Introduction et Rappels à Python Scientifique pour PhyExp

Ces notes concernent Python 2.6-2.7. La version la plus récente Python 3. introduit quelques modifications dans le langage qui n'ont pas encore été répercutées dans la plupart des librairies scientifiques. Pour des raisons de compatibilité, on restera dans les versions 2.x de Python. [1]

1 Installer Python

Python est un langage de programmation interprété très flexible et efficace dans de nombreux domaines d'applications. L'utilisation facile de modules permet d'y rajouter des fonctions spécialisées (scientifiques, bases de données, jeux, administration système, internet, etc...). Le langage est en licence libre et fonctionne sur la plupart des plateformes (Window, MacOs, Unix, ...).

Etant donné la versatilité et la multitude des domaines d'applications de Python, il n'existe pas un environnement de programmation unique pour y travailler (pas d'interface utilisateur unique). Cette versatilité peut-être déroutante au débutant, c'est le pendant de la puissance du système. Pour un usage scientifique vous pouvez installer une distribution pré-préparée, déjà accompagnée des principaux modules de science Python comme :

La distribution *Enthought* Python `http://enthought.com/products/epd.php` qui regroupe de nombreuses librairies scientifiques que vous pourrez avoir besoin. Une version minimale gratuite disponibles pour tous rassemble l'essentiel, déjà très suffisant. Etudiants et universitaires ont accès gratuitement à la distribution complète. Choix le plus simple sur MacOS.

La distribution PythonXY `http://code.google.com/p/pythonxy/`pour Windows et Linux uniquement, rassemble les librairies scientifiques principales et un environnement de développement et d'exécution intégré proche de matlab. Peut-être le meilleurs choix sur Windows.

L'environnement de calcul scientifique `SAGE` `http://www.sagemath.org/fr/` existant pour tout OS combine la puissance de nombreux programmes scientifiques libres dans un système commun incluant et basée sur Python et proposant aussi une interface graphique d'utilisation avancée. `SAGE` est un système qui étend fortement Python pour le calcul scientifique et les mathématiques (y compris calcul symbolique) et offre des possibilités non conventionnelles dans Python standard. Très bon choix universel, mais moins standard.

Si on a besoin de Python sans pouvoir l'installer, `http://www.pythonanywhere.com/` est un service proposant de travailler à distance via un navigateur sur un serveur.

2 Utiliser Python

Suivant le type d'installation sur votre système le lancement d'une session Python peut se faire de manière différente et suivant les préférences de chacun. La manière de base la plus rudimentaire mais la plus sûre dans toutes les circonstances (par exemple aussi sur un ordinateur distant via un conduit internet) est de travailler dans une console textuelle (un `shell` unix, dit encore `terminal`). Alternativement, toute distribution Python inclut aussi un éditeur et environnement d'exécution graphique

1. Notes préparées par M. Santolini et S. Bottani adaptées du cours PhyNum M1 PMA 2012.

nommé IDLE. Ou bien vos distributions vous donnent accès à d'autres interfaces (spyder sous PyhtonXY, notebook avec SAGE, mode phyton de emacs). Bref, vous avez le choix !
Sous Unix/Linux/BSD/MacOS il suffit d'ouvrir une fenêtre de shell (nommé aussi terminal), et lancer l'instruction python (ou ipython si installé, pour plus de fonctionnalités) pour ouvrir une session textuelle.

2.1 Comme une super calculatrice : mode interactif

Pour utiliser au mieux Python, vous pouvez lancer une session interactive, c'est-à-dire une session durant laquelle vous exécutez à la main des instruction Python les unes après les autres afin de réaliser des calculs simples, faire de l'analyse ou simplement demander de l'aide sur une commande. Par ailleurs, vous pouvez aussi exécuter au cours de la session un programme Python préalablement écrit : ceci vous permet de garder en mémoire le contenu des variables qui y sont définies pour effectuer des tests ou des plots. Il y a plusieurs manières de démarrer une session interactive :

