Simulation de système solaire en Python

Version F2PY des exercices de Konrad Hinsen par Pierre Navaro

9 décembre 2010

Le problème consiste à calculer les trajectoires des planètes du système solaire.

1 Lecture des données initiales

Nous reprenons les lignes de python de la version numpy pour lire le fichier de données et initialiser les tableaux :

```
from Scientific.Geometry import Vector
import itertools
import numpy as N
AU = 149.6
h = 3600e-5
G = 6.67428e-4
MU = 0.45359237e4
delta_t = 0.05

names ,positions ,velocities ,masses , radii = [], [], [], [], []

def group(seq, n):
    return itertools.izip(*[itertools.islice(seq, i, None, n)  for i in range(n)])

for data in group(file('systeme_solaire').readlines(), 5):
    names.append(data[0].strip())
    positions.append(Vector(*[float(x) for x in data[1].split()])*AU)
    velocities.append(Vector(*[float(x) for x in data[2].split()])*AU/h)
    masses.append(float(data[3].strip())*MU)
    radii.append(float(data[4].strip())*AU/2.)

#Listes Python -> tableaux numpy
masses = N.array(masses)
positions = N.array(positions)
velocities = N.array(velocities)
radii = N.array(radii)
```

Dans la suite des exercices on considère que cette partie du code Python est effectuée au préalable.

2 Calcul du centre de masse dans une subroutine

Nous allons remplacer la fonction python calculant le centre de masse du système par une subroutine Fortran.

1. Editez le fichier f90routines.f90 :

```
subroutine center_of_mass(masses, positions, s, n)
implicit none
integer, intent(in) :: n
real(8), dimension(n), intent(in)   :: masses
real(8), dimension(n,3), intent(in) :: positions
real(8), dimension(3), intent(out)  :: s
integer :: i
real(8) :: m
s(:) = 0.0
do i = 1, n
   s(:) = s(:) + masses(i) * positions(i,:)
end do
m = sum(masses)
s(:) = s(:) / m
end subroutine center_of_mass
```

2. Compilez le module

```
f2py -c f90routines.f90 -m f90mod
```

3. Testez votre fonction f2py en comparant le résultat avec la version numpy :

```
>>> print center_of_mass(masses, positions)
[-0.00088762  0.00157605  0.00066594]
>>> import f90mod
>>> print f90mod.center_of_mass(masses, positions)
[-0.00088762  0.00157605  0.00066594]
```

On remarque que les arguments `n` et `s` sont devenus optionnels dans l'appel Python. f2py analyse la source Fortran et détecte que `n` est la dimension de l'un des tableaux d'entrée et que `s` est la valeur de sortie.

3 Calcul des forces dans un module

Le calcul des forces est effectué maintenant par une subroutine implémentée dans un module Fortran 90. Éditez le fichier f90module.f90 :

```
module f90module
implicit none
real(8) :: G
contains
subroutine calc_forces(masses, positions, forces, n)
```

```
integer, intent(in) :: n
real(8), dimension(n),   intent(in) :: masses
real(8), dimension(n,3), intent(in) :: positions
real(8), dimension(n,3), intent(out) :: forces
real(8) :: r_ij(3), f_ij(3), r_ij_abs
integer :: i, j

forces(:,:) = 0
do i = 1, n
   do j = i+1, n
      r_ij(:) = positions(i,:)-positions(j,:)
      r_ij_abs = sqrt( dot_product(r_ij,r_ij))
      f_ij(:) = G*masses(i)*masses(j)*r_ij(:)/r_ij_abs**3
      forces(i,:) = forces(i,:) - f_ij(:)
      forces(j,:) = forces(j,:) + f_ij(:)
   end do
end do
end subroutine calc_forces
end module f90module
```

Compilez votre nouveau module f2py :

```
f2py -m f90mod -c f90module.f90 f90routines.f90
```

