")
title("Champ scalaire (x,y)-->f(x,y) par pcolor()")
show()
```

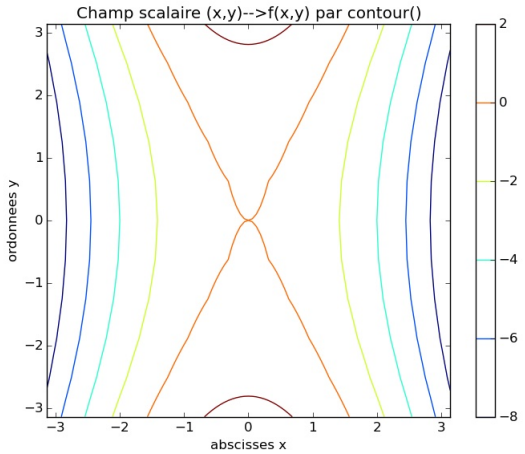


Observer que les abscisses sont finement discrétisées alors que les ordonnées ne le sont que grossièrement sur cet exemple.

*** Champ scalaire par lignes de niveaux** :
Nous souhaitons encore représenter la fonction $f(x,y) = -x^2 + \frac{y^2}{4}$ sur $[-\pi,\pi] \times [-\pi,\pi]$ mais par lignes de niveaux cette fois. Nous construisons le graphique 2.5D par la fonction `C=contour(X,Y,Z)`.

```
# -*- coding: latin-1 -*-
from pylab import *

figure(1)
x=arange(-pi, pi*1.001, 2*pi/100)
y=arange(-pi, pi*1.001, 2*pi/10 )
[X,Y]=meshgrid(x,y)
Z = -X**2 + Y**2/4
C=contour(X,Y,Z)    # :tracé graphique en lignes de niveaux
colorbar(C)         # :échelle de couleurs en légende
xlabel("abscisses x"); ylabel("ordonnees y")
title("Champ scalaire (x,y)-->f(x,y) par contour()")
show()
```



*** Champ de vecteurs** :

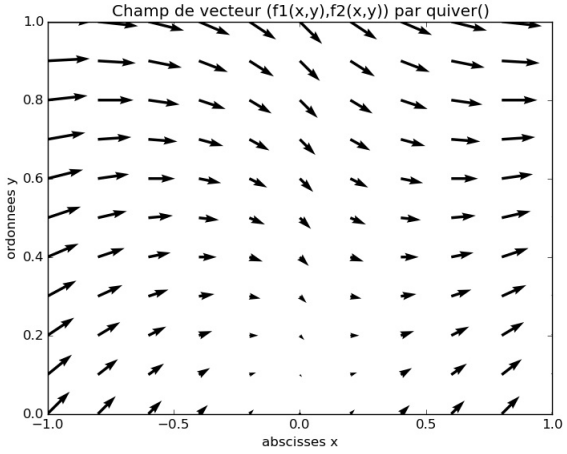
Nous souhaitons représenter sur $[-1,1]\times[0,1]$ le champ de vecteurs suivant à valeurs dans $\mathbb{R}^2$:

$$\begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix} = \begin{pmatrix} |x|+|y| \\ |x|-|y| \end{pmatrix}$$

D'une manière similaire à ce qui précède, ce sera réalisé par la commande `quiver(X,Y,F1,F2)` :

```
# -*- coding: latin-1 -*-
from pylab import *

figure(1)
x=arange(-1, 1.001, 0.2)
y=arange( 0, 1.001, 0.1)
[X,Y]=meshgrid(x,y)
quiver(X,Y,abs(X)+abs(Y),abs(X)-abs(Y)) # :tracé graphique du champ de vecteurs
xlabel("abscisses x"); ylabel("ordonnees y")
title("Champ de vecteur (f1(x,y),f2(x,y)) par quiver()")
show()
```



6.3 Graphiques 3D

Pour créer un graphique 3D, il est nécessaire d'introduire l'entête suivante au fichier source :

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

*** Préliminaire : créer l'objet "Axes 3D"** :

Pour réaliser un graphique dans $\mathbb{R}^3$, il va falloir au préalable créer **un objet "axes 3D"** par la commande :