- Sous Unix/Linux/BSD/MacOS il suffit d'ouvrir une fenêtre de shell (nommé aussi terminal), et à l'invite de commande lancer l'instruction pyhton (ou ipython si installé, pour plus de fonctionnalités) pour ouvrir une session textuelle. L'instuction pyhton lance l'interpréteur de commande de base Python. L'instuction ipyhton lance l'interpréteur avancé IPython. *Conseil : utilisez* IPython *si celui-ci est installé sur votre ordinateur.*
  Sur une installation Windows, la distribution *Enthought* par exemple sur le bureau un alias nommé PyLab qui ouvre une console avec IPython (voir plus loin)
- Lancez l'éditeur et interpréteur graphique natif de Python IDLE. Soit à l'invite de commande dans un terminal avec l'instruction idle, soit en cliquant dans une icone qui vous serait accessible dans votre environnement de travail. IDLE est un éditeur et interpréteur de commande pour Python avec des fonctions qui rendent l'interaction avec Python plus facile. IDLE est une bonne alternative à IPython. Une lien d'introduction à IDLE en français : http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/indexfr.html.
- spyder (dans les installations PythonXY) propose aussi une console enrichie pour le travail interactif.
- Le notebook de l'installationSAGE offre une autre interface graphique enrichie pour un usage interactif.

Cette liste est loin d'être exhaustive. Les possibilités sont nombreuses et la plupart gratuites. Dans le cadre d'une initiation il est raisonnable de se familiariser avec le système le plus "rudimentaire", qui sera aussi le plus robuste dans toutes les circonstance via un terminal textuel (shell). Fonctionnant sur ce principe ipython offre les fonctionalités les plus intéressantes.

2.2 IPython

L'interpréteur Python IPython offre des fonctionnalités poussées comme la complétion des instructions, l'historique des commandes, le rappel d'instructions déjà exécutées, ou encore l'enregistrement automatique de toute une session de travail. IPython est un outil d'aide au développement et au débuggage. [2]

Dans l'interpréteur IPython chaque ligne présente un numéro de ligne courante, typiquement,In [1]: pour entrer une commande et, Out[1]: pour le résultat de la commande. Ces lignes (entrée et sortie) peuvent être par la suite rappelée par leur numéro. Quelques exemples de commandes IPython :
Les lignes de commandes déjà exécutée peuvent être réutilisées par la suite en faisant référence à leur numéro.

- Reprendre une ligne passée déjà exécutée %rep num_ligne ou
  pour toutes les lignes

2. Voir : http://eric-pommereau.developpez.com/tutoriels/introduction-ipython/?page=introduction

– Parcourir la liste des commandes précédemment exécutées : `Ctrl-p`, `Ctrl-n` ou flèches haut et bas ?
– Utilisation de la touche de tabulation `Tab` pour compléter les entrées (évite de tout taper)
– Explorer des objets créés (fonctions, variables, libstes) `:object_name?` (le nom de l'objet suivi du point d'interrogation)
– Exécution d'une commande shell avec l'opérateur '!' : `In [1]: !ls` dossier /
– Navigation dans le système de fichiers : `%cd /home/user/mon_nom`
(pour aller au répertoire `/home/user/mon_nom`)
– Historique des instruction exécutées durant la session interactive `%hist`
– Se souvenir de toutes les commandes exécutées durant une session :
la commande `In [1]: %logstart mon_fichier_session.log` en début de session va enregistrer tout ce que vous faites dans le fichier `mon_fichier_session.log`.
– Sortir de IPython `%exit` ou `%quit`

Nous vous conseillons de choisir soit `IDLE` soit `IPython` comme environnement de travail interactif.

2.3 Pour des programmes indépendants

Alternativement à l'usage interactif de Python, vous pouvez écrire un programme en Python (que l'on appelle aussi *script*), dont vous lancez l'exécution a posteriori [3]. Une session interactive devient alors très utile pour tester des instructions avant de les inclure dans votre programme.

2.3.1 Sous Windows

Les scripts Python sont exécutés par l'application `python.exe` ou `pythonw.exe` (si on ne veut pas que la console soit affichée), fichiers que l'on retrouve dans le répertoire d'installation de Python (typiquement `c:/python25`). En général, à l'installation de Python, cette association est faite d'elle-même et un double-clic sur votre script Python suffit pour l'exécuter. Vous pouvez également l'exécuter à partir de la console DOS en tapant directement `c:`python25/python.exe monscript.py/ ou bien simplement `python monscript.py` si le répertoire `c:/python25/` a été ajouté auparavant à la variable d'environnement `PATH` de windows (Panneau de Configuration->Système->Avancé->Variables d'environnement).

2.3.2 Sous linux

En début de script, il vous suffit d'insérer en première ligne `#!`usr/bin/env python/. Cette ligne permet au système Unix de trouver quel logiciel doit être utilisé pour interpréter la suite du fichier. Donnez ensuite l'attribut exécutable au fichier (soit en ligne de commande avec `chmod+x monscript.py` soit à partir d'un gestionnaire de fichier) et exécutez le fichier à partir la ligne de commande
`python monscript.py` [4].

