Introduction à Cython

Loïc Gouarin

Laboratoire de Mathématiques d'Orsay

2 au 6 décembre 2013 - Biarritz

Python, c'est lent !

oui mais ...

- Python est un langage facile à apprendre.
- Il existe énormément de modules pour le calcul scientifique.
- On peut faire bien plus que du calcul scientifique.
- La communauté est gigantesque.

Reste donc aux développeurs le soin de faire l'optimisation de leurs codes pour avoir des utilisateurs conquis.

Comment optimiser son code

- trouver les régions lentes du code en faisant du profiling,
- optimiser ces parties en se ramenant à un langage bas niveau,
- l'interfaçage se fait via l'API Python/C.

Comment faire l'interface ?

- en écrivant tout seul l'interface,
- en utilisant SWIG,
- en utilisant f2py,
- en utilisant cython.

Cython : comment ça marche ?

- On écrit un fichier `.pyx` indiquant comment s'opère l'interfaçage entre les objets Python et du C.
- On utilise la commande `cython` pour créer l'interface `.c` écrite en API Python/C.
- On compile ce fichier `.c` en `.o`.
- On crée la librairie `.so` associée.
- On peut maintenant importer cette librairie dans Python et utiliser les fonctions ou classes ainsi créées.

Cython : la compilation ?

- Utiliser `cython` manuellement.
- Ecrire un fichier `setup.py` utilisant `distutils`.
- Utiliser `pyximport`.
- Utiliser Sage.

On suppose dans la suite que l'on a un fichier `piCython.pyx` dans lequel on a une fonction `calculPi` qui prend en entrée un entier et retourne un réel.

Exemple avec `setup.py`

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name = "calcul de pi",
      ext_modules = cythonize("piCython.pyx"),
      )
```

```
$ python setup.py build_ext --inplace
$ python
>>> from piCython import calculPi
>>> print calculPi(100)
```

Exemple avec `pyximport`

```
$ python
>>> import pyximport
>>> pyximport.install()
>>> from piCython import calculPi
>>> print calculPi(100)
```

Cython : principe général

Définir des types statiques permettant à Cython de comprendre que nous ne sommes plus dans une partie Python mais dans une partie pouvant être écrite facilement en C et donc optimisée.

Les mots-clefs `def`, `cdef`, `cpdef`

`def` : fonction utilisable par Python

```
def fact(n):
  res = 1
  for i in xrange(n + 1):
    res *= i
  return res
```

`cdef` : fonction utilisable en C

```
cdef long factc(int n):
  cdef:
    long res = 1
    int i
  for i in xrange(1, n + 1):
    res *= i
  return res
```

Les mots-clefs `def`, `cdef`, `cpdef`

`cpdef` : fonction utilisable en Python et en C

```
cdef long factc(int n):
  cdef:
    long res = 1
    int i
  for i in xrange(1, n + 1):
    res *= i
  return res
```

Premier exemple concret

On souhaite approcher π en calculant l'intégrale suivante

$$\pi = \int_0^1 f(x)dx \text{ avec } f(x) = \frac{4}{1+x^2}dx.$$

Pour ce faire, on approche l'intégrale en utilisant une méthode des rectangles

$$\pi \approx \frac{1}{n}\sum_{i=0}^{n-1} f(x_i), \text{ avec } x_i = \frac{i+0.5}{n} \text{ pour } i = 0, \cdots, n-1.$$

Premier exemple concret

```
import time

def f(x):
    return 4./(1 + x**2)

def calculPi(n):
    h, pi = 1./n, 0.
    for i in xrange(n):
        pi += f(h*(i+.5))
    return h*pi

if __name__=="__main__":
    n, nrep = 100000, 10
    t1 = time.time()
    for i in xrange(nrep):
        pi = calculPi(n)
    print pi, (time.time() - t1)/nrep
```

Passage à Cython

- Récupérer l'archive des TPs Cython se trouvant à l'adresse de l'école http://calcul.math.cnrs.fr/spip.php?article236.
- Recopier les fonctions `f` et `calculPi` du fichier `pi.py` dans un fichier `piCython.pyx`.
- Utiliser le fichier `setup.py` pour créer la librairie en utilisant la commande

```
python setup.py build_ext --inplace
```

- Importer la fonction `calculPi` se trouvant dans la librairie créée et tester dans le script `pi.py`.

La commande magique

- Faire `cython -a piCython.pyx`.
- Ouvrir dans votre navigateur le fichier `piCython.html` ainsi créé.

Quelques directives importantes

- boundscheck
  si `False`, Cython assume que les indices demandés existent (`True` par défaut).

- wraparound
  si `False`, Cython ne vérifie pas si l'indice est négatif (`True` par défaut).

- cdivision
  si `True`, Cython fait une division en C (`False` par défaut).

Comment les utiliser ?

- En ligne de commande

```
cython -X boundscheck=True monfichier.pyx
```

- A toutes les fonctions du fichier `.pyx`

