Interfaçage C–Python

Xavier Juvigny

ONERA/CHP

Décembre 2010

Plan

Motivations

- Performances Python vs. C
- Divers Interfaçages Python - C/Fortran

Plan

Motivations

- Performances Python vs. C
- Divers Interfaçages Python - C/Fortran

API C Python

- Présentation de l'API Python
- Application de l'API C à la suite de Syracuse

Plan

Motivations

- Performances Python vs. C
- Divers Interfaçages Python - C/Fortran

API C Python

- Présentation de l'API Python
- Application de l'API C à la suite de Syracuse

l'API C de numpy

- Présentation de l'API C de numpy
- Application à la suite de Syracuse

Plan

Plan

Problème modèle

Suite de Syracuse :

1. Choisir u_0 ;
2. Si u_k pair, $u_{k+1} \leftarrow \dfrac{u_k}{2}$;
3. Si u_k impair, $u_{k+1} \leftarrow 3 \times u_k + 1$

Conjecture : $\forall u_0 \in \mathbb{N}^\star, \exists n_0$ t.q $u_{n_0} = 1$

Définition : Temps de vol : $f(u_0) = \min_n \{n; u_n = 1\}$

But du programme :

Calculer les temps de vol $f(u_0)$ pour $1 \leq u_0 \leq N$, N fixé.

Problème modèle en C

```c
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>

long syracuse(long n){
  long compteur = 0L;
  while (n > 1){
    if ((n&1)==0) n /= 2; else n = 3*n+1;
    compteur ++;
  }
  return compteur;
}

int main()
{
  const long N = 1000000;
  double t1, t2;
  long i, *flights;

  flights = (long*)malloc(N*sizeof(long));
  for (i = 0; i < N; i++) flights[i] = syracuse(i+1);
  return EXIT_SUCCESS;
}
```

Problème modèle en Python

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
##########################
import numpy

def syracuse(n) :
    compteur = 0
    x = n
    while (x != 1) :
        if ((x&1)==0) : x /= 2
        else: x = 3*x+1
        compteur += 1
    return compteur

N = 1000000
flights = numpy.empty((N),numpy.int)
flights[:] = map(syracuse,xrange(1,N+1))
```

Performances

Langage	Temps d'exécution
C	0.30 sec.
Python	30.31 sec.
Python + UFunc	25.9

- *Programme python plus simple qu'en C*
- **Python offre de faibles performances** :
  Rapport de performance $\approx$ 100 !
- Comment avoir des performances équivalentes au C ?
- Comment appeler des fonctions C (ou FORTRAN) en Python ?
- Appeler du C directement en Python ;
- API Python en *C* : création d'un module Python en C.

Plan

Appeler des fonctions C provenant d'une bibliothèque dynamique

Un module python : `ctypes`

```c
/* Compilation :
 gcc -fPIC -shared -O3
     -o syrac.so syrac.c
*/
long syracuse(long n)
{
  long compteur = 0L;
  while (n > 1)
  {
    if ((n&1)==0)
      n /= 2;
    else
      n = 3*n+1;
    compteur ++;
  }
  return compteur;
}
```

Source C de `syrac.so`

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
########################
from ctypes import *
from numpy import *
import time

syracDLL=CDLL("./syrac.so")
syracuse = syracDLL.syracuse

N = 1000000
flights = empty((N),int)
t1 = time.time()
flights[:] =map(syracuse,
                xrange(1,N+1))
t2 = time.time()
```

Programme Python

Performance des `ctypes`

Langage	Temps d'exécution
Python	30.31 sec.
`c-types`	1.05 sec.
C	0.30 sec.

- Performances bien meilleures qu'en Python pur ;
- Permet d'appeler des fonctions définies dans des `DLL` sous windows ;
- Appel aux fonctions diffère selon le système d'exploitation employé !
- Syntaxe différente du reste du code Python (on apprend un "nouveau langage").

