Modules de calcul scientifique avec Python (introduction)

20 février 2009

1 Utilisation de "modules" (standards et librairies extérieures)

Pour importer un module, il faut utiliser la commande "`import` *`NomDuModule`*". Il est alors possible d'obtenir une aide sur le module avec la commande "`help(`*`NomDuModule`*`)`". Les fonctions s'utilisent sous la forme "*`NomDuModule.NomDeLaFonct`*

Il est également possible d'importer le contenu du module sous la forme "`from NomDuModule import *`" et alors les fonctions peuvent être utilisées directement par "`NomDeLaFonction(paramètres)`".

1.1 math

Ce module standard comporte toutes les fonctions mathématiques usuelles :

```
>>> import math
>>> print math.sin(math.pi/4)
0.707106781187
>>> print math.log(10.0)
2.30258509299
```

ou :

```
>>> from math import *
>>> print sin(pi/4)
0.707106781187
>>> print log(10.0)
2.30258509299
```

Pour utiliser des fonctions mathématiques sur des **complexes**, il faudra importer le module `cmath` en plus du module `math`. Néanmoins en général il est plus simple d'utiliser directement **scipy**, qui inclut toutes les fonctions de math et cmath, et permet également de les utiliser sur des tableaux.

1.2 Scipy (scientific python, http ://www.scipy.org)

Ce module ne fait pas partie par défaut de Python mais est en général présent, ou peut être obtenu à partir de l'adresse http ://www.scipy.org.

Ce module permet en particulier d'effectuer des calculs sur des tableaux :

```
>>> from scipy import *
>>> a=zeros([2,5],floating)     # un tableau de 2 lignes et 5 colonnes
>>> print a      # rempli de 0.0 (nombres réels)
[[ 0. 0. 0. 0. 0.]
[ 0. 0. 0. 0. 0.]]
>>> a[1,3]=4.0            # Modification d'un élément
>>> print a
[[ 0. 0. 0. 0. 0.]
[ 0. 0. 0. 4. 0.]]
>>> a[0, :]=1.0            # Modification de la première ligne
>>> print a
[[ 1. 1. 1. 1. 1.]
[ 0. 0. 0. 4. 0.]]
>>> print a[ :,3]          # Affichage de la quatrième colonne
[ 1. 4.]
>>> sin(a)
array([[ 0.84147098, 0.84147098, 0.84147098, 0.84147098, 0.84147098],
[ 0. , 0. , 0. , -0.7568025 , 0. ]])
```

Attention sur la dernière ligne on utilise une fonction sin(), qui provient du module `scipy`. Si les deux modules (`math` et `scipy`) étaient importés avec "`from NomDuModule import *`", il y aurait une ambiguité sur quelle fonction `sin`() à utiliser... En fait la dernière fonction `sin()` importée écraserait la précédente...

Dans ces cas là, n'utiliser que "`import NomDuModule`" et ensuite `math.sin()` et `scipy.sin()` suivant la fonction désirée. On peut même n'importer qu'une fonction d'un module, par exemple avec "`from math import sin`".
Lire l'aide de scipy pour voir les différentes fonctions disponibles. En particulier, on peut créer des tableaux avec les fonctions `ones`, `zeros` , et en utilisant comme type : complex64, complex128, float32, float64, int16, int32, int64, suivant le type de donnée et la précision souhaitée.

1.2.1 Exemples

Commencer par le **didacticiel de scipy**, consacré aux manipulations algébriques de matrics, qui se trouve à :
`http ://www.scipy.org/SciPy_Tutorial`
Essayer "`Example 1`", "`Example 2`" et "`Numerical integration`".

Regarder des petits exercices dans le cookbook de scipy : `http ://www.scipy.org/Cookbook`
En particulier "`linear regression`", "`optimization`", "`smoothing a signal`"
Des **petits exemples** d'utilisation de fonctions scipy peuvent être trouvé à :
`http ://www.scipy.org/Numpy_Example_List`

1.2.2 Sous-modules de scipy

Scipy est un ensemble qui comprend de nombreux modules utiles pour des scientifiques :
* cluster : information theory functions (currently, vq and kmeans)
* weave : compilation of numeric expressions to C++ for fast execution
* fftpack : fast Fourier transform module based on fftpack and fftw when available
* ga : genetic algorithms
* io : reading and writing numeric arrays, MATLAB .mat, and Matrix Market .mtx files
* integrate : numeric integration for bounded and unbounded ranges. ODE solvers.
* interpolate : interpolation of values from a sample data set.
* optimize : constrained and unconstrained optimization methods and root-finding algorithms
* signal : signal processing (1-D and 2-D filtering, filter design, LTI systems, etc.)
* special : special function types (bessel, gamma, airy, etc.)
* stats : statistical functions (stdev, var, mean, etc.)
* linalg : linear algebra and BLAS routines based on the ATLAS implementation of LAPACK
* sparse : Some sparse matrix support. LU factorization and solving Sparse linear systems.

