Table des matières

1 Modules scientifiques de python

Les modules pour le calcul scientifique sous python :

1. numpy : Outils pour créer, manipuler, et appliquer de nombreuses opérations sur des tableaux de nombres.
   Aide en ligne : http://docs.scipy.org/doc/numpy/reference/ (en anglais).
2. scipy : Fonctions mathématiques s'appliquant aux tableaux générés par numpy. Elle contient des opérations spécifiques (algèbre linéaire, intégration, statistiques,...) de manipulation de tableaux, de plus haut niveau que celles de numpy.
   Aide en ligne : http://docs.scipy.org/doc/scipy/reference/ (en anglais).
3. matplotlib.pyplot : Permet le tracé de graphes de fonctions.
   Aide en ligne : http://matplotlib.org/users/pyplot_tutorial.html (en anglais).

2 Tableaux unidimensionnels avec numpy

• Le module numpy permet de créer, de manipuler, des tableaux numériques, homogènes, non-redimensionnables et de leur appliquer des opérations mathématiques courantes.

• **La fonction** array() permet de créer un tableau, ou array, à partir d'un tableau python c'est à dire d'une liste de listes ou tuples de nombres :

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4])
>>> A
array([1, 2, 3, 4])
>>> A[0] , A[-1]
 (1, 4)
```

• Toutes les opérations sur les listes et séquences python, en dehors de celles qui redimensionnent sont possible sur un tableau numpy.

• **La fonction** arange() crée un tableau de façon assez analogue à la fonction range(), à ceci près que les coefficients ne sont pas forcément entiers :

```
>>> v = np.arange(0, 1.5, 0.5)
>>> v
array([ 0., 0.5, 1. ])
>>> 2*v
array([ 0., 1., 2. ])
>>> (2*v) ** 2 + 100
array([ 100., 101., 104. ])
```

On peut appliquer sur les tableaux les opérations mathématiques ainsi que les fonctions mathématiques du module numpy ; elles s'appliquent terme à terme.

• **La fonction** linspace(a, b, n) crée le tableau des n valeurs régulièrement espacées prises entre a et b, tous deux inclus.

C'est à dire le tableau de la subdivision régulière de l'intervalle $[a, b]$ par n points (ou par $n-1$ segments). Exemple :

```
>>> v = np.linspace(0,1,10)
>>> v
array([ 0. , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
    0.55555556, 0.66666667, 0.77777778, 0.88888889, 1. ])
```

• numpy contient aussi toutes les fonctions mathématiques (aussi présentes dans math) :

```
>>> np.cos(v)
array([ 1. , 0.99383351, 0.97541009, 0.94495695, 0.90284967,
    0.84960756, 0.78588726, 0.71247462, 0.63027505, 0.54030231])
```

Mais elles sont universelles : elles s'appliquent terme à terme à un tableau.

Tableaux (uni-dimensionnels) sous numpy	
`array(liste)`	crée un tableau à partir d'une liste ou séquence `liste`
`arange(a,b,k)`	crée le tableau de tous les `a+k.N` entre `a` (inclu) et `b` (exclu).
`linspace(a,b,n)`	crée le tableau des `n` valeurs régulièrement espacées entre `a` et `b` (inclus)
`zeros(p)`	crée un tableau de taille `p` rempli de zéros
`mean()`	retourne la moyenne d'un tableau
`size()`	retourne le nombre d'éléments d'un tableau

3 Fonction `vectorize()` de `numpy`

• En général on peut appliquer une fonction à un tableau `numpy` :

```
>>> def f(x,y):
...     return x * y
>>> X = np.array([1, 2, 3, 4])
>>> f(X,2)
array([2, 4, 6, 8])
```

• Mais ce n'est pas possible avec toutes les fonctions. Par exemple :

```
>>> def min(x,y):
...     if x < y: return x
...     else: return y
>>> X = np.array([1, 2, 3, 4])
>>> f(X,2)
ValueError : The truth value of an array with more than one element is
ambiguous
```

• Pour cela il faut 'vectoriser' la fonction à l'aide de `np.vectorize()`. Pour une fonction vectorisée, lorsqu'un paramètre est un tableau de valeurs, elle retourne le tableau des résultats obtenus pour chacune de ces valeurs.