Utilisation de f2py

```
SUBROUTINE syracuse( n, f)
  IMPLICIT NONE
  INTEGER(8),INTENT(IN)::n
  INTEGER(8),INTENT(OUT)::f

  INTEGER(8) :: x

  x = n
  f = 0
  DO WHILE (x>1)
     IF (IAND(x,1).EQ.0) THEN
        x = x/2
     ELSE
        x = 3*x+1
     END IF
     f = f + 1
  END DO
  RETURN
END SUBROUTINE
```

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
########################
from numpy import *
import time
import FSyracuse

syracuse = FSyracuse.syracuse
N = 1000000
flights = empty((N),int)
flights[:] = map(syracuse,
                 xrange(1,N+1))
```

f2py -c Syracuse.f90 -m FSyracuse

Performances f2py

Langage	Temps d'exécution
Python	30.31 sec.
`c-types`	1.05 sec.
f2py	0.60 sec.
C/FORTRAN	0.30 sec.

- Bonnes performances ! (Temps dû à la boucle python ?)
- Simple d'emploi : ne demande aucune connaissance particulière autre que le Fortran et Python
- Interfaçage "intelligent" du fortran 90
- Livrée en standard avec numpy
- Contrôle fin de l'interface (mais alors moins simple d'emploi !)
- Ne permet d'interfacer que le Fortran

Swig

- Reprendre le source C utilisé pour l'utilisation des `ctypes`
- Création d'un fichier d'interfaçage :

```
%module syracuseC
%{
  extern long syracuse(long n);
%}
extern long syracuse(long n);
```

- Créer un fichier C `syracuse_wrap.c` et un fichier python `syracuseC.py` permettant l'interfaçage C-Python :

```
swig -python iterSyracuse.i
```

- Compiler ce fichier en bibliothèque dynamique :

```
gcc `python2.6-config --cflags` -fPIC -shared
    -O3 -o _syracuseC.so syracuse_wrap.c  syracuse.c
```

Performances de Swig

Langage	Temps d'exécution
Python	30.31 sec.
`c-types`	1.05 sec.
f2py	0.60 sec.
Swig	0.50 sec.
C/FORTRAN	0.30 sec.

- Très bonnes performances !
- Permet l'interfaçage avec d'autres langages que Python...
- Coupure nette entre C et Python
- Obligation d'apprendre un nouveau langage (pour définir l'interface).

Utilisation de l'API C de Python

- Python propose une API C pour créer des modules Pythons en C ;
- De documenter les modules Pythons dans le source C ;
- De créer de nouveaux types Pythons ;
- Supporte l'héritage de classes Python ;
- De voir les nouveaux types comme des conteneurs, des scalaires, . . .

Plan

Règles d'écritures de bases

- Le premier `include` d'un fichier utilisant l'API C de Python doit toujours être `Python.h` ;
- Le header est compatible C et C++ ;
- Toutes les définitions de l'API commencent soit par `Py` soit par `_Py` ;
- Les fonctions renvoyant à Python une valeurs doivent renvoyer un pointeur sur l'objet générique `PyObject` ;
- Tout objet python possède un compteur de référence permettant de savoir quand Python peut le détruire (quand le compteur est égal à zéro) ;
- Bien s'assurer d'utiliser la même version de Python que celle de l'API à la production.

Écriture d'une fonction C appelée par Python

On distingue trois parties distinctes :

1. La documentation Python de la fonction : chaîne de caractères (de préférence `static`). Exemple :

```
static char Syracuse_doc[] =
  "Calcul longueur de vol d'une suite de Syracuse.";
```

2. Une fonction (de préférence `static`) ayant pour signature

```
static PyObject* PySyracuse(PyObject* self,PyObject* args);
```

si la fonction attend des arguments en entrée ou

```
static PyObject* PySyracuse2(PyObject* self);
```

si la fonction n'attend pas d'arguments en entrée. !

3. L'interfaçage Python de la fonction et de sa doc. :

```
static PyMethodDef Syracuse_methods[] = {
  {"syracuse",(PyCFunction)PySyracuse,METH_VARARGS,
   PyDoc_STR(Syracuse_doc)},
  {"syracuse2",(PyCFunction)PySyracuse2,METH_NOARGS,
   PyDoc_STR(Syracuse2_doc)},
  {NULL,NULL}
};
```

Lecture des arguments passés par Python à une fonction

Une seule commande à retenir : `PyArg_ParseTuple`

- Analyse le `PyObject args` passé en argument à l'aide d'un `format` passé en chaîne de caractère ;
- Initialise des variables correspondant au format passé.
- Retourne 0 en cas d'erreur.

`format` contient une ou plusieurs unités :

Type Python	Format unitaire	Type(s) C
String	s	char *
String	s#	char * + int
Scalaire	i,l,f,d	int, long, float, double
Objet	O	PyObject*
Objet	O !	TypeObject + PyObject*
Tuple	(. . .)	Types définis dans Tuple
Optionnel	|	Types définis dans Tuple avec valeurs par défaut à définir.

Exemples de lectures d'arguments passés par Python

```
int ok, i, j, size;
double left, top, right, bottom, h, v;
char *s;