Donc, pour lancer le script Python nommé `monscript.py` :
– à partir d'un shell (console) unix, `python monscript.py` ou directement `monscript.py` si on a suivi les indications ci-dessus.
– à partir d'une session interactive Python : `execfile("monscript.py")`.
– à partir d'une session interactive IPython : `run monscript.py`.
Certains éditeurs permettent également d'exécuter directement des scripts. Par exemple sous `IDLE`, un simple `F5` exécute votre source. Sous `emacs`, pour un fichier ouvert en mode python, on lance l'exécution du fichier par `crtl-c crtl-c` ou d'une région sélectionnée à la souris par `\ctrl-c |`.

3. N.B. Utilisez la terminaison `.py` pour les scripts Python.
4. Référence : `http://python.developpez.com/faq/?page=Generalites`

3 Quelques rappels de Python

De nombreux tutoriels sont disponibles en ligne [5], consultez le site du cours sur le Didel. Ici, dans la suite, juste un bref rappel.

3.1 L'en-tête

3.1.1 Modules

Afin de profiter pleinement du potentiel de Python, il faut importer des modules contenant des fonctions (méthodes) avancées. Il y a plus de 11 000 modules [6], mais nous n'en utiliserons ici que principalement que trois : **NumPy** pour la manipulation de vecteurs, matrices ou l'utilisation de fonctions complexes, **SciPy** pour des algorithmes scientifiques de statistiques, traitement de données, intégration etc. . .et **PyLab** version étendue du module **Matplotlib**, pour le tracé de courbes [7] [8]. On peut les importer de plusieurs façons :

```
import scipy                  # sans alias : scipy.nom_methode()
import scipy as S                  #  utilisation avec alias : S.nom_methode()
from scipy import *                  # utilisation directe : nom_methode()
```

Il faut aussi noter que ces modules peuvent présenter des sous-modules (*subpackages*) pour des fonctions plus spécialisées qui sont à importer explicitement pour être utilisés :

```
import scipy.linalg         # utilisation: scipy.linalg.nom_methode()
from scipy import linalg         # utilisation directe: linalg.nom_methode()
```

L'obligation de l'importation des package peut apparaître comme une lourdeur. Mais il s'agit d'une caractéristique dont l'intérêt apparaît surtout dans un usage avancé de Pyhton, qui confère au langage versatilité et robustesse. [9]

3.1.2 Documentation

Pour se documenter sur les modules, leurs méthodes et sous-modules, python possède une documentation complète accessible via le shell. Il suffit de lancer python et d'utiliser la commande `help(nom_a_chercher)`. Par exemple, `help(int)` documente la conversion en entier. Pour se documenter sur un module, il faut d'abord l'importer :

```
>>> import scipy
>>> help(scipy)
```

Si vous êtes dans une session `Ipython` alors on peut aussi obtenir de l'aide pour toute instruction ou fonction d'un module par le nom de l'instruction suivi d'un point d'interrogation. Ceci retourne une description complète avec souvent des exemples d'utilisation (et beaucoup d'informations que nous ne détaillerons pas ici). Par exemple pour se rappeler l'usage de l'instruction `range()`, écrire à l'invite `range?` (le nom de l'instruction sans paranthèse suivi de `?`) :

5. Dans un esprit scientifique, par exemple `http://www.dakarlug.org/pat/scientifique/html/python.html`
6. Voir par exemple `http://pypi.python.org/pypi`.
7. Le "Python Scientifique" est en fait la combinaison des 3 extensions du langage : **Numpy/Scipy/Matplotlib**.
8. Quelques autres tutoriels Python scientifique en français `http://www-fourier.ujf-grenoble.fr/~faure/enseignement/python_M1/ModulesScientifiques.pdf` `http://www-fourier.ujf-grenoble.fr/~faure/enseignement/python_M1/Didacticiel_fred.pdf`
9. En mode de travail interactif on pourra se simplifier l'écriture des instruction en important toutes les instructions d'un module pour les utiliser sans qu'il soit nécessaire de le préfixer par `from <nom_module> import *`. A éviter dans un programme.

```
In [1]:  range?

Type:                builtin_function_or_method
Base Class:         <type 'builtin_function_or_method'>
String Form:         <built-in function range>
Namespace:         Python builtin
Docstring:
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

Notez que dans la pratique moderne connectée à internet, l'usage est de s'appuyer fortement sur les moteurs de recherche, documents et exemples accessibles par internet.

3.1.3 Encodage

Pour que les caractères français soient reconnus, on peut mettre cette ligne en début de document :