```
#cython: boundscheck=True
```

- uniquement à une fonction

```
@cython.boundscheck(True)
def mafonction(...):
```

`cimport` et `.pxd`

Un fichier `.pxd` est un fichier de définitions comportant

- des types C,
- des fonctions C externes,
- des fonctions C définies dans le module,
- une partie d'un nouveau type.

On peut importer son contenu en utilisant `cimport` qui fonctionne exactement pareil que `import`.

Exemple

mod1.pyx

```
cdef int cadd(int i, int j):
    print 'cadd'
    return i+j

def padd(int i, int j):
    print 'padd'
    return i+j

cpdef int cpadd(int i, int j):
    print 'cpadd'
    return i+j
```

mod1.pxd

```
cdef int cadd(int i, int j)
cpdef int cpadd(int i, int j)
```

Exemple

mod2.pyx

```
cimport mod1 as cmod1
import mod1 as pmod1

def cadd(int i, int j):
    return cmod1.cadd(i, j)

def padd(int i, int j):
    return pmod1.padd(i, j)

def cpadd1(int i, int j):
    return cmod1.cpadd(i, j)

def cpadd2(int i, int j):
    return pmod1.cpadd(i, j)
```

Exemple

```
$ python
>>> import mod2
>>> mod2.padd(3, 4)
padd
7
>>> mod2.cadd(3, 4)
cadd
7
>>> mod2.cpadd1(3, 4)
cpadd
7
>>> mod2.cpadd2(3, 4)
cpadd
7
```

Les `.pxd` livrés avec Cython

Cython offre quelques fichiers `.pxd` de base.

exemple

```
from libc.math cimport cos, sin
cimport numpy
from libcpp.vector import vector
```

La liste est dans le répertoire `Includes` de l'installation de Cython.

Cython et les tableaux

Ancienne méthode

```
cimport numpy as cnp
import numpy as pnp
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
def sumOldBuffer(cnp.ndarray[cnp.int32_t, ndim=2] A):
  cdef int nx = A.shape[0], ny = A.shape[1]
  cdef int i, j
  cdef long int s=0

  for i in xrange(nx):
        for j in xrange(ny):
            s += A[i, j]

  return s
```

Cython et les tableaux

Nouvelle méthode : memory view

```
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
def sumMemoryView(int[:, ::1] A):
  cdef int nx = A.shape[0], ny = A.shape[1]
  cdef int i, j
  cdef long int s=0

  for i in xrange(nx):
        for j in xrange(ny):
            s += A[i, j]

  return s
```

Cython et les tableaux

Les fonctions précédentes ont été testées sur un tableau d'entiers de taille 1000×1000. On répète 100 fois les opérations.

méthode	temps (s)	rapport
sumMemoryView	0.034	1
sumOldBuffer	0.048	1.43
numpy.sum	0.12	3.53

Memory view

Création

```
cdef int [:, :, :] mv

#memory view d'un tableau numpy
mv = np.zeros((10, 10, 10), dtype=np.int32)

#memory view d'un tableau c
cdef int a[5][3][7]
mv = a

#memory view d'une autre memory view
cdef int [:, :, :, :] mv4d
mv = mv4d[1]
```

Accès

```
# index comme en numpy
mv[2, 3, 1]

# slice comme en numpy
mv [10, :, :] == mv[10] == mv[10, ...]
```

Memory view

Tableaux contigus

```
cdef int [:, :, ::1] mvc
mvc = np.zeros((10, 10, 10), dtype=np.int32)

cdef int [::1, :, :] mvf
mvf = np.zeros((10, 10, 10), dtype=np.int32
                order = 'F')
```

Le jeu de la vie

Pour illustrer l'utilisation de Cython avec des tableaux numpy, nous allons programmer le très connu **jeu de la vie**.

Rappel du principe

- Si une cellule a exactement trois voisines vivantes, elle est vivante à l'étape suivante.
- Si une cellule a exactement deux voisines vivantes, elle reste dans son état actuel à l'étape suivante.
- Si une cellule a strictement moins de deux ou strictement plus de trois voisines vivantes, elle est morte à l'étape suivante.

Le jeu de la vie

- Aller dans le répertoire jeuVie de l'archive des TPs Cython téléchargée précédemment.
- Lancer `gui.py`.
- Editer le fichier `jeuVie` et regarder les fonctions `cycleWithLoop`, `cycleVectorized`.
- Ecrire les fonctions cython qui font la même chose que ces 2 fonctions en utilisant
  1. l'ancienne version pour accéder aux tableaux numpy,
  2. les memory views.
- Comparer les temps d'exécution.

pyShift

- Aller dans le répertoire pyShift de l'archive des TPs Cython téléchargée précédemment.
- Cythonizer la version `cartTh` écrite en Python pur et en Numpy en utilisant
  - l'ancienne version pour les tableaux Numpy,
  - les memory views.
- Comparer les temps de calcul en utilisant le fichier `bench_cartTh` qui se trouve dans le répertoire `benchmark`.

Quelques liens

- Site web de cython
- Cython: Speed up Python and NumPy, Pythonize C, C++, and Fortran, Kurt Smith à SciPy2013
- memoryview benchmarks 1 sur le blog de Jake VanderPlas
- memoryview benchmarks 2 sur ce même blog