/* Appel python: f() */
ok = PyArg_ParseTuple(args, ""); /* Pas d'arguments */

/* Exemple appel python : f('hoops!') */
ok = PyArg_ParseTuple(args, "s", &s); /* Une string */

/* Une paire d'entier et un string avec sa taille
   Exemple appel Python : f((1, 2), 'trois') */
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
h = v = 10;
/* Un rectangle et un point.
   Exemples appel Python : f((0.,0.),(400.,300.), (10.,10.))
                           f((0.,0.),(400.,300.))
*/
ok = PyArg_ParseTuple(args, "((dd)(dd))|(dd)",
                      &left, &top, &right, &bottom, &h, &v);
```

Retourner des objets à Python

Principale fonction à retenir : `Py_BuildValue`

- Fonction “similaire” à `PyArg_ParseTuple` ;
- Prends le même format que `PyArg_ParseTuple` ;
- Mais convertit en Python les variables C et les retourne sous forme de tuple

Exemples :

```
return Py_BuildValue("");  // Retourne None
return Py_BuildValue("i",size);  // Retourne entier size
// Retourne les deux tuples (left,top) et (right,bottom)
return Py_BuildValue("(dd)(dd)",left,top,right,bottom);
// Retourne la liste [left,top,right,bottom]
return Py_BuildValue("[dddd]",left,top,right,bottom);
```

Écriture d'un module

Trois parties distinctes :

- Définition des fonctions appelables par Python (optionnel) ;
- Documentation du module :

```
PyDoc_STRVAR(ModSyracyse_doc, "Doc du Module Syracuse");
```

- Initialisation du module :

```
PyMODINIT_FUNC initSyracuse(void) {
  /* Creation du module */
  PyObject *m =
    Py_InitModule3("Syracuse", Syracuse_Methods,
                   ModSyracyse_doc );
  if ( m == NULL ) return;
}
```

Attention : Le nom de la fonction d'initialisation doit être en adéquation avec le nom du module et de la librairie :
`initSyracuse`, `Syracuse.so`, `import Syracuse`

Plan

Suite de Syracuse avec API C

```c
/** Interface python pour calculer une suite de Syracuse */
# include <Python.h>
# include "syracuse.h"

static char _Syracuse_doc[] =
  "Calcul la longueur de vol d'une suite de Syracuse.";
static PyObject* PySyracuse( PyObject* self, PyObject* args ) {
  long x;
  if (!PyArg_ParseTuple(args, "l", &x))  return NULL;
  return PyInt_FromLong(_syracuse(x));
}
/* ================================================== */
static PyMethodDef Syracuse_methods[] = {
  {"syracuse", (PyCFunction)PySyracuse, METH_VARARGS,
   PyDoc_STR(_Syracuse_doc)},
  {NULL,NULL}
};
/* ================================================== */
PyDoc_STRVAR(Syracuse_doc, "Calcul vol suite de Syracuse.");
//
PyMODINIT_FUNC
initSyracuse(void) {
  return Py_InitModule3("Syracuse", Syracuse_methods,
                        Syracuse_doc );
}
```

Compilation et utilisation en Python

- Compilation du source C :

```
gcc `python2.6-config --cflags` -fPIC -shared
    -O3 -o Syracuse.so PySyracuse.c  syracuse.c
```

- Utilisation dans un code python :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
##########################
import time
import Syracuse

N = 1000000
flights = numpy.empty((N),numpy.int)
t1 = time.time()
flights[:] = map(Syracuse.syracuse, xrange(1,N+1))
t2 = time.time()
print "Vols : ", flights
print "Temps mis : %7.5f secondes"%(t2-t1)
```

Performance du module API C

Langage	Temps d'exécution
Python	30.31 sec.
`c-types`	1.05 sec.
f2py	0.60 sec.
Swig	0.50 sec.
API C	0.47 sec.
C/FORTRAN	0.30 sec.

- Code API relativement simple ;
- Définition d'une interface python souple via le C ;
- Temps équivalent à Swig ;
- Mise en place de la documentation aisée ;
- Inutile d'apprendre un nouveau langage : programmation en C/C++ ;

Plan

Introduction

L'API C de numpy permet :

- De créer des tableaux numpy ;
- De récupérer des tableaux numpy ;
- De lire leurs dimensions, propriétés (arrangement Fortran ou non, détruit par Python ou non, etc. . .)
- Deux nouveaux types python :
  1. `PyArray_Type` qui correspond aux tableaux numpy : inclure `numpy/arrayobject.h` après `Python.h` ;
  2. `PyUFunc_Type` qui correspond aux ufunc de numpy : inclure `numpy/ufuncobject.h` après `Python.h`.