```
# -*- coding: utf-8 -*-
```

Ceci permet l'utilisation d'accents dans de chaines de caractères. Mais il ne faut pas en utiliser pour des noms de variables !

3.2 Structure des boucles et conditions logiques

3.2.1 Les opérateurs à connaître

Les outils de comparaison de Python sont les suivants :

- $<$ strictement inférieur
- $>$ strictement supérieur
- $<=$ inférieur ou égal
- $>=$ supérieur ou égal
- $==$ égal
- **!**$=$ différent

La valeur renvoyée est un Booléen (True ou False). Il est aussi possible d'enchaîner les comparaisons : `x<y<z` réalise deux tests sur y, `x<y` et `y<z` .

Deux variables booléennes peuvent être comparées :

- X **is** Y renvoie True si X et Y représentent le même objet.
- X **is not** Y renvoie True si X et Y représentent deux objets différents.
- **not** X renvoie le booléen opposé à X.
- X **and** Y renvoie True si X et Y sont vrais.

- X **or** Y renvoie True si X ou Y est vrai.

De même, on peut enchaîner les comparaisons : `X is Y and not Z`.
Enfin, tout objet non nul (une valeur réelle différente de 0 ou une chaîne de caractère non vide) est évalué à True.

3.2.2 Boucles et conditions logiques

En python, la règle la plus importante à respecter est l'indentation. Elle suit l'indicateur deux points (« : ») et définit les blocs de codes : l'indentation (touche tabulation) démarre un bloc et la désindentation le termine. Elle permet aussi une lecture plus aisée du programme. Les structures suivantes sont autant d'exemples de cette règle.

La condition if

```
if <condition>:
        <commandes>
elif <condition>:
        <commandes>
else:
        <commandes>
```

Par exemple, voici une condition if simple :

```
x = 10
if x > 0:
        print 1
elif x == 0:
        print 0
else:
        print -1
## -> 1
```

La boucle for

```
for <loop_var> in <sequence>:
        <commandes>
```

La séquence sur laquelle on itère peut être un vecteur de nombres, ou n'importe quelle liste d'objets :

```
# Boucle sur des entiers
for i in range(5):
        print i
## -> 0 1 2 3 4

# Boucle sur une chaîne de caractères
for i in 'abcde':
        print i
## -> a b c d e

# Boucle sur une liste
```

```
list=["chiens","chats"]
animaux=""
for item in list:
        animaux = animaux + item + " "
        print animaux
## -> chiens
## -> chiens chats
```

On note ici la concaténation de chaînes de caractères via l'opérateur « + ».

Par ailleurs, dans le cas des listes, une fonction pratique est `enumerate`, qui permet d'itérer à la fois sur le contenu de la liste et sur l'indice d'itération :

```
for index,item in enumerate(list):
        print index,item
## -> 0 chiens
## -> 1 chats
```

On remarquera ici que la forme `x,y` est équivalente à `(x,y)` : ces objets sont des *tuples* et diffèrent des *listes* (type `[x,y]`) par le fait qu'on ne peut pas modifier leur taille. Cette notation permet des raccourcis pratiques pour l'assignement, du type `x,y,z = 1,2,3` au lieu de `x=1`, `y=2`, `z=3`.

On peut aussi faire une boucle sur plusieurs objets à la fois avec la fonction `zip` :

```
for nombre,animal in zip([3,6],["chiens","chats"]):
        print "Il y a", nombre, animal
## -> Il y a 3 chiens
## -> Il y a 6 chats
```

La boucle while

```
while <condition>:
        <commandes>
```

Par exemple :

```
x=2
while x>0:
        x=x-1
        print x
## -> 2 1
```

3.3 Définition d'une fonction

Les fonctions sont introduites pas `def` et suivent le schéma de l'exemple suivant :

```
import scipy as S

def polar2cartesian(r,phi):
        x = r * S.cos(phi)
        y = r * S.sin(phi)
        return x,y
```

La fonction est alors appelée de la manière suivante :

```
r,phi = 1,S.pi/2
x,y = polar2cartesian(r,phi)
print x,y
## -> 0,1
```

3.4 Lire des valeurs au clavier

La fonction `raw_input()` lit des caractères au clavier et les retourne dans une variable de type chaine de caractères. Pour récupérer une valeur u numérique il faut convertir une chaine en variable numérique.