```
>>> min_v = np.vectorize(min)    # min_v() est la version vectorisée de min()
>>> min_v(X,2)
array([1, 2, 2, 2])
>>> min_v(X, X)
array([1, 2, 3, 4])
```

4 Tableaux bi-dimensionnel et matrices avec `numpy`

4.1 Tableaux bi-dimensionnels avec `array()`

• **La fonction `array()`** permet aussi de créer un tableau bi-dimensionnel, à partir d'une liste de listes de nombres (de même longueur) :

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> A
 array([[1, 1, 1],
        [0, 1, 1],
        [0, 0, 1]])
>>> print(A[0][0], A[0,0])
1 1
```

• On accède aux éléments d'un tableau comme en `python`, grâce à deux indices entre crochets : `A[0][0]`, ou plus simplement, en séparant les deux indices d'une virgule : `A[0,0]`.

• On peut appliquer du slicing :

```
>>> A[0,:]    # Première ligne
array([1, 1, 1])
>>> A[:,1]    # Deuxième colonne
array([[1], [1], [0]])
```

• Le type d'un tableau s'obtient grâce à la fonction `shape()`, son nombre d'élément grâce à `size()`. La fonction `reshape()` permet de changer la forme (=type) d'une matrice :

```
>>> L=np.arange(0,10)
>>> np.reshape(L,(2,5))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> L
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> M = np.reshape(L,(2,5)) ; np.shape(M)
(2, 5)
```

• La fonction `transpose()` permet d'obtenir la transposée :

```
>>> np.transpose(A)
array([[1, 0, 0],
       [1, 1, 0],
       [1, 1, 1]])
```

• La fonction `rank()` permet d'obtenir le rang :

```
>>> np.rank(A)
3
```

• Le produit matriciel s'obtient à l'aide de la fonction `dot()`.

Un vecteur peut s'écrire à l'aide d'un tableau uni-dimensionnel. Par exemple : `v= np.arange(0,3)` ou `v=array([0, 1, 2])` mais aussi comme la matrice colonne v=array([[0],[1],[2]]).

Exemple : avec la matrice 3×3 : $A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ et le vecteur $v = (0,\ 1,\ 2)$.

Le produit $A \times v$ est défini égal à $(3, 3, 2)$; le produit $v \times A$ n'est pas défini. Mais ${}^t v \times A = (0,\ 1,\ 3)$.

```
>>> v = array([0,1,2])
>>> np.dot(A,v)
array([3, 3, 2])
>>> np.dot(v,A)
array([0, 1, 3])
```

On le voit, lorsque la multiplication est impossible, pour $v.A$, la fonction `dot()` effectue $A \times {}^t v$, qui a retourné pour résultat $(0, 1, 3)$. On pourra saisir des matrices lignes sous forme de vecteurs (leur transposée). Ce fonctionnement est commun aux logiciels de calcul sur des matrices (`matlab`, `scilab`).

Pour effectuer le produit scalaire de 2 "vecteurs" utiliser la fonction `vdot()`.

• Attention la multiplication `'*'` entre tableaux s'effectue terme à terme. En particulier l'opération `A ** 2` correspond à une élévation au carré terme à terme :

```
>>> A ** 2
array([[1 1 1]
       [0 1 1]
       [0 0 1]])
>>> (2*A) ** 2
array([[4 4 4]
       [0 4 4]
       [0 0 4]])
```

• Pour élever `A` au carré faire plutôt :

```
>>> np.dot(A,A)
array([[1, 2, 3],
       [0, 1, 2],
        [0, 0, 1]])
```

• Pour élever une matrice carrée A à une puissance n :

```
>>> B = A
>>> for i in range(1,n):
       B = np.dot(A,B)    # à la sortie de boucle B contient A^ n
```

• De même l'inversion `A ** -1` se fait terme à terme. Elle produira ici une erreur (division par 0) alors même que la matrice `A` est inversible.