Testez votre module en utilisant la fonction f2py pour le calcul des forces.
On modifiera la fonction `move` :

```
import f90mod as F
F.f90module.G = G

def move(masses, positions, velocities, time):
    n = len(masses)
    assert positions.shape == (n, 3)
    assert velocities.shape == (n, 3)
    assert time >= 0
    steps = int(time/delta_t)
    forces = F.f90module.calc_forces(masses, positions)
    for k in range(steps):
        v_midstep = velocities + 0.5*delta_t*forces/masses[:, N.newaxis]
        positions[:] = positions + delta_t*v_midstep
        for i in range(n):
            spheres[i].pos = positions[i]
            curves[i].append(pos = positions[i])
        forces = F.f90module.calc_forces(masses, positions)
        velocities[:] = v_midstep + 0.5*delta_t*forces/masses[:, N.newaxis]

>>> print F.center_of_mass(masses, positions)
[-0.00088762  0.00157605  0.00066594]
>>> move(masses, positions, velocities, 365*24*h)
>>> print F.center_of_mass(masses, positions)
[-0.00086632  0.00100156  0.00067109]
```

Remarque : Il faut être prudent lorsque l'on utilise des variables désignées par des majuscules en Fortran car la version en minuscule représentera la même variable. En Python les lettres majuscules et minuscules désigne des variables différentes. Dans cet exemple `F.f90module.g = F.f90module.G` alors que `G` $\neq$ `g`.

4 Calcul des trajectoires

Utilisons maintenant le Fortran pour calculer les trajectoires, éditez le fichier f90wrap.f90 où les données sont des tableaux "allocatable" du module :

```
module f90wrap
implicit none
real(8) :: G, delta_t
real(8), dimension(:),   allocatable :: masses, radii
real(8), dimension(:,:), allocatable :: positions, velocities

contains

subroutine calc_forces(forces, n)
integer, intent(in) :: n
real(8), dimension(n,3), intent(out) :: forces
real(8) :: r_ij(3), f_ij(3), r_ij_abs
integer :: i, j

forces(:,:) = 0
do i = 1, n
   do j = i+1, n
      r_ij(:) = positions(i,:)-positions(j,:)
      r_ij_abs = sqrt( dot_product(r_ij,r_ij))
      f_ij(:) = G*masses(i)*masses(j)*r_ij(:)/r_ij_abs**3
      forces(i,:) = forces(i,:) - f_ij(:)
      forces(j,:) = forces(j,:) + f_ij(:)
   end do
end do
end subroutine calc_forces

subroutine move(time, n)
integer :: steps, i, k
real(8), intent(in) :: time
integer, intent(in) :: n
real(8), dimension(n,3) :: forces
real(8), dimension(n,3) :: v_midstep

steps = int(time/delta_t)
call calc_forces(forces, n)
do k = 1, steps
```

```
   do i = 1, n
      v_midstep(i,:) = velocities(i,:) + 0.5*delta_t*forces(i,:)/masses(i)
      positions(i,:) = positions(i,:) + delta_t*v_midstep(i,:)
   end do
   call calc_forces(forces, n)
   do i = 1, n
      velocities(i,:) = v_midstep(i,:) + 0.5*delta_t*forces(i,:)/masses(i)
   end do
end do
end subroutine move
end module f90wrap
```

Compilation

```
f2py -m f90mod -c f90routines.f90 f90module.f90 f90wrap.f90
```

On s'assure que les tableaux numpy ont un stockage Fortran pour la mise à jour du tableau des positions.

```
>>> import f90mod as F
>>> F.f90wrap.G = G
>>> F.f90wrap.delta_t = delta_t
>>> F.f90wrap.masses = N.array(masses,dtype='f8')
>>> F.f90wrap.positions = N.array(positions,dtype='f8',order='F')
>>> F.f90wrap.velocities = N.array(velocities,dtype='f8',order='F')
>>> print F.center_of_mass(F.f90wrap.masses, F.f90wrap.positions)
[-0.00088762  0.00157605  0.00066594]
>>> n = len(masses)
>>> F.f90wrap.move(365*24*h,n)
>>> print F.center_of_mass(F.f90wrap.masses, F.f90wrap.positions)
[-0.00086632  0.00100156  0.00067109]
```

f2py assure la conversion des tableaux multidimensionnels au format C vers le format Fortran, cette étape de conversion nécessite une copie qui peut s'avérer pénalisante dans le cas de tableaux de grande taille.