```
x = float(raw_input('Entrer  valeur x: '))   #avec conversion en variable flottante
```

3.5 Lire un fichier de données

De très nombreuses modules existent pour les entrées et sorties avec des fichiers en fonction des contextes. On peut bien sûr utiliser les outils standards de Python, tels que `readlines()`, `readline()`. Quand les données sont ordonnées en colonnes, il est encore plus simple d'utiliser la méthode très pratique `loadtxt()` de Pylab.
Considérons, par exemples, un jeu de données placées dans un fichier nommé `donnees.dat`, avec le format suivant, où la première ligne contient les noms des colonnes :

```
t  x  y
0.1,   0.45,  2.3
0.3,   1.20,  1.8
0.4,   1.3,   1.4
0.7,   2.1,   0.5
1.1,   4.3,  -0.3
1.3,   2.7,   0.1
```

Pour lire le fichier il faut d'abord localiser le chemin vers le fichier /v/donnees.dat/. Par exemple il pourrait se trouver dans le répertoire `C:\documents\` sous Windows, ou sous Unix,MacOsX `/Users/mon_compte/documents/` (les OS ont des écritures différentes pour les chemins). Repérez le nom complet, avec chemin, de votre fichier (donc `/Users/mon_compte/documents/donnees.dat` sous unix).

Lisez le fichier avec :

```
import pylab as P
donnees=P.loadtxt('<chemin jusqu'à donnees.dat>', skiprows=1, delimiter=',')
```

où v/<chemin jusqu'à donnees.dat>/ est le nom complet avec chemin du fichier, `skiprows=1` indique de sauter la première ligne, et `delimiter=','` indique que le séparateur de colonnes est la virgule (vous pouvez indiquer tout autre séparateur, pour les tabulation le code à indiquer est `\t`).

L'instruction de pylab `loadtxt()` va lire et placer toutes les données dans le tableau `donnees`, qui contiendra ici 3 colonnes de 6 lignes. On récupère des arrays à une dimension pour tracer par de la manière suivante [10] :

10. On peut tout condenser dans une instruction unique :
`t,x,y=P.loadtxt('<chemin jusqu'à donnees.dat>', skiprows=1, delimiter=',', unpack=True)`

```
t=donnees[:,0]
x=donnees[:,1]
y=donnees[:,2]
P.plot(t,x,t,y)
```

Lire des fichiers n'est pas toujours facile. En effet les fichiers informatiques peuvent prendre un grand nombre de formats différents et les instructions de lecture doivent être compatibles. Dans certains cas les fichiers de données sont en format binaire, qui occupe moins de place mémoire et est plus rapide à traiter. Mais le fichier n'est accessible que si on connait exactement les spécifications du format. On peut avoir des problèmes aussi avec des fichiers en fomat *text* a priori standard, surtout si l'on passe d'un ordinateur à un autre, un système à un autre ou utilise des données passant par internet. En effet les conventions peuvent-être très différentes pour l'*encodage* des caractères spéciaux commes les accents et surtout pour les codes pour les terminaisons de lignes : sur des Macs les fin de lignes sont encodés par un symbole "retour à la ligne" (carriage-return en anglais) (`\r`), en Linux le symbole de fin de ligne est "avance de ligne" (linefeed) `\n`, en Windows c'est les deux `\r\n` ! Donc suivant l'origine des fichier il faut s'attendre à devoir jongler et chercher des astuces sur internet...

Avec le module `csv` de lecture d'un fichier en format colonnes on peut utiliser l'option *U*

```
reader = csv.reader(open('mon_fichier.csv', "U")) # or "rU"
```

Cette option ouvre le fichier en mode "universel" pour les terminaisons de ligne, et s'adaptera au format de la fin de ligne du fichier.

3.6 Quelques outils utiles de Python Scientifique...

Alors que la librairie Numpy, enrichit Python notamment de fonctionalités pour le calcul efficace de tableaux et matrices de nombres (*arrays*), SciPy est une librairie scientifique générale implémentant de nombreux algorithmes de calcul. Numpy permet la manipulation de tableaux de nombres comme des matrices (tableaux de scalaires), par exemple pour de l'algèbre linéaire (par exemple valeurs propres, vecteurs propres). Scipy offre une impressionnante collection de procédures pour tout type de calcul scientifique (statistiques, optimisation, interpolation, intégration, etc...)

Attention, la convention Python pour les indices est $i = 0$ pour le premier élément.

3.6.1 Manipulation de tableaux, vecteurs, matrices...