4.2 Le sous-module `linalg`

• Certaines fonctions plus spécifiques à l'algèbre linaire sont disponibles dans le sous-module `linalg` de `numpy`. Toutes les fonctions de ce sous-module devront être saisies avec le préfixe `np.linalg.`.

• Pour l'inversion de matrice : utiliser la fonction `inv()` du sous-module `linalg` :

```
>> np.linalg.inv(A)
array([[ 1., -1., 0.],
       [ 0., 1., -1.],
       [ 0., 0., 1.]])
```

• le sous-module `linalg` contient aussi, entre autres :

1. la fonction `det()` qui retourne le déterminant d'une matrice.

2. La fonction `solve(A,b)` qui résout le système linéaire de matrice `A` et de vecteur second membre `b`.

Exemple : résoudre le système linéaire :

$$\begin{cases} x & -y & +z & = & 1 \\ -x & +y & +z & = & 1 \\ 2x & -y & -z & = & 0 \end{cases}$$

```
>>> A=np.array([[1,-1,1],[-1,1,1],[2,-1,-1]])
>>> b= np.array([1,1,0])
>>> np.linalg.solve(A,b)    # avec solve()
array([ 1., 1., 1.])
```

4.3 La classe `matrix`

• On peut aussi définir une matrice à partir d'une liste ou d'une chaine de caractère qui comprenne
la multiplication matricielle, grâce à `matrix` :

```
>>> A = np.matrix([[1,1,1],[0,1,1],[0,0,1]])
>>> A
matrix([[1, 1, 1],
        [0, 1, 1],
        [0, 0, 1]])
>>> B = np.matrix('1 2 3; 2 3 4; 4 5 6')
>>> B
matrix([[1, 2, 3],
        [2, 3, 4],
        [4, 5, 6]])
>>> A * B
matrix([[ 7, 10, 13],
        [ 6, 8, 10],
        [ 4, 5, 6]])
```

Le produit matriciel est correct. L'inversion aussi :

```
>>> A ** -1
matrix([[ 1., -1., 0.],
        [ 0., 1., -1.],
        [ 0., 0., 1.]])
```

Attention, erreur lorsque la matrice n'est pas inversible.

5 Fonctions principales de numpy

TABLEAUX :	
`array(l)`	crée un tableau à partir d'une liste `l`
`arange(a,b,k)`	crée un vecteur dont les coefs sont les `a+k.N` entre `a` (inclu) et `b` (exclu).
`linspace(a,b,n)`	crée un vecteur de `n` valeurs régulièrement espacées entre `a` et `b` (inclus)
`zeros(p)`	crée un tableau de taille `p` rempli de zéros
`ones(p)`	crée un tableau de taille `p` rempli de uns
`vdot()`	pour effectuer un produit scalaire de 2 "vecteurs"
`mean()`	valeur moyenne d'un tableau
`size()`	pour obtenir le nombre d'éléments d'un tableau
`reshape()`	pour redimensioner un tableau
`zeros((p,q))`	crée un tableau de taille `(p,q)` rempli de zéros
`ones((p,q))`	crée un tableau de taille `(p,q)` rempli de uns
`dot()`	pour effectuer un produit matriciel de 2 matrices
`transpose()`	pour transposer une matrice
`rank()`	rang d'une matrice
`shape()`	pour obtenir la taille d'un tableau (= type d'une matrice)
Dans `linalg` :	
`inv()`	inversion d'une matrice
`det()`	déterminant d'une matrice
`solve(A,b)`	résolution du système linéaire $A.X = b$
`meshgrid()`	crée deux matrices de même taille à partir de deux listes
`vectorize()`	pour 'vectoriser' une fonction (la force à s'appliquer à un tableau)
AUTRES :	
`matrix()`	pour créer un objet -matrice- de la classe `matrix`
`poly1d()`	pour créer un objet -polynome- de la classe `poly1d`

`numpy` contient aussi constantes et fonctions mathématiques usuelles.

6 Tracé avec matplotlib.pyplot

• Pour le simple tracé de courbes nous n'utiliserons que le sous-module `pyplot`, importé, avec alias, à l'aide de la commande :

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

cf. documentation à : `http://www.matplotlib.org`.