Là encore, de la documentation est disponible (après import du module) avec la commande "`help(NomDuModule)`"
Par exemple le module FFTPACK :

```
>>> import scipy
>>> from scipy import fftpack
>>> a=scipy.array(range(20),scipy.floating)
>>> print a
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14.
15. 16. 17. 18. 19.]
>>> print scipy.fftpack.fft(a)
[ 190. +0.j -10.+63.13751515j -10.+30.77683537j
-10.+19.62610506j -10.+13.7638192j -10.+10.j
-10. +7.26542528j -10. +5.09525449j -10. +3.24919696j
-10. +1.5838444j -10. +0.j -10. -1.5838444j
-10. -3.24919696j -10. -5.09525449j -10. -7.26542528j
-10.-10.j -10.-13.7638192j -10.-19.62610506j
-10.-30.77683537j -10.-63.13751515j]
```

1.2.3 Extensions ou applications de scipy

Se référer à : http ://www.scipy.org/Topical_Software

1.3 Tracé de données scientifiques (1D, 2D) avec matplotlib/pylab

Le tracé de toutes les courbes "scientifiques" se fait à l'aide de **matplotlib**, qui est une librairie dont les commandes sont similaires à matlab. Pour l'utiliser, il faut importer le module "pylab" (rcommandé) ou matplotlib. Si on utilise ipython, le plus simple est de démarrer ipython avec l'option -pylab, i.e. : "`ipython -pylab`"[1]. Cela importe tout le module et permet de travailler en mode "interactif", ç'est-à-dire que la figure s'affiche au fur et à mesure qu'on tape les instructions de tracé (plutôt que de ne voir la figure qu'à la fin).

La référence complète de matplotlib est lisible à l'adresse : **http ://matplotlib.sourceforge.net/matplotlib.pylab.html** . Regarder la section Il est en particulier recommandé de regarder les "**screenshots**", qui sont donnés avec le code utilisé pour les générer.

1.3.1 Courbe 1D

Pour tracer une sinusoïde :

```
x=linspace(-5,5,101) # coordonnées de -5 à 5 avec 101 valeurs
y=sin(x)
plot(x,y) # Tracé de la courbe !
```

On obtient une courbe sur laquelle on peut zoomer, modifier les marges, et sauvegarder dans différents formats (jpg, png, eps...)

On peut rajouter un titre, une légende, du texte avec les fonctions `title()`, `legend()` et `text()`. Voir les documentations de ces fonctions soit sur le site web de matplotlib, soit en utilisant " `?`" après le nom de la fonction dans `ipython`.

Pour tracer plusieurs courbes, on peut les mettre les unes à la suite des autres, par exemple :

```
plot(x,y,"r-",x,cos(x),"g.")
```

"r-" indique que la première courbe est à tracer en rouge avec des traits, et "g." que la deuxième est à tracer en vers avec des points. Voir l'aide de `plot()` pour connaître les autres options de ce tracé.

1.3.2 Courbe 2D

Pour cela on utilise `imshow(z)` ou `pcolor(x,y,z)`.

```
x=linspace(-5,5,201)
y=linspace(-7,7,201)[ :,newaxis] # newaxis indique que ce vecteur est selon la 2ème dimension
z=sin(x**2+y**2)
imshow(z) # Affiche l'image en 2D
imshow(z,extent=(x.min(),x.max(),y.min(),y.max())) # On précise les coordonnées des axes
jet() # Pour avoir une plus jolie table de couleurs
```

L'inconvénient de imshow est que cette fonction suppose que le tableau 2D affiché (z) correspond à des coordonnées régulières en x et y, ce qui n'est pas forcément le cas. Pour cela on utilise pcolor, mais x,y,z doivent alors tous être des tableaux 2D ! Par exemple :

```
x=linspace(-5,5,201)
y=linspace(-7,7,201)[ :,newaxis]
x=x+y*0.4 # transforme x en tableau 2D
y=y+x*0
z=sin(x**2+y**2)
#imshow(z,extent=(x.min(),x.max(),y.min(),y.max())) # Représentation non correcte, inclinée
clf() # Efface la figure avant un nouveau tracé (si on affiche plusieurs figures à la suite)
pcolor(x,y,z,shading='flat') # effectue le tracé
```