```
import scipy as S

# Création d'un tableau vide (resp de zéros, de uns)
emptyarray=S.empty((nrow,ncol)) # matrice vide
zerosarray=S.zeros(nlength)        # vecteur nul
onesarray=S.ones((nx,ny,nz)) # tableau 3D rempli de uns

# Très utile!!
S.arange(xmin,xmax,step) # vecteur de valeurs espacées de step entre xmin et xmax
S.linspace(xmin,xmax,count) # vecteur de count valeurs réparties entre xmin et xmax

# Création d'un vecteur (matrice 1D) de valeurs réelles
a = np.array([1, 4, 5, 8], float)
```

```
print a
## -> array([ 1., 4., 5., 8.])
type(a)
## -> type 'numpy.ndarray'

# Création d'une matrice 2*3
A=S.array([[1,2,3],[4,5,6]])
print A
## -> array([[ 1, 2, 3 ] ,
##                         [ 4, 5, 6 ]])

# Assignation d'une nouvelle valeur
A[0,1]=7
print A
## -> array([[ 1, 7, 3 ] ,
##                         [ 4, 5, 6 ]])
```

Extraction de parties d'une matrice On appelle *slicing* des méthodes pour découper et extraires des parties d'un tableau (ou matrice). Très utile, par exemple pour récupérer les n-premiers éléments d'une liste de valeurs, ou la m-ième colonne d'un tableau de valeurs.

```
a = S.arange(10)
print a
## ->  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print  a[2:9:3] # affiche les valeurs du tableau ‘‘a’’ à partir de la
                      # case ‘‘debut’’ jusqu'à la case  ‘‘fin’’  de
                      #‘‘pas’’ en ‘‘pas’’  [début:fin:pas]
## -> array([2, 5, 8])
```

Dans l'instruction [début :fin :pas] deux des arguments peuvent être omis : par défaut l'indice de début vaut 0 (le 1er élément), l'indice de fin est celui du dernier élément et le pas vaut 1.

```
print a[2:9]
## -> array([2, 3, 4, 5, 6, 7, 8])
print  a[2:] array([2, 3, 4, 5, 6, 7, 8, 9]
```

On peut étendre ces sélections à chaque dimension d'un tableau. Par exemple pour sélectioner un sous-tableau :

```
A=S.array([[1,2,3],[4,5,6]])
B=A[:,1:] #Toutes les lignes (premier argument ‘‘:’’) et les colonnes de
               #1 à la fin (second argumen 1:)
print B
## -> array([[ 7, 3 ] ,
##                         [ 5, 6 ]])
```

3.6.2 Exemples de calcul matriciel

```
# Calcul des valeurs propres
# Utilisation du sous-module linalg de scipy
```

```
from scipy import linalg
eigens=S.linalg.eigvals(B)
print eigens
## -> array([ 10.40512484+0.j,   2.59487516+0.j])

# Solutions de an*x**n+...+a1*x+a0=0
S.roots([an,...,a1,a0])
```

3.6.3 Intégration d'équations différentielles

Le sous-module `integrate` de SciPy contient diverses méthodes d'intégration. Notamment, la fonction `odeint(dX_dt,X0,t)` permet d'intégrer le système d'équation différentiel `dX_dt` étant donnés un vecteur `X0` de conditions initiales et un vecteur `t` contenant les points d'intégrations. Voici par exemple comment intégrer l'équation $\ddot{z} = -g$:

```
import scipy as S
from scipy.integrate import odeint

def dX_dt(X,t,(acceleration,frottement)):
        v = X[1]
        dz = v
        dv = -acceleration+frottement*v
        return S.array([dz,dv])

parametres=(9.8,0.1)
t=S.arange(0,5,0.1)
X0=S.array([100,10])
Y=odeint(dX_dt,X0,t,args=(parametres,))
```

On remarque d'abord que la variable `t` (par défaut 0) est implicite mais doit être précisée pour que la fonction `dX_dt` soit reconnue par `odeint`. Par ailleurs, les équations différentielles doivent être du premier ordre : on a utilisé une variable intermédiaire (la vitesse) pour convertir l'équation $\ddot{z} = -g$ en un système de deux équations du premier ordre : $\dot{v} = -g$ et $\dot{z} = v$. Les conditions initiales ont été ici choisies comme étant $z(t = 0) = 100$ m, et $v(t = 0) = 10$ m.s^{-1}. On passe en argument aussi un tuple `args=(parametres,)` contenant les paramètres de l'équation différentielle (Attention à la syntaxe, il faut une virgule après le nom du tuple). Le résultat `Y` est un tableau à deux colonnes : les positions et vitesses aux différents temps du vecteur `t`.

3.7 Tracer un graphe avec PyLab (Matplotlib)

Voici maintenant la procédure pour tracer un graphe avec PyLab [11] Il est important que les vecteurs de points (abscisses, ordonnées) aient exactement la même taille. Nous reprenons ici notre exemple précédent.