Récupération d'un tableau numpy

Récupérer un tableau numpy dans l'API C à l'aide de `PyArg_ParseTuple` en s'assurant de son type :

```
PyArrayObject *numArray;
if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &numArray))
  return NULL;
```

Puis interrogation du tableau numpy :

```
// Nombre de dimensions du tableau
int ndim = PyArray_NDIM(numArray);
bool isColumnMajor = false;
if (ndim==2) isColumnMajor = PyArray_ISFORTRAN(numArray);
// Recuperation des dimensions :
int dim0 = PyArray_DIM(numArray,0);
int dim1 = 0;
if (ndim==2) dim1 = PyArray_DIM(numArray,1);
double* coefs = NULL;
// Verification du type d'elements dans le tableau
if (PyArray_TYPE(numArray) != NPY_DOUBLE) return NULL;
// Recuperation des coefficients du tableau
coefs = PyArray_DATA(numArray);
```

Accès aux données d'un tableau numpy

Quelques fonctions sont proposées pour accéder aux données d'un tableau numpy :

```
npy_intp ind[2] = {3,4};
// Retourne le coefficients numArray(3,4) :
double A34 = *(double*)(PyArray_GetPtr(numArray,ind);
// Retourne un pointeur sur le 3e ligne de numArray
double* ligne3 = (double*)PyArray_GETPTR1(numArray,3);
// Retourne un pointeur sur  coefficient en 3e ligne, 4e colonne
double* ptA34 = (double*)PyArray_GETPTR2(numArray,3,4);
```

Il existe également des fonctions `PyArray_GETPTR3` et `PyArray_GETPTR4` pour accèder à des données d'un tableau de dimensions respectivement supérieures à 3 et 4.

Création d'un tableau numpy

Plusieurs fonctions pour créer un tableau numpy :

- Créer un tableau à *n* dimensions :

```
npy_intp nd[2]= {200,3};
pts=(PyArrayObject*)PyArray_SimpleNew(2,nd,NPY_DOUBLE);
...
return pts;
```

- Créer un tableau à *n* dimensions pointant sur les données passées en paramêtre :

```
double coords[200][3] = ...;
const int ndim = 2;
npy_intp nd[ndim] = {200,3};
pts=(PyArrayObject*)PyArray_SimpleNewFromData(ndim,nd,
                                              NPY_DOUBLE, coords);
return pts;
```

- Création d'un tableau contenant des 0 :

```
npy_intp nd[2] = {200,3};
// Tableau stocke a la fortran
for_array =(PyArrayObject*)PyArray_ZEROS(2,nd,NPY_DOUBLE,1);
// Tableau stocke a la C
c_array   =(PyArrayObject*)PyArray_ZEROS(2,nd,NPY_DOUBLE,0);
```

Création d'un tableau numpy...

- Création d'un tableau à partir d'un ensemble (liste, tuple, ...) :

```
PyTupleObject* tuple;
// Transforme le tuple en un tableau a une ou deux
// dimensions (selon que le tuple contient lui-meme des
// tuples ou non...
PyArray_ContiguousFromAny(tuple, NPY_DOUBLE, 1, 2);
```

  - Si l'objet est un tableau numpy, on se contente de faire référence au même tableau.
  - Si l'objet est une autre séquence, on recopie ses valeurs dans un nouveau tableau.

Importation de l'API numpy

Afin d'utiliser dans son propre module l'API C de numpy, il faut s'assurer d'avoir appeler `import_array` à l'initialisation de son module.

```
PyMODINIT_FUNC initSyracuse(void) {
  PyObject *m;
  /* Creation du module */
  m = Py_InitModule3("Syracuse",Syracuse_methods,Syracuse_doc);
  if ( m == NULL ) return;
  // Important : initialise la bibliotheque numpy
  import_array();
}
```

Plan

Amélioration du module

- Le module C est plus lent qu'un programme C/Fortran à cause de la boucle en Python ;
- Il peut être intéressant de créer une fonction dans le module permettant d'itérer sur les diverses valeurs de u_0 ;
- On doit stocker les résultats dans un tableau ;
- Pour cela, on va utiliser l'API C proposé avec numpy !
- Cette API permet de :
  1. Récupérer des tableaux provenant de codes Fortran/C/C++
  2. Créer des tableaux python dynamiquement en C ; qu'on récupère sous forme de tableaux numpy (sans que Python cherche à les détruire) ;
  3. De préciser pour les tableaux multi–dimensionnels si le stockage est selon la convention Fortran ou C.