1.4 Autres modules scientifiques

PyMol et ***cctbx*** : pour la cristallographie (en particulier des protéines)
PyQuante : Chimie Quantique
Snack : pour le traitement du signal
pyVTK et Mayavi, OpenDX : pour la représentation graphique en 3D

etc... Une majeure partie des librairies de calcul développées en fortran ou en C/C++ sont ainsi disponibles en Python (voir plus bas comment créer une interface avec python grâce à ***Boost.Python*** ou ***SWIG***)

Une liste (très incomplète) de modules disponibles pour Python est disponible sur le site web consacré au language (*http ://www.python.org/pypi* ; *http ://www.python.org/moin/NumericAndScientific*). De nombreux modules peuvent

[1] Pour les utilisateurs de windows, il faut soit modifier le raccourci permettant de lancer ipython (en ajoutant "-pylab" à sa suite), soit copier ce raccourci dans une console, toujours pour ajouter l'option " -pylab" à la fin.

également se trouver à partir du site Sourceforge (en cherchant dans les projets utilisant python comme langage : *http ://sourceforge.net/softwaremap/trove_list.php ?form_cat=178*).

Mais le meilleur moyen de savoir si un outil particulier a été développé en Python est encore d'utiliser *Google* !

De nombreux laboratoires utilisent et développent des outils à base de Python pour le calcul scientifique, à Grenoble (*ILL*, *ESRF*, *CEA*, *CNRS*) et ailleurs (en particulier le Laboratoire Lawrence Livermore *LLNL*, le *CERN*).

2 Performance et interface C/C++ <-> Python *[Facultatif]*

2.1 Performance : C/C++ vs Python

Etant un langage interprété, Python présente des perfomances en terme de calcul assez médiocres *si on ne prend pas garde à la méthode utilisée*. Par exemple pour une simple opération d'addition de 2 tableaux de 100 000 éléments, on peut calculer le nombre d'opérations par seconde :

```
# -*- coding : iso-8859-15 -*-
import scipy
import time      # Module de fonction sur le temps (date, heure)
taille=100000

# 1ère version avec une boucle en Python
nbiter=10
a=range(taille) # Avec une boucle
b=range(taille)
c=range(taille)
t1=time.time()    # time.time() renvoie le nombre de secondes écoulées
for i in xrange(nbiter) :
   for i in xrange(taille) :
      c[i]=a[i]+b[i]

t2=time.time()
print "Boucle : %6.3f Mflops"%( (taille*nbiter)/(t2-t1)/1e6)

# 2ème version vectorisée à l'aide d'un tableau
nbiter=1000
a=scipy.ones(taille) # Avec des tableaux
b=scipy.ones(taille)
c=scipy.ones(taille)
t1=time.time()
for i in xrange(nbiter) :
   c=a+b
t2=time.time()
print "Tableau : %6.3f Mflops"%( (taille*nbiter)/(t2-t1)/1e6)
```

La différence de temps d'éxécution est très nette, plus qu'un facteur 50 ! Si on compare avec un programme en c++ (compiler en utilisant "g++ vite.cpp -o vite" et éxécuter le programme vite avec "./vite") :

```
# 3ème version en c++
#include <stdlib.h>
#include <time.h>
#include <iostream>
using namespace std ;
int main()
{
   const unsigned long taille=100000 ;
   const unsigned long nbiter=1000 ;
   long *a=new long[taille] ;
   long *b=new long[taille] ;
   long *c=new long[taille] ;
   clock_t time0=clock() ;
   for(unsigned long i=0 ;i<nbiter ;i++)
   {
      long *pa=a ;
      long *pb=b ;
```

```
        long *pc=c ;
        for(unsigned long j=0 ;j<taille ;j++)
           *pc++ = *pa++ + *pb++ ;
     }
     clock_t time1=clock() ;
     const float seconds=(float)(time1-time0)/(float)CLOCKS_PER_SEC ;
     cout <<"C++ : "<< (float)taille * (float)nbiter/ seconds /1e6<<endl ;
     delete[]a ;
     delete[]b ;
     delete[]c ;
  }
```

La différence reste importante avec Python (facteur 2 par rapport au calcul vectorisé), mais sans être trop gênante pour des calculs pas trop longs (<1h). Ce d'autant que la version en c++ est deux fois plus longue, et utilise des pointeurs qui sont une source fréquente d'erreurs de programmation.