5 Programmation objet avec f2py

Utilisons les classes Python pour encapsuler les données du module Fortran f90wrap.

```
class SolarSystem(object):
   def __init__(self, masses, positions, velocities, radii):
      self.index      = 0
      self.numberof   = len(masses)
      self._masses    = masses
      self._positions  = positions
      self._velocities = velocities
      self._radii     = radii

   @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]
   @property
   def radius (self):
      return self._radii[self.index]
```

Vérifiez le fonctionnement de la classe

```
>>>import f90mod as F
>>> ss = SolarSystem(F.f90wrap.masses,F.f90wrap.positions, \
... F.f90wrap.velocities, F.f90wrap.radii)
>>> ss.index=3
>>> ss.position
array([ -27.224064  ,  132.87468503,   57.64483778])
>>> ss.mass
5.4734986941894528
```

Donnons à notre objet `SolarSystem` les propriétés d'une liste Python avec cette nouvelle classe :

```
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
```

Nous pouvons maintenant facilement manipuler les données du module Fortran. Testez cet exemple pour visualiser les résultats :

```
>>> system = SolarSystemList(ss)
>>> import visual
>>> spheres = [visual.sphere(pos=visual.vector(p[0], p[1], p[2]),
...                          radius = 10., color = visual.color.yellow)
...           for p in system[:].position]
>>> curves = [visual.curve(color = visual.color.green, radius=3.)
...          for i in range(len(system))]
>>> time  = 365*24*h
>>> steps = int(time/delta_t)
>>> for k in range(steps):
...        print F.f90wrap.center_of_mass(n)
```

```
...         F.f90wrap.move(F.f90wrap.delta_t, n)
...         for i in range(len(system)):
...            spheres[i].pos = system[i].position
...            curves[i].append(pos = system[i].position)
>>> print F.center_of_mass(system[:].mass, system[:].position, n)
[-0.00088762  0.00157605  0.00066594]
```

6 Test de performance

Comparons les performances des fonctions numpy avec les fonctions f2py à l'aide du programme suivant :

```
import f90mod as F
import time
t0 = time.time()
print center_of_mass(masses, positions)
move(masses, positions, velocities, 365*24*h)
print center_of_mass(masses, positions)
t1 = time.time()
print F.center_of_mass(masses, positions)
F.f90wrap.move(masses, positions, velocities, 365*24*h)
print F.center_of_mass(masses, positions)
print 'Temps pour la version numpy = ', t1-t0
print 'Temps pour la version f2py = ', time.time() -t1
```

7 Modification d'un fichier signature : dgemm

Nous allons créer la fonction f2py de DGEMM : la subroutine lapack permettant de calculer le produit de deux matrices. Générer votre fichier signature `dgemm.pyf` :

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

Modifier ce fichier pour que cette fonction soit utilisable dans Python avec les lignes suivantes.

`f2py -c dgemm.pyf`

```
>>> 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(a,b)
>>> print c
[[ 39.  54.  69.]
 [ 19.  26.  33.]
 [  9.  12.  15.]]
```

Correction :

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

8 Ajout de la fonction F2PY dans la classe Grid du module Heat

Nous allons créer maintenant la subroutine Fortran qui va permettre de calculer le laplacien.

```
subroutine laplacien(u,  xsize, ysize, residual)
implicit none
!f2py real(8), dimension(xsize,ysize), intent(in), intent(out) :: u
real(8), dimension(xsize,ysize) :: u
integer, intent(in) :: xsize, ysize
real(8), intent(out) :: residual
integer :: i, j
real(8) :: tmp, up
residual=0
do j = 2, ysize-1
do i = 2, xsize-1
   up = u(i,j)
   u(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))
   if ((u(i,j)>0) ) then
      tmp = abs((up - u(i,j)) / u(i,j))
      if (residual<tmp) residual=tmp
   end if
end do
end do
end subroutine laplacien
```

Compilation du module f2py

```
f2py -c laplacien.f90 -m fmod
```

On ajoute la fonction dans la classe Grid

```
import fmod

class Grid:
    ...
    def laplacien_fortran(self):
        import fmod
        self.residual = 0
        a = self._data
        self._data, self._residual = fmod.laplacien(a)
        return self._residual
...
```

On remarque que les dimensions du champ sont optionnelles. Si on modifie l'appel du laplacien dans la méthode `nextStep` on doit obtenir les mêmes résultats.

```
 def nextStep(self, step=sys.maxint):
        if (self._step == 0):
            self.initialize()
            self._last = step
            while ((self._step <= self._last) and (self._residual > self.eps)):
                print 'Step # ', self._step
                self.laplacien_fortran()
                self._step += 1
        return self
```