• Les fonctions essentielles de `pyplot` sont :

1. `plot()` pour le tracé de points, de courbes, et
2. `show()` pour afficher le graphique créé.

6.1 Syntaxe de `plot()`

• Utiliser `plot()` avec :

1. en 1^{er} argument la liste des abscisses,
2. en 2^{eme} argument la liste des ordonnées,
3. en 3^{eme} argument (optionnel) le motif des points :
   (a) `'.'` pour un petit point,
   (b) `'o'` pour un gros point,
   (c) `'+'` pour une croix,
   (d) `'*'` pour une étoile,
   (e) `'-'` points reliés par des segments
   (f) `'--'` points reliés par des segments en pointillés
   (g) `'-o'` gros points reliés par des segments (on peut combiner les options)
   (h) `'b'`, `'r'`, `'g'`, `'y'` pour de la couleur (bleu, rouge, vert, jaune, etc...)
   (i) cf. `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot`.

6.2 Exemples de tracés

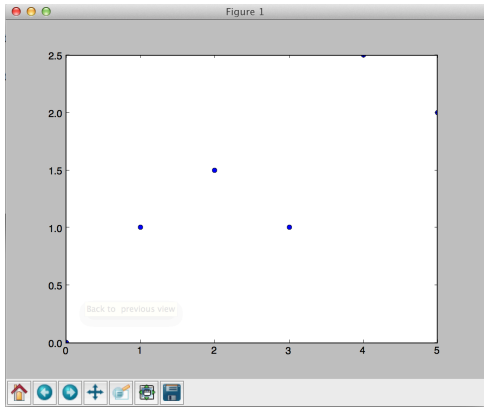
• Exemple : pour le tracé d'un nuage de points :

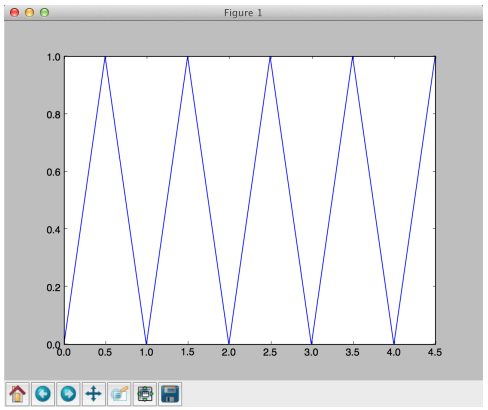
```
>>> import matplotlib.pyplot as plt
>>> abs = [0, 1, 2, 3, 4, 5]
>>> ord = [0, 1, 1.5, 1, 2.5, 2]
>>> plt.plot(abs, ord, 'o')
>>> plt.show()
```

produit un graphique (au format .png).

• Exemple : pour le tracé d'une ligne brisée :

```
>>> import matplotlib.pyplot as plt
>>> abs = [n/2. for n in range(10)]
>>> ord = [n % 2 for n in range(10)]
>>> plt.plot(abs,ord,'-b')
>>> plt.show()
```



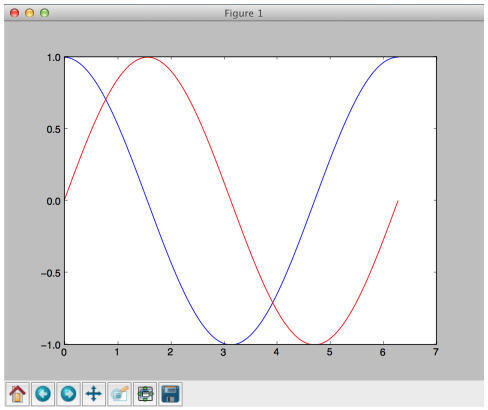


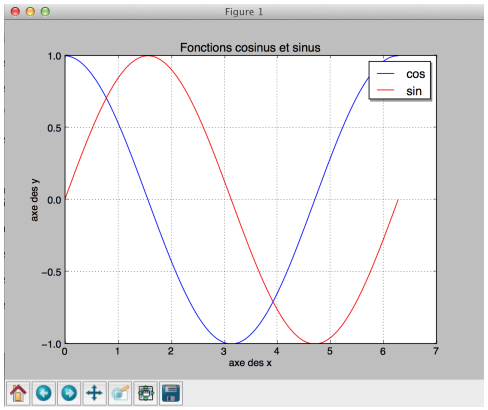
• Exemple : pour le tracé de courbes représentatives de fonctions réelles :