2.1.1 Utilisation de scipy.weave

Pour une petite partie de calcul qui doit être exécutée très rapidement, il est possible *d'insérer du code C++ à l'intérieur d'un programme Python* - cette partie du code est alors compilée automatiquement, est peut accélérer considérablement certaines opérations. Pour cela il faut importer `scipy` et le sous-module `scipy.weave` - pour insérer du code de type C++ il faut alors utiliser la fonction `scipy.weave.inline`. L'exemple ci-dessus devient alors :

```
# -*- coding : iso-8859-15 -*-
import scipy
from scipy import weave
import scipy
nbiter=1000
a=scipy.ones(taille)
b=scipy.ones(taille)
c=scipy.ones(taille)
def testweave() :
   code="""
        for(unsigned int ct=0 ;ct<nbiter ;++ct)
        {
           const double *pa=a.data() ;
           const double *pb=b.data() ;
           double *pc=c.data() ;
           for(unsigned int i=0 ;i<taille ;++i) *pc++ = *pa++ + *pb++ ;
        }
        """
   err = weave.inline(code,['a', 'b', 'c', 'taille', 'nbiter'],
                      type_converters=weave.converters.blitz,
                      compiler = 'gcc')

testweave() # Le premier appel est long, car le code est alors compilé !
nbiter=1000
t1=time.time()
testweave() # On recommence, en utilisant les code pré-compilé
t2=time.time()
print "SciPy.weave : %6.3f Mflops"%( (taille*nbiter)/(t2-t1)/1e6)
```

2.2 Interface C/C++ <-> Python avec SWIG

Lorsque des calculs longs (soit intrinsèquement, soit parce qu'il sont répétés) sont nécessaires, il vaut mieux sous-traiter les calculs à un langage plus performant (en pratique C++ ou fortran), tout en organisant le programme (entrées/sorties, représentation des données, interface utilisateur etc...) en Python.

Lorsqu'aucune librairie proposant ce calcul n'est disponible sous Python (cela devient de plus en plus rare ! Un outil essentiel du programmateur scientifique est aujourd'hui Google !), il peut être nécessaire de créer cette interface (et éventuellement le programme en C+C++/fortran). Il existe pour cela des programmes spécialisés, les plus utiles aujourd'hui étant ***Boost.Python*** (*http ://www.boost.org/libs/python/doc/index.html*) et ***SWIG*** (*http ://www.swig.org/*). Nous allons présenter (très) rapidement ce dernier.

Séparer le petit programme en C++ ci-dessus en 2 fichiers :

- le fichier d'en-tête `test1.h` qui ne comprend qu'une seule ligne de déclaration "`void FonctionTest(long taille,long nbiter) ;`"
- le ficher de code `test1.cpp` qui contient le même code que la fonction ci-dessus, en remplaçant la fonction main() par une fonction `FonctionTest()` prenant comme argument le nombre d'éléments et le nombre d'itérations à effectuer.

Ensuite créer un fichier d'interface "`test1.i`" qui définira les éléments à partager entre SWIG et Python :

```
%module test1
%{
#include "test1.h"
%}
extern void FonctionTest(long,long) ;
```

Enfin, compiler et créer l'inteface avec SWIG avec :

```
swig -python -module test1.i               # crée test1_wrap.c
g++ -c test1_wrap.cxx test1.cpp -I /usr/include/python2.3/
g++ -shared test1_wrap.o test1.o -o _test1.so       # crée _test1.so
```

Ensuite il n'y a plus qu'à tester dans l'interpréteur Python :

```
>>> import test1
>>> help(test1)
>>> test1.FonctionTest(100000,100)
C++ : 4.54545e+07
```

Bien sûr ce travail d'interface peut être complexe, donc le réserver aux calculs réellements intenses. *Rien ne sert de passer plus de temps à écrire un programme que vous n'en passerez à l'utiliser* !

3 Fenêtres et graphiques avec wxPython

wxPython (**http ://wxpython.org**) est une librairie graphique très générale, multi-plateforme. Elle est écrite en c++ (wxWidgets, anciennement appelée wxWindows - **http ://wxwidgets.org**), et permet de créer des fenêtre avec des graphiques 2D, 3D, des boutons/contrôles, la communication avec d'autres ordinateurs (*sockets*), les exécutions en parallèle (*threads*), etc... Elle est donc assez complexe, nous n'en présenterons ici qu'une *très* petite partie !