```
ax=subplot(111,projection="3d")
```

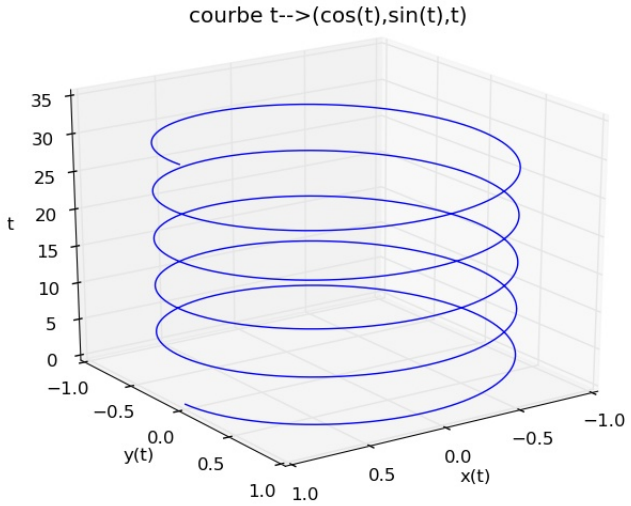
Certaines fonctions (alors appelées **méthodes**) découleront de cette objet "axes 3D" `ax`, qu'il est donc important de ne pas oublier.

*** Tracé d'une courbe paramétrée dans l'espace** :

Pour tracer le graphique dans $\mathbb{R}^3$ de la courbe paramétrée $t \mapsto (\cos(t), \sin(t), t)$ sur $[0, 10\pi]$, nous utilisons la méthode `ax.plot3D(x,y,z)` de l'objet `ax` "axes 3D". Notez aussi que la fonction `zlabel()` de Matlab/Octave n'existe pas en python, mais on la retrouve tout de même grâce à la méthode `ax.set_zlabel()` ;

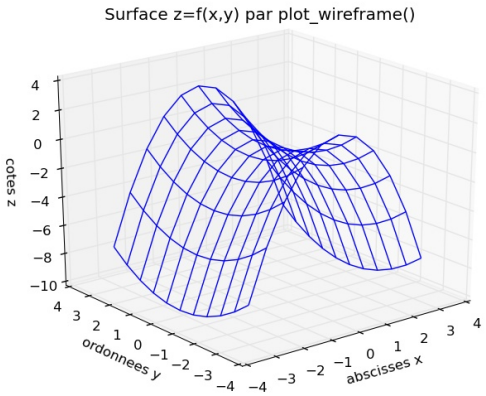
```
# -*- coding: latin-1 -*-
from pylab import *
from mpl_toolkits.mplot3d.axes3d import Axes3D

figure(1)
ax=subplot(111,projection="3d") # :<-- ne pas l'oublier pour un graphique 3D
t=arange(0.0, 10*pi, 0.01)
x=cos(t); y=sin(t); z=t
ax.plot3D(x,y,z)                # :<-- méthode de ax
xlabel("x(t)"); ylabel("y(t)")
ax.set_zlabel("t")              # :<-- méthode de ax
title("courbe t-->(cos(t),sin(t),t)")
show()
```



*** Tracé d'une surface** :
Reprenons notre surface représentée par la fonction $z = f(x, y) = -x^2 + \frac{y^2}{4}$ sur $[-\pi, \pi] \times [-\pi, \pi]$.
Notant $M_{ij} = \begin{bmatrix} X_{ij} \\ Y_{ij} \\ Z_{ij} \end{bmatrix} \in \mathbb{R}^3$; la surface issue de la commande `ax.plot_wireframe(X,Y,Z)` reliera les points M_{ij} avec $M_{i-1,j}, M_{i+1,j}, M_{i,j-1}, M_{i,j+1}$ dans une grille cartésienne 3D.