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np        # pour linspace() et les fonctions mathématiques
>>> X = np.linspace(0, 2*np.pi, 255)        # X = 255 pts régulièrement espacés
>>> Ycos = np.cos(X)                        # image directe de X par cos
>>> Ysin = np.sin(X)                        # image directe de X par sin
>>> plt.plot(X,Ycos,'b')                     # tracé de la courbe de cos en
bleu
>>> plt.plot(X,Ysin,'r')                     # tracé de la courbe de sin en
rouge
>>> plt.show()
```

• On améliore le tracé en remplissant quelques options avant de la sauvegarder (au format .png dans le répertoire utilisateur).

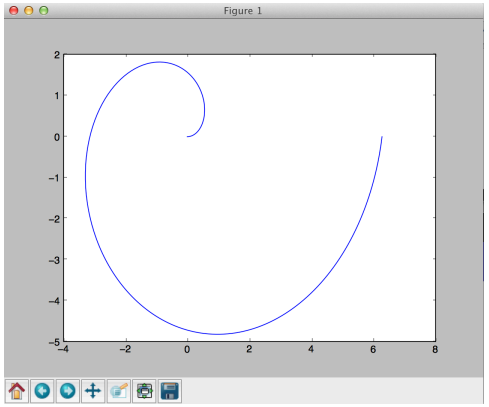
```
>>> plt.plot(X, Ycos, 'b', X, Ysin, 'r')    # Tracé simultané des 2 courbes
>>> plt.grid(True)                # Affiche la grille
>>> plt.legend(('cos','sin'), 'upper right', shadow = True)    # Légende
>>> plt.xlabel('axe des x')        # Label de l'axe des abscisses
>>> plt.ylabel('axe des y')        # Label de l'axe des ordonnées
>>> plt.title('Fonctions cosinus et sinus')    # Titre
>>> plt.savefig('ExempleTrace')    # sauvegarde du fichier ExempleTrace.png
>>> plt.show()
```





• On peut tout aussi bien tracer des courbes paramétrées.

```
>>> T = np.linspace(0,2*np.pi,255)        # paramètre t
>>> X = T * np.cos(T)          # x(t) = t.cos(t)
>>> Y = T * np.sin(T)          # y(t) = t.sin(t)
>>> plt.plot(X,Y,'b')          # Tracé de la courbe paramétrée {(x(t),y(t))}
>>> plt.show()
```



6.3 Conversion Tableau/Image avec `imshow()`

• Le module `matplotlib.pyplot` contient la fonction `imshow()` qui permet d'afficher une image à partir d'un tableau numérique.

6.4 Utilisation de la fonction `meshgrid()` de `numpy`

Commençons par utiliser la fonction **`meshgrid()`** du module **`numpy`**. Elle prend en argument deux tableaux unidimensionnel `L` et `C` et retourne deux matrices , de même type :
nombre de colonne = `len(L)`, nombre de lignes = `len(C)`.

```
X, Y = np.meshgrid(L,C)
```

`X` aura toutes ses lignes identiques à `L` ; `Y` aura toutes ses colonnes identiques à C.

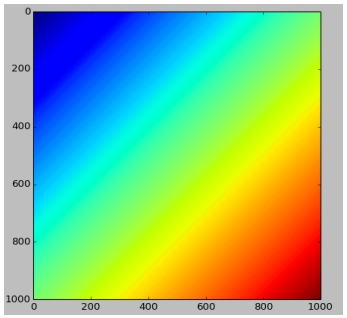
L'avantage c'est que tous les couples $(x, y) \in L \times C$ sont en bijection avec tous les couples $(X(i, j), Y(i, j))$ lorsque i, j varient.

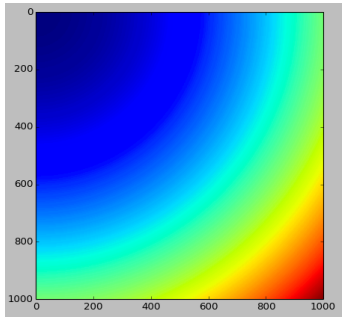
Par exemple :