De manière générale lorsqu'on crée une application avec des calculs et un affichage graphique, il faut savoir que plusieurs "fils" (*threads*) doivent s'éxécuter en parallèle : l'un pour gérer le graphisme (retracer une fenêtre si elle a été effacée par une autre passant devant, par exemple), l'autre pour les calcul (ou pour lire les commandes tapées sur la console). Pour travailler dans ce mode multi-fil (*multi-thread*), il faut lancer ipython avec l'option wthread "`ipython -wthread`".

3.1 Animation : corde pendante

Voici un petit exemple d'animation - on crée une fenêtre, qui possède une fonction "OnPaint" dans laquelle est défini tout ce qui doit être dessiné. Un objet wxTimer est utiliser pour demander régulièrement à la fenêtre d'être redessinée.

```
import wx
from wxPython.wx import *
import time
from math import cos

TIMER_ID = wxNewId()

class Fenetre(wxPanel) :
   def __init__(self, parent) :
      wxPanel.__init__(self, parent, -1)
      wx.EVT_PAINT(self, self.OnPaint)
      self.t0=time.time()
      wx.EVT_TIMER(self, TIMER_ID, self.OnPaint)
   def OnPaint(self, event=None) :
      dc = wxPaintDC(self)
      dc.Clear()
      dc.SetPen(wx.Pen("BLACK", 4))
      angle=0.2*cos(time.time()-self.t0)
      dc.DrawLine(50, 0, 50+100*sin(angle), 100*cos(angle))

app=wxPySimpleApp()
```

```
cadre = wxFrame(None, -1, "Mon cadre!")
fenetre=Fenetre(cadre)
cadre.Show(True)

t = wxTimer(fenetre, TIMER_ID)
t.Start(200)
#app.MainLoop() #Pour laisser la main à l'application plutôt qu'à la ligne de commande
```

En commentant la dernière ligne app.MainLoop(), on peut continuer à utiliser la ligne de commande pour taper des instructions. Essayez par exemple "`fenetre.t0+=1`" pour voir le pendule faire un bon dans le temps.

Pour réaliser d'autres graphiques qu'un seimple pendule, il faut utiliser d'autres fonctions de wxPaintDC - regarder pour cela la référence de wxDC à : **http ://wxwidgets.org/manuals/2.6/wx_classref.html**. En particulier `DrawCircle`, `DrawArc`, `DrawText`,...

3.2 Animation et calcul

Si on souhaite en même temps effectuer des calculs et avoir un affichage graphique, cela se complique car un seul fil du programme peut s'éxécuter à la fois. Par exemple si on sépare le calcul et l'affichage du pendule :

Ajouter à la fin du programme précédent une boucle de calcul (ou la taper à la console) :

```
for i in xrange(10) :
   print i
   time.sleep(1) # Attend 1 seconde
```

Le pendule cesse de s'afficher ! Parce que tant que la boucle ci-dessus n'est pas terminée, elle monopolise l'ordinateur... Dès que la boucle est finie, l'affichage reprend.

Pour permettre au graphique de se tracer, il suffit de temps à autres de premettre au "fil graphique" de travailler lui aussi. Il suffit pour cela d'insérer l'instruction wxYield() au milieu de la boucle de calcul :

```
for i in xrange(10) :
   time.sleep(1) # Attend 1 seconde
   wxYield()
```

Par contre cette méthode ne marche pas toujours, car lorsque wxYield est appelé rien ne garanti que le temps laissé serve au fil d'éxécution graphique... Plutôt que d'utiliser un wxTimer qui demande périodiquement un rafraichissement de l'affichage, il vaut mieux commander cet affichage de la boucle de calcul avec wxPostEvent() suivi de wxYield() :

```
import wx
from wxPython.wx import *
import time
from math import cos,sin

class Fenetre(wxPanel) :
   def __init__(self, parent) :
      wxPanel.__init__(self, parent, -1)
      wx.EVT_PAINT(self, self.OnPaint)
      self.t0=time.time()
   def OnPaint(self, event=None) :
      dc = wxPaintDC(self)
      dc.Clear()
      dc.SetPen(wx.Pen("BLACK", 4))
      angle=0.2*cos(time.time()-self.t0)
      x,y=50+100*sin(angle),100*cos(angle)
      dc.DrawLine(50, 0, x, y)
      dc.DrawCircle(x, y,20)

app=wxPySimpleApp()
cadre = wxFrame(None, -1, "Mon cadre!")
fenetre=Fenetre(cadre)
cadre.Show(True)

#Boucle de "calcul"
for i in xrange(100) :
   print i
   time.sleep(0.05) #Mettre ici le calcul
   wxPostEvent(fenetre,wxPaintEvent()) # Demande un nouvel affichage
   wxYield() # Laisse du temps à l'affichage
```