```
import pylab as P
# On trace la courbe z(t) rouge ('r-') et on précise sa légende
P.plot(t,Y[:,0],'r-',label='Altitude')
# on peut dans certains cas préférer un tracé en échelles log log
```

11. PyLab est une interface interactive du module général python pour des graphiques Matplotlib. Ce dernier offre des fonctions graphiques très avancées qui peuvent être utilisées pour la programmation d'applications professionnelles. PyLab offre une interface simplifiée à Matplotlib, commode notamment pour un usage interactif à la console de commandes.

```
P.loglog(t,Y[:,0])

# On peut renommer les axes et mettre un titre
P.xlabel(‘temps’)
P.ylabel(‘Altitude’)
P.title(‘Chute d’un objet’)

P.axis([0,5,0,150]) # Pour fixer leur longueur

P.grid()  # Pour afficher une grille pour plus de clarté

P.legend(loc=‘best’) # Pour afficher la légende où il y a de la place

P.savefig(‘altitude.pdf’) # Pour sauver le graphe dans un fichier pdf

P.show() # Pour afficher le graphe dans une fenêtre
```

Pour connaître toutes les options, le mieux est de se référer à la documentation de Matplotlib (http ://-matplotlib.sourceforge.net/contents.html). Dans la pratique, visualisez un des nombreux exemples dans la gallerie (http ://matplotlib.sourceforge.net/gallery.html) pour une figure du type que vous souhaitez faire et copiez/adaptez le code !

Quelques instructions utiles :
– `loglog(x,y,option)` Trace graphique log-log
– `semilogx(x,y,option)` Trace graphique log-normal
– `semilogy(x,y,option)` Trace graphique normal-log
– `show()` Génère le graphique à l’écran
– `savefig('fichier.png')` Sauvegarder dans un fichier

3.8 Tracer une figure avec plusieurs graphes

L’instruction de Pylab `subplot(l,c,num)` trace sur une même figure $l \times c$ graphes, rangés en l lignes et c colonnes. Les graphes sont numérotés de 1 à $l \times c$ de gauche à droite et de haut en bas.

```
import pylab as P

x = arange(0, 10, 0.5)           #Definir intervalle  x: 0 a 10 avec pas de 0.5
y = x**2                         # Fonction x *x
z = 5*x                          # Fpnction 5*x

P.figure(1) #ouvrir une fenêtre de figure 1

P.subplot(1,2,1)     # dans la figure 1 on aura 1 ligne de 2 colonnes de graphes,
                     # on trace le premier
P.plot(x,y,’bo-’,label=’$x^2$’,markersize=8.0,linewidth=2.0)
#utilisant des cercles bleus et une ligne

P.subplot(1,2,2)   #on trace le second graphe
P.plot(x,z,’rx’,label=’$5x$’,markersize=10.0,linewidth=2.0)
#utilisant des ’x’ rouges sans ligne
```

```
P.legend()   # ajout des légendes
P.xlabel('axe x')
P.ylabel('axe y')
P.title('Graphe simple')
P.grid(True)
P.show()
```

3.9 Tracer une image 2D

Pour tracer une image 2D, instruction de PyLab `imshow(Z)` (chaque case de la matrice Z donne un pixel d'une couleur suivant sa valeur).

Pour tracer en projection 2D une fonction Z(X,Y), instruction de PyLab : `pcolormesh(X,Y,Z)`

3.10 Tracer un histogramme

```
import pylab as P
x = randn(1000)
y = randn(1000)+5

#distribution normale centee a x=0 et y=5
P.hist2d(x,y,bins=40) #40 intervalles
P.show()
```

3.11 Faire une transformée de Fourier

`fft()` pour la transformation directe, `ifft()` pour la transformation inverse. La fonction `fftfreq()` retourne un vecteur de fréquences ordonné identiquement à la transformée de Fourier. Cette fonction doit être importée du sous-module fft de Numpy.