```
# -*- coding: latin-1 -*-
from pylab import *
from mpl_toolkits.mplot3d.axes3d import Axes3D
figure(1); ax=subplot(111,projection="3d")
x=arange(-pi, pi*1.001, 2*pi/10);
y=arange(-pi, pi*1.001, 2*pi/10)
[X,Y]=meshgrid(x,y)
Z= -X**2 + Y**2/4
ax.plot_wireframe(X,Y,Z)    # :surface en grille
xlabel("abscisses x"); ylabel("ordonnees y")
ax.set_zlabel("cotes z")
title("Surface z=f(x,y) par plot_wireframe()")
show()
```



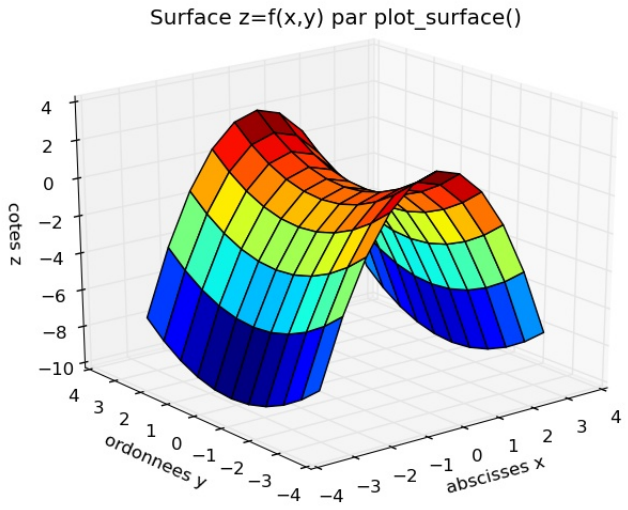
Une autre commande un peu plus complexe :

`ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap=cm.jet)`

permet cette fois-ci de colorier les facettes de la grille en niveaux de couleurs calorimétriques (bleu=valeurs basses, rouge=valeurs hautes) :

```
# -*- coding: latin-1 -*-
from pylab import *
from mpl_toolkits.mplot3d.axes3d import Axes3D

figure(1)
ax=subplot(111,projection="3d")
x=arange(-pi, pi*1.001, 2*pi/10)
y=arange(-pi, pi*1.001, 2*pi/10)
[X,Y]=meshgrid(x,y)
Z= -X**2 + Y**2/4
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap=cm.jet) # :surface colorée
xlabel("abscisses x"); ylabel("ordonnees y")
ax.set_zlabel("cotes z")
title("Surface z=f(x,y) par plot_surface()")
show()
```



commande Pylab	description
`meshgrid`	grille de discrétisation 2D
`colorbar`	échelle de couleur
`contour`	lignes de niveaux
`pcolor`	champ de scalaires
`quiver`	champ de vecteurs
`ax.plot3D`	courbe dans l'espace 3D
`ax.plot_surface`	surface colorée dans l'espace 3D
`ax.plot_wireframe`	grille dans l'espace 3D
`ax.set_zlabel`	label de l'axe Oz

TABLE 9 – Commandes pour les graphiques 2.5D et 3D.

7 Fichier texte de données

*** Enregistrement d'un fichier au format texte/ASCII** :
Il est effectué par la procédure `savetxt()` :

`savetxt(`*nom_fich, var*`)`

où *nom_fich* est une chaîne de caractères contenant le nom du fichier pour l'enregistrement, et *var* est la variable (tableau 1D, 2D) à enregistrer. Ce fichier enregistré au format texte est éditable par n'importe quel éditeur de texte, donc lisible (pour contrôle visuel par exemple).

*** Lecture d'un fichier au format texte/ASCII** :
Elle est réalisée par la fonction `loadtxt()` :

tab `= loadtxt(`*nom_fich*`)`

où *nom_fich* est une chaîne de caractères contenant le nom du fichier à lire, et *tab* est le tableau créé en mémoire qui résultera de la lecture du fichier. Exemple :

```
In [1]: A=ones((4,2))
In [2]: A
Out[2]:
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
In [3]: savetxt("FichierA.txt",A)
In [4]: %reset -f -s  # :on nettoie l'espace de travail
In [5]: whos
Interactive namespace is empty.

In [6]: B=loadtxt("FichierA.txt")
In [7]: B
Out[7]:
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

8 Espaces de noms ou pas ?

- Écrire `import numpy` en entête obligera l'usage par exemple de la fonction cosinus par l'expression `numpy.cos(2*numpy.pi)` : on va la chercher dans l'espace de noms `numpy`.
- Écrire `import numpy as np` en entête obligera l'usage de cette fonction cosinus par l'expression `np.cos(2*np.pi)` : création d'un espace de noms nommé `np`.
- Écrire `from numpy import *` en entête permet l'usage de cette fonction cosinus directement `cos(2*pi)` : sans espace de noms, par contre on vient encombrer l'espace de travail de tout l'ensemble des fonctions de la librairie. Pour permettre encore cette expression simplifiée, on peut faire un import sélectif par : `from numpy import cos, pi`.
- Si `toto.py` est un fichier de fonctions python, vous pouvez l'intégrer à un autre fichier contenant le programme principal en spécifiant dans ce fichier la commande `from toto import *`, ou bien encore `import toto as tt` (création de l'espace de noms `tt`), ou bien encore `import toto` (espace de noms `toto` par défaut)... Comme vu ci-dessus avec `numpy` !

