F2PY

Interface avec le langage Fortran

Pierre Navaro

Institut de Recherche Mathématique Avancée, Strasbourg

Autrans 6-10 décembre 2010

Introduction

Python possède un certain nombre d'avantages :

- Langage simple, interprété avec une syntaxe claire.
- Facile à programmer et la gestion de la mémoire est automatique.
- Open source, gratuit et portable.
- Dispose de nombreux modules pour le calcul scientifique.

Il est néanmoins trop lent pour les taches numériques intensives. Dans ce cas on peut l'interfacer avec C ou Fortran. Cette technique permet d'utiliser le meilleur des deux mondes :

- Taches de pré et post-traitement en Python.
- Parties numériques intensives en Fortran (ou C)

f2py

f2py nécessite l’installation de Numpy et trois méthodes pour créer une interface sont proposées :

- Interfacer des subroutines simples sans écrire de code supplémentaire.
- Ajouter des directives pour f2py dans le source Fortran pour un interfaçage plus complexe.
- Écrire un fichier d’interface décrivant les subroutines et les données à interfacer. f2py génère automatiquement un fichier d’interface simple qu’on peut ensuite éditer et modifier.

Ressources

- *Site f2py* `http://cens.ioc.ee/projects/f2py2e/`
- *Transparents E. Sonnendrücker* `http://calcul.math.cnrs.fr/Documents/Journees/dec2006/python-fortran.pdf`
- *SciPy* `http://www.scipy.org/F2py`
- *Documentation Sagemath* `http://www.sagemath.org/doc/numerical_sage/f2py.html`
- *Wiki de l'IRMA* `http://www-irma.u-strasbg.fr/irmawiki/index.php/Utilisation_de_F2PY`
- Hans Petter Langtangen.
  Python Scripting for Computational Science.
  Springer 2004

Exemple de subroutine simple

Calcul de la norme.

Fortran 90/95 format libre

```
subroutine norme (a, b, c)
real(8), intent(in) :: a, b
real(8), intent(out) :: c
c= sqrt (a*a+b*b)
end subroutine norme
```

Fortran 77 format fixe

```
      subroutine norme (a, b, c)
      real*8 a,b,c
Cf2py intent(out) c
      c=sqrt (a*a+b*b)
      end
```

Compilation et exécution

- Génération de l'interface Python avec f2py

```
f2py -c norme.f90 -m vect  --fcompiler=gnu95 --f90flags=-O3
```

- Appel depuis un shell Python

```
>>> import vect
>>> vect.norme(3,4)
5.0
>>> c = vect.norme(3,4)
>>> c
5.0
```

- Documentation générée automatiquement par f2py

```
>>> print vect.norme.__doc__
norme - Function signature :
c = norme(a,b)
Required arguments :
a : input float
b : input float
Return objects :
c : float
```

Ajout de directive f2py dans le source Fortran

Ces ajouts dans le code source fortran permettent préciser le rôle et la définition des variables d'entrés-sorties. Sont utilisés :

- Les attributs du F90 : `intent(in), dimension(2,3).`
- Les attributs spécifiques : `!f2py intent(hide), depend(a).`

```
subroutine norme(a,c,n)
integer :: n
real(8),dimension(n),intent(in) :: a
!f2py optional , depend(a) :: n=len(a)
real(8),intent(out) :: c
real(8) :: sommec
integer :: i
sommec = 0
do i=1,n
   sommec=sommec+a( i )*a( i )
end do
c=sqrt (sommec)
end subroutine norme
```

Liste Python ou tableau numpy en argument

```
>>> from vect import *
>>> a=[2,3,4]  #Une liste Python
>>> type(a)
<type 'list'>
>>> norme(a)
5.3851648071345037
>>> from numpy import *
>>> a=arange(2,5)  # Un tableau numpy
>>> type(a)
<type 'numpy.ndarray'>
>>> norme(a)
5.3851648071345037
>>> print norme.__doc__  # Documentation
norme - Function signature :
c = norme(a,[n])
Required arguments :
a : input rank-1 array('d') with bounds (n)
Optional arguments :
n := len(a) input int
Return objects :
c : float
```

Utilisation d'un fichier signature

- On peut générer automatiquement un fichier signature

```
f2py vecteur.f90 -h vecteur.pyf
```

- Contenu de vecteur.pyf

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

subroutine norme(a,c,n) ! in norme.f90
    real(kind=8) dimension(n),intent(in) :: a
    real(kind=8) intent(out) :: c
    integer optional,check(len(a)>=n),depend(a) :: n=len(a)
end subroutine norme

! This file was auto-generated with f2py (version:2).
! See http://cens.ioc.ee/projects/f2py2e/
```

Appel d'une fonction Python depuis Fortran

```
subroutine sommef (f ,n,s)
!Calcule la somme (f(i), i=1,n)
external f
integer, intent(in) :: n
real, intent(out) ::s
s=0.0
do i=1,n
   s=s+f(i)
end do
end subroutine sommef
```