```
import scipy as S
import pylab as P
from numpy.fft import fftfreq

#Importation des donnees
fname='Mes_donnees.dat' #Le nom de votre fichier de donnees
t,x=loadtxt(fname,usecols=(0,1),skiprows=0,unpack=True)
#Voir les options de loadtxt pour personnaliser l'import

# pas de temps entre  chaque point il y a ``echantillonnage'' secondes
dt=echantillonnage

#pour la TF il faut un nombre de points puissance de 2
#on ne retient qu'un  nombre de points dans le fichiers correspondant à
#la plus puissance de 2 la plus grande possible
# (PS on pourrait tout retenir, mais TF plus lente)

exposant_FFT=int(S.log(len(x)/log(2))
FFT=2**exposant_FFT   #Nombre de points dans la TF

w=fftfreq(FFT) #generation d'une array des frequences
```

```
ffx=S.fft(x-S.mean(x),FFT) #Transformee de Fourier centree sur moyenne
w=S.array(w)/dt    #mise a l'echelle de l'axe x en Hertz

#Calcul du spectre de puissance
spectre_puiss=(abs(ffx[0:len(ffx)/2-1]))**2/(max(abs(ffx[0:len(ffx)/2-1])))**2

P.plot(w[0:len(w)/2-1],spectre_puiss)  #Plot
```

3.12 Résoudre un système d'équations non linéaires

On utilise la fonction de Scipy `fsolve()` [12]

```
import pylab as P
import scipy as S
from scipy.optimize import fsolve
"""
Exemple utilisation de fsolve() pour trouver les intersections de
exp(x)-1 avec cos(x). Remarque: on doit donner une estimation initiale
"""
def f(x):
    '''La fonction de l'équation'''
    return (S.exp(x)-1)-S.cos(x)
result = fsolve(f,-1.5)  # la résolution ave estimation initiale
print result
x = S.linspace(-5,1,101)
P.plot(x,exp(x)-1,x,cos(x))
P.show()
```

Notez, on donne ici des exemples simples. La fonction `fsolve` comme celles dans les autres exemples, possèdent de nombreuses options, à voir dans la documentation en ligne des module ou bien avec l'aide intégrée à python (ci-dessus faire dans une session python `help(fsolve)`, après avoir importé la fonction bien-sûr).

3.13 Integration numérique

Utilisation de Scipy pour intégration numérique [12] :

```
import scipy as S
from scipy.integrate import quad
"""
Exemple d'intégration utilisant la fonction quad() de scipy.integrate
appliqué à la fonction exponentielle pour illustration
"""
def f(x):
    '''la fonction à intégrer'''
    return  S.exp(-x)

result = quad(f,0,5)   #Intégration entre 0 et 5

P.disp(['Numerical result: ',result]) # Affiche le résultat numérique
```

12. Exemple adapté de `http://www.uncg.edu/phy/hellen/Python_Instructions.html`

```
analytique = 1-S.exp(-5)
P.disp(['Analytic result: ',analytic]) # Affiche le résultat analytique
```

3.14 Fit de graphes

Utilisation de Scipy pour fit de données [12] avec `leastsq()` (algorithme de Levenberg-Marquardt) :

```
import scipy as S
import pylab as P
from scipy.optimize import leastsq
"""
Exemple fit de données avec la fonction
a1*exp(-k1*t) + a2*exp(-k2*t)
"""
def fonction(t,p):
    return(p[0]*S.exp(-p[1]*t) + p[2]*S.exp(-p[3]*t))

def residuals(p,data,t):
    '''Fonctions residus on veut minimiser l'erreur err
      entre les donnees data et la fonction '''
    err = data - fonction(t,p)
    return err

#  DONNEES ici génération de données avec bruit pour illustration
# on peut ici importer des données d'un fichier
a1,a2 = 1.0, 1.0 # paramètres
k1,k2 = 0.05, 0.2
t=S.arange(0,100,0.1)  #intervalle t des valeurs

data = fonction(t,[a1,k1,a2,k2]) + 0.02*S.randn(len(t)) #on ajoute du bruit

# FIT
p0 = [0.5,1,0.5,1] # estimation initiale des parametres

p_meilleurs = leastsq(residuals,p0,args=(data,t),full_output=1) #FIT

bestparams = pbest[0] #parametres de fit dans premiere colonne
cov_x = pbest[1] #covariance du fit dans seconde colonne

print 'meilleur fit ',bestparams
print cov_x
datafit = fonction(t,bestparams)
P.plot(t,data,'x',t,datafit,'r')
P.xlabel('Temps')
P.title('Exemple de fit')
P.grid(True)
P.show()
```