```
>>> L = [1,2]
>>> C = [3,4,5]
>>> X, Y = meshgrid(L,C)
>>> X
[[1, 2],
 [1, 2],
 [1, 2]]
>>> Y
[[3, 3],
 [4, 4],
 [5, 5]]
```

6.5 Affichage d'un tableau avec `imshow()` de `pyplot`

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0,100,0.1)
y = np.arange(0,100,0.1)
X, Y = np.meshgrid(x,y)
plt.imshow(X+Y)
plt.imshow(X**2+Y**2)
```





6.6 Tracé d'une surface dans l'espace

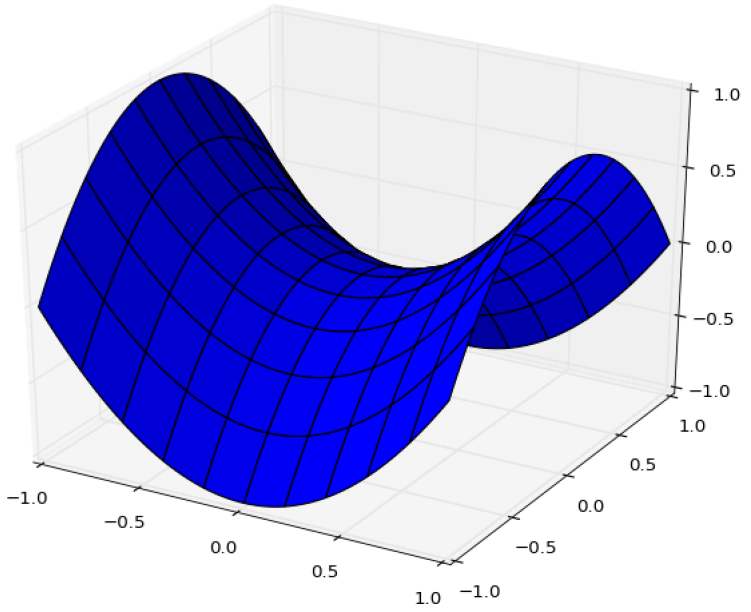
On peut représenter graphiquement une fonction $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ par la surface dans l'espace muni d'un repère des points de coordonnées :

$$\mathscr{S}_f = \left\{(x, y, z) \in \mathbb{R}^3 \mid (x, y) \in \mathscr{D}_f,\ z = f(x, y)\right\}$$

Pour cela on utilisera **`pyplot`** avec la fonction **`Axes3D`** importée du module `mpl_toolkits.mplot3d`, et sa fonction `plot_surface` :

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig3d = Axes3D(plt.figure())    # Création d'une figure fig3d en 3D
f = lambda x,y : x**2-y**2    # Définition de la fonction f(x,y) = x^2 - y^2
x = np.linspace(-1,1,100)
y = np.linspace(-1,1,100)
X, Y = np.meshgrid(x,y)    # Matrices des abscisses et ordonnées
Z = f(X,Y)
fig3d.plot_surface(X,Y,Z)    # Tracé
plt.show()
```



6.7 Tracé d'une courbe dans l'espace

Enfin pour tracer une courbe dans l'espace on utilise encore la fonction `Axes3D` importée de `mpl_toolkits.mplot3d`. Par exemple pour représenter la courbe :

$$\begin{array}{rcl} [0,4\pi] & \longrightarrow & \mathbb{R}^3 \\ t & \longmapsto & (\cos(t),\sin(t),t) \end{array}$$

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig3d = Axes3D(plt.figure(2))
T = np.linspace(0,4*np.pi,1000)
X = np.cos(T)
Y = np.sin(T)
Z = T
fig3d.plot(X,Y,Z)
plt.show()
```