```
>>>from vect import *
>>>def fonction(i) : return(i*i)
>>>sommef(fonction,3)
14.0
>>>sommef(lambda x :x**3,3)
36.0
```

Les tableaux multi dimensionnels

```
subroutine move( positions, vitesses, dt, n)
integer, intent(in) :: n
real(8), intent(in) :: dt
real(8), dimension(n,3), intent(in) :: vitesses
real(8), dimension(n,3) :: positions
do i = 1, n
  positions(i,:) = positions(i,:) + dt*vitesses(i,:)
end do
end subroutine move
```

```
>>> print vitesses
[[0, 1, 2], [0, 3, 2], [0, 1, 3]]
>>> print positions
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> move(positions, vitesses, 0.1)
>>> print positions #le tableau n'est pas mis a jour, stockage C
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> positions = numpy.array(positions, dtype='f8', order='F')
>>> move(positions, vitesses, 0.1)
>>> print positions #le tableau est modifie, stockage Fortran
[[ 0.   0.1  0.2]
 [ 0.   0.3  0.2]
 [ 0.   0.1  0.3]]
```

Quelques directives

- *optional*, *required*. Cet attribut est automatique lorsqu'un argument possède une valeur par défaut.
- *intent*(*in*|*inout*|*out*|*hide*) , *intent*(*in*) par défaut.
- *intent*(*out*) devient automatiquement *intent*(*out*, *hide*).
- *intent*(*copy*) et *intent*(*overwrite*) permet de contrôler les changements dans les arguments d'entrée.
- *check* permet de vérifier les arguments. Souvent géré automatiquement par f2py.
- *depend* : f2py détecte les dépendances cycliques.
- *allocatable*, *parameter*
- *intent*(*callback*), *external* : pour les arguments de type fonction.
- *intent*(*c*) argument de type C, tableau ou fonction.
- Expressions C : *rank*, *shape*, *len*, *size*, *slen*.

Module f2py2e et distutils

```
>>> import f2py2e
>>> fsource = '''
...       subroutine foo
...       print*, "Hello world!"
...       end
... '''
>>> f2py2e.compile(fsource,modulename='hello',verbose=0)
0
>>> import hello
>>> hello.foo()
 Hello world!
```

Exemple de setup.py

```
#!/usr/bin/env python
from numpy.distutils.core import Extension
ext1 = Extension(name = 'scalar',
                 sources = ['scalar.f'])
ext2 = Extension(name = 'fib2',
                 sources = ['fib2.pyf','fib1.f'])
if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(name = 'f2py_example', ext_modules = [ext1,ext2])
```

Appel d'une fonction lapack avec F2PY

```
f2py -m mylapack -h degmm.pyf dgemm.f
```

```
python module mylapack ! in
interface  ! in :mylapack
subroutine dgemm(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
character*1 :: transa, transb
integer :: m, n, k
double precision :: alpha, beta
double precision dimension(lda,*) :: a
integer optional,check(shape(a,0)==lda),depend(a) :: lda=shape(a,0)
double precision dimension(ldb,*) :: b
integer optional,check(shape(b,0)==ldb),depend(b) :: ldb=shape(b,0)
double precision dimension(shape(a,0),shape(b,1)), intent(out) :: c
integer optional,check(shape(c,0)==ldc),depend(c) :: ldc=shape(c,0)
end subroutine dgemm
end interface
end python module mylapack
```

Édition de liens avec lapack

f2py permet d'utiliser une bibliothèque écrite en Fortran sans modifier les sources.

```
f2py -c degmm.pyf -llapack
```

```
>>> import numpy
>>> import mylapack
>>> a = numpy.array([[7,8],[3,4],[1,2]])
>>> b = numpy.array([[1,2,3],[4,5,6]])
>>> c = mylapack.dgemm('N','N',3,2,2,1.0,a,b,1.0)
>>> print c
[[ 39.  54.  69.]
 [ 19.  26.  33.]
 [  9.  12.  15.]]
>>> print numpy.dot(a,b) #fonction numpy pour le produit de matrices
[[39 54 69]
 [19 26 33]
 [ 9 12 15]]
```

Les tableaux alloués dans le script Python seront alloués dans le module Fortran

```
module f90module
   implicit none
   real(8), dimension(:),   allocatable :: farray
contains
   subroutine init( n ) !Allocation du tableau farray
   integer, intent(in) :: n
   allocate(farray(n))
   end subroutine init
end module f90module
```

f2py -m f90mod -c f90module.f90

```
>>> import f90mod as F
>>> F.f90module.init(10)
>>> len(F.f90module.farray)
10
```

Les tableaux alloués dans le module Fortran seront alloués dans le script Python

```
module f90module
   implicit none
   real(8), dimension(:),   allocatable :: farray
contains
   subroutine test_array()
   print*, allocated(farray), size(farray)
   end subroutine test_array
end module f90module
```

f2py -m f90mod -c f90module.f90

```
>>> import numpy
>>> import f90mod
>>> f90mod.f90module.farray = numpy.random.rand(10).astype(numpy.float64)
>>> f90mod.f90module.test_array()
 T          10
```

Encapsulation de données Fortran dans une classe Python