Version optimale

```
# include <Python.h>
# include <numpy/arrayobject.h>
# include "syracuse.h"

static char flight_doc[] = "Temps de vol pour i allant de 1 a n";
static void _flightSyracuse(long n, int* flights) {
  for (long i = 0; i < n; i++) flights[i] = _syracuse(i+1);
}

static PyObject* flightSyracuse( PyObject* self, PyObject* args)
{
  PyArrayObject *py_flight = NULL;
  long n;
  if (!PyArg_ParseTuple(args, "l", &n)) return NULL;
  if (n==0) Py_RETURN_NONE;
  if (n>0) {
    int *pFl;
    npy_intp nd[1]; nd[0] = n;
    py_flight =
      (PyArrayObject*)PyArray_SimpleNew(1, nd, PyArray_INT);
    if (!py_flight) return NULL;
    _flightSyracuse(n,(int*)PyArray_DATA(py_flight));
  }
  return (PyObject*)py_flight;
}
```

Version optimale (Suite)

```
static PyMethodDef Syracuse_methods[] = {
  {"flight", (PyCFunction)flightSyracuse, METH_VARARGS,
   PyDoc_STR(flight_doc)},
  {NULL,NULL}
};
/* ========================*/
PyDoc_STRVAR(Syracuse_doc, "Calcul vol suite de Syracuse.");
PyMODINIT_FUNC initSyracuse(void) {
  PyObject *m;
  /* Creation du module */
  m = Py_InitModule3("Syracuse",Syracuse_methods,Syracuse_doc);
  if ( m == NULL ) return;
  // Important : initialise la bibliotheque numpy
  import_array();
}
```

Programme python associé :

```
import numpy
import Syracuse
N = 1000000
flights = Syracuse.flight(N);
print "Vols : ", flights
```

Performances de la version optimisée

Langage	Temps d'exécution
Python	30.31 sec.
`c-types`	1.05 sec.
f2py	0.60 sec.
Swig	0.50 sec.
API C	0.47 sec.
numpy + API C	0.30 sec.
C/FORTRAN	0.30 sec.

- Même vitesse que le C ou le Fortran !
- Gestion de la mémoire faite par Python au travers de l'API C
- Pour utiliser l'API numpy, ne pas oublier le `import_array` à l'initialisation du module (sinon plantage) ;
- De nombreuses fonctions pour créer des tableaux numpy existent !

Gestion du compteur de référence

- Le programmeur peut gérer explicitement le compteur de référence d'un object à l'aide des fonctions `Py_INCREF` et `Py_DECREF` ;
- A la création d'un objet python, le compteur est mis à 1 ;
- A l'appel de son destructeur, le compteur est décrémenté et l'objet détruit si son compteur est nul ;
- Lorsqu'on extrait un objet d'une liste ou d'un tuple python, il faut incrémenter le compteur ;
- Certaines fonctions (par exemple, insertion dans une liste ou un tuple) "volent" le compteur de référence (ne l'incrémente pas) ;
- Toujours regarder dans la documentation de l'API comment est géré le compteur de référence (bien documenté).

Premier exemple de gestion du compteur de référence

```
// unObjetPython est un objet recupere de Python
PyObject* unObjetPython;

PyObject* tuple;
t = PyTuple_New(3);
// PyTuple_SetItem vole les references
// Pour les deux premiers elements, aucun probleme, on
// cree deux objets pythons dont les references sont
// volees par le tuple qui les detruira donc a sa destruction.
PyTuple_SetItem(t, 0, PyInt_FromLong(101L));
PyTuple_SetItem(t, 1, PyString_FromString("Hello World"));
// pour unObjetPython, il faut incrementer le compteur
// pour qu'il ne soit pas detruit lors de la destruction
// du tuple.
Py_INCREF(unObjetPython);
PyTuple_SetItem(t,2, unObjetPython);
```

Deuxième exemple de gestion du compteur de référence

```
  // Des objets a renvoyer a Python
  PyObject *obj1, *obj2, *obj3;
  obj1 = obj2 = obj3 = NULL;

  obj1 = PyString_FromString("Hello World");
  ...
  obj2 = PyInt_FromLong(101L);
  ...
  if (uneErreur) goto error;

  obj3 = Py_BuildValue("(iis)",1,2,"three");
  ...
  return Py_BuildValue("OOO",obj1,obj2,obj3);
error:
  Py_XDECREF(obj1);
  Py_XDECREF(obj2);
  Py_XDECREF(obj3);
  return NULL;
}
```