À vous donc de choisir si vous souhaitez faire usage ou pas d'un espace de noms !

9 Annexe 1 : le "help plot" de Matlab (très similaire en ipython)

```
>> help plot

plot   Linear plot.
  plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix, then the vector
  is plotted versus the rows or columns of the matrix, whichever line up.
  If X is a scalar and Y is a vector, length(Y) disconnected points are plotted.

  plot(Y) plots the columns of Y versus their index. If Y is complex, plot(Y) is
  equivalent to plot(real(Y),imag(Y)).
  In all other uses of PLOT, the imaginary part is ignored.

  Various line types, plot symbols and colors may be obtained with plot(X,Y,S) where S
  is a character string made from one element from any or all the following 3 columns:

        b     blue          .     point              -     solid
        g     green         o     circle             :     dotted
        r     red           x     x-mark             -.    dashdot
        c     cyan          +     plus               --    dashed
        m     magenta       *     star             (none)  no line
        y     yellow        s     square
        k     black         d     diamond
                            v     triangle (down)
                            ^     triangle (up)
                            <     triangle (left)
                            >     triangle (right)
                            p     pentagram
                            h     hexagram

  For example, plot(X,Y,'c+:') plots a cyan dotted line with a plus at each data point;
  plot(X,Y,'bd') plots blue diamond at each data point but does not draw any line.

  plot(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by the (X,Y,S) triples,
  where the X's and Y's are vectors or matrices and the S's are strings.

  For example, plot(X,Y,'y-',X,Y,'go') plots the data twice, with a solid yellow line
  interpolating green circles at the data points.

  The plot command, if no color is specified, makes automatic use of the colors
  specified by the axes ColorOrder property. The default ColorOrder is listed in the
  table above for color systems where the default is blue for one line, and for multiple
  lines, to cycle through the first six colors in the table.  For monochrome systems,
  plot cycles over the axes LineStyleOrder property.

  If you do not specify a marker type, plot uses no marker.
  If you do not specify a  line style, plot uses a solid line.
```

10 Annexe 2 : un programme python complet

```
from pylab import *
from time import time

# ============================================================
# Chronométrage d'Algorithmes pour le produit Matrice-Vecteur
# Copyright: François Lefèvre, 2021
# ============================================================
# Résultat des temps d'exécution:
# temps CPU (par lignes  ) = 0.20 s, vitesse = 2383 Mflops
# temps CPU (par colonnes) = 3.05 s, vitesse =  160 Mflops
# BILAN: *** privilégier en Python les ALGOS PAR LIGNES ! ***
# ============================================================

# ============================================================
def matvec_lig(A,x): # :produit MATrice-VECteur par LIGnes
  [m,n]=shape(A)
  y=zeros(m)
  for i in range(m):
    y[i] = dot(A[i,:],x)
  return y

# ============================================================
def matvec_col(A,x): # :produit MATrice-VECteur par COLonnes
  [m,n]=shape(A)
  y=zeros(m)
  for j in range(n):
    y += x[j]*A[:,j]
  return y

# ============================================================
def affiche_temps(chaine, tps, cmpt): # :C'EST une PROCEDURE !
  print(" temps CPU ("+chaine+") = "+format(tps,"4.2f")+" s,"+\
        " vitesse = "+format(cmpt/tps/1024/1024,"4.0f")+" Mflops")

# ============================================================
# le programme principal:
# ============================================================
if __name__ == '__main__':
  n=16000
  x=rand(n)
  A=ones((n,n))
  cmpt=2*n**2   # :complexité algorithmique

  t0=time(); y=matvec_lig(A,x); t=time()
  affiche_temps("par lignes  ",t-t0,cmpt)

  t0=time(); y=matvec_col(A,x); t=time()
  affiche_temps("par colonnes",t-t0,cmpt)
# ============================================================
```

Index

Références

[App] http://www.apprendre-python.com

[IPy] https://ipython.org/documentation.html

[Mpl] http://matplotlib.org/contents.html

[Pyt] https://www.python.org/doc/

[Swi] https://inforef.be/swi/download/apprendre_python3_5.pdf