```
module f90module
   real(8), dimension(:),   allocatable :: masses
   real(8), dimension(:,:),   allocatable :: positions, velocities
contains
 subroutine init_data( n )
 ...
 end subroutine init_data

 subroutine calc_forces( forces, n)
 ...
 end subroutine calc_forces
end module f90module
```

La classe Python

```
class SolarSystem(object):
   def __init__(self, masses, positions, velocities):
      self.index       = 0
      self.numberof    = len(masses)
      self._masses     = masses
      self._positions  = positions
      self._velocities = velocities
   @property
   def mass (self):
      return self._masses[self.index]
   @property
   def position (self):
      return self._positions[self.index]
   @property
   def velocity (self):
      return self._velocities[self.index]
```

Accès aux données Fortran

```
>>> import f90mod as F
>>> seystem = SolarSystem(F.f90module.masses, F.f90module.positions,F.f90modul
>>> system.index = 0
>>> system.mass
1800262.614350586
>>> system.position
array([-1.24902032,  0.02831779,  0.04840533])
>>> system.index = 1
>>> system.mass
0.29846375576171874
>>> system.position
array([-54.8958322, -34.85936287,-13.02314489])
```

Encapsulation de données Fortran dans une classe Python

```
class SolarSystemList(object):
   def __init__(self, solarsystem):
      self.solarsystem = solarsystem
      self.numberof = solarsystem.numberof
   def __getitem__(self, index):
      self.solarsystem.index = index
      return self.solarsystem
   def __len__(self):
      return self.numberof
   def __iter__(self):
      for i in xrange(self.numberof):
         self.solarsystem.index=i
         yield self.solarsystem
```

Encapsulation de données Fortran dans une classe Python

```
>>> systemList = SolarSystemList(system)
>>> systemList[0].mass
1800262.614350586
>>> systemList[0].position
array([-1.24902032,  0.02831779,  0.04840533])
```

Nous pouvons ajouter des fonctions (conversion, nouvelles caractéristiques, representation graphique,...)

```
class MySolarSystem(SolarSystem):
    def density(self):
        return self.mass / ( 4./3.*N.pi* self.radius **3)


>>>myss = MySolarSystem(F.f90wrap.masses,F.f90wrap.positions, \
...    F.f90wrap.velocities, F.f90wrap.radii)
>>>myssList = SolarSystemList(myss)
>>>print myssList[3].density()
5138322.1813
```

Types dérivés Fortran 90

```
module mesh
implicit none
type :: geometry
   real(8) :: x0, x1, dx    ! coordinates of origin and grid size
   integer :: nx               ! number of grid points
   real(8), dimension(:), pointer :: xgrid  ! coordinates of points
end type geometry

contains

subroutine create(geom,x0,nx,dx)
!f2py integer, intent(out) :: geom
type(geometry), pointer :: geom
real(8), intent(in) :: x0, dx
integer, intent(in) :: nx
integer :: i
allocate(geom)
geom%x0=x0; geom%x1=x0+nx*dx; geom%dx=dx; geom%nx=nx
allocate(geom%xgrid(nx))
do i=1,nx
   geom%xgrid(i)=geom%x0+(i-1)*geom%dx
end do
end subroutine create
```

Types dérivés Fortran 90

```
subroutine view(this)
!f2py integer, intent(in) :: this
type(geometry),pointer :: this
print*, 'nx = ', this%nx
print*, this%x0,this%x1
print*, this%xgrid(:)
end subroutine view

end module mesh
```

```
>>> import test
>>> geom = test.mesh.create(0.0,10,0.1)
>>> test.mesh.view()
>>> test.mesh.view(geom)
 nx =            10
0.0000000000000000       1.0000000000000000        0.0000000000000000
0.1000000000000001       0.2000000000000001       0.3000000000000004
0.4000000000000002       0.5000000000000000       0.6000000000000009
0.7000000000000007       0.8000000000000004       0.9000000000000002
```

f2py dans un notebook SAGEMATH

```
%fortran
!f90
subroutine norme(a,b,c)
real(8), intent(in) :: a, b
real(8), intent(out) :: c
c = sqrt(a*a+b*b)
end
```

```
print norme(3,4)
5.0
```

```
fortran.add_library('lapack')
fortran.add_library('blas')
fortran.add_library_path('/usr/lib')
```

Les fonctionalités de F2PY

- Gère tous les types du Fortran.
- Fonctionne avec le F77, le F90 et également des fonctions C.
- Gère les common du F77, les modules du F90 ainsi que les tableaux alloués dynamiquement.
- Permet d'appeler des fonctions Python depuis du C ou du Fortran (callback)
- Gère les différences de stockage mémoire du C et du Fortran.
- Génère la documentation Python
- Compilateurs supportés : GNU, Portland, Sun,...
- F2py fait partie intégrante de Scipy.

Limite principale actuelle :

- Ne gère pas de manière simple les types dérivées Fortran et les pointeurs.