Atelier COMPIL : Utilisation de IRPF90

Écriture d'un code de dynamique moléculaire

Anthony Scemama

—

Laboratoire de Chimie et Physique Quantiques
CNRS-IRSAMC, Université de Toulouse, France
scemama@irsamc.ups-tlse.fr
http://irpf90.ups-tlse.fr

—

31 Mai 2011

1 Introduction

La dynamique moléculaire permet d'observer le mouvement d'atomes au cours temps en fonction de leurs positions et vitesses initiales. Dans cet atelier, nous allons écrire un programme de dynamique moléculaire pour illustrer l'utilisation de l'outil IRPF90. Ce programme va lire les paramètres du champ de forces et les positions initiales des atomes, puis va imprimer dans un fichier les coordonnées des atomes après chaque petit déplacement des atomes en fonction de leur vitesse. Ce fichier pourra ensuite être visualisé avec des outils adaptés (Molden par exemple).

Nous allons devoir programmer :

- L'énergie potentielle d'une paire d'atomes (potentiel de Lennard-Jones). Cela nous familiarisera avec l'environnement IRPF90.
- L'énergie potentielle et cinétique d'un système de N atomes. Ici, nous créerons des tableaux dont les dimensions sont des entités IRP.
- L'accélération des particules par différences finies. Nous verrons que tout ceci est extrêmement simple avec le mot-clé `TOUCH`.
- L'algorithme de Verlet pour faire bouger le tout.

Et en bonus :

- un programme fortran qui donne la liste des entités IRP et leur documentation. Nous verrons comment inclure des scripts pour générer du code.

Avant toute chose, vous devez télécharger irpf90 sur `http://irpf90.ups-tlse.fr`. Vous devez aussi disposer de

- Python 2.4 ou supérieur

- Un compilateur Fortran
- make

Pour le programme qui suit, on utilisera les paramètres de l'atome d'Argon:

- masse : 39.948 g/mol
- $\epsilon = 0.0661$ j/mol
- $\sigma = 0.3345$ nm

Les coordonnées des atomes seront exprimées en nm.

2 Écriture du programme

2.1 Préparation de l'environnement

Créez un nouveau répertoire de travail. Allez dans ce répertoire et tapez la commande

```
$ irpf90 --init
```

Deux répertoires temporaires sont créés:

```
$ ls
IRPF90_man   IRPF90_temp   Makefile
```

et un Makefile standard est produit, utilisant par défaut `gfortran`.

```
$ cat Makefile
IRPF90 = irpf90  #-a -d
FC     = gfortran
FCFLAGS= -ffree-line-length-none -O2

SRC=
OBJ=
LIB=

include irpf90.make

irpf90.make: $(wildcard *.irp.f)
	$(IRPF90)
```

Dans le makefile, activez les assertions grace à l'option `-a` de `irpf90`. Activez aussi le debug pour suivre le parcours de l'arbre dans la sortie.

```
IRPF90 = irpf90 -a -d
```

2.2 Potentiel de Lennard-Jones

Exercice

Ecrire un programme qui calcule le potentiel de Lennard-Jones:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{1}$$

en demandant à l'utilisateur les valeurs de σ, r et ϵ. Créez le programme principal dans un fichier nommé `test.irp.f` et les providers nécessaires dans un fichier nommé `potential.irp.f`. Vous n'avez pas besoin de modifier le `Makefile`.

Pour compiler le programme, tapez juste

```
$ make
Makefile:9: irpf90.make: No such file or directory
irpf90
IRPF90_temp/potential.irp.module.F90
IRPF90_temp/potential.irp.F90
IRPF90_temp/test.irp.module.F90
IRPF90_temp/test.irp.F90
gfortran -O2 -c IRPF90_temp/test.irp.module.F90 -o IRPF90_temp/test.irp.module.o
gfortran -O2 -c IRPF90_temp/potential.irp.module.F90 -o IRPF90_temp/potential.ir
p.module.o
gfortran -O2 -c IRPF90_temp/test.irp.F90 -o IRPF90_temp/test.irp.o
gfortran -O2 -c IRPF90_temp/irp_stack.irp.F90 -o IRPF90_temp/irp_stack.irp.o
gfortran -O2 -c IRPF90_temp/potential.irp.F90 -o IRPF90_temp/potential.irp.o
gfortran -O2 -c IRPF90_temp/irp_touches.irp.F90 -o IRPF90_temp/irp_touches.irp.o
gfortran -o test IRPF90_temp/test.irp.o IRPF90_temp/test.irp.module.o IRPF90_te
mp/irp_stack.irp.o IRPF90_temp/potential.irp.o IRPF90_temp/potential.irp.modul
e.o IRPF90_temp/irp_touches.irp.o
```

Et un binaire exécutable nommé `test` est créé

```
$ ls
irpf90_entities  IRPF90_man/   Makefile          test*
irpf90.make      IRPF90_temp/  potential.irp.f   test.irp.f
```

À la compilation, ignorez la ligne :

```
Makefile:9: irpf90.make: No such file or directory
```

C'est un warning, pas une erreur. Le fichier `irpf90.make` n'existe effectivement pas. Il va donc être créé automatiquement en appelant `irpf90`.

Solution

```
1 ! Fichier test.irp.f
2 program test
3   print *, V_lj
4 end program
5
```

```
! Fichier potential.irp.f
BEGIN_PROVIDER [ double precision, V_lj ]
  implicit none
  BEGIN_DOC
! Lennard Jones potential energy.
  END_DOC
  double precision :: sigma_over_r
  sigma_over_r = sigma_lj / interatomic_distance
  V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 - sigma_over_r**6 )
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj ]
  implicit none
  BEGIN_DOC
! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)
  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER

BEGIN_PROVIDER [ double precision, interatomic_distance ]
  implicit none
  BEGIN_DOC
! Distance between the atoms
  END_DOC
  print *, 'Inter-atomic_distance?'
  read (*,*) interatomic_distance
  ASSERT (interatomic_distance >= 0.)
END_PROVIDER
```

Sortie

```
$ ./test
           0 : -> provide_v_lj
           0 :  -> provide_epsilon_lj
           0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
0.3345
           0 :   <- epsilon_lj  1.0000000000000002E-003
           0 :  <- provide_epsilon_lj  1.0000000000000002E-003
           0 :  -> provide_interatomic_distance
           0 :   -> interatomic_distance
 Inter-atomic distance?
.3
```

```
          0 :   <- interatomic_distance    0.0000000000000000
          0 :  <- provide_interatomic_distance    0.0000000000000000
          0 :  -> v_lj
          0 :  <- v_lj    0.0000000000000000
          0 : <- provide_v_lj   1.0000000000000002E-003
          0 : -> test
 0.46819241808782691
          0 : <- test    0.0000000000000000
```

2.3 Description des atomes

Exercice

Dans le même répertoire, écrivez un programme qui lit dans l'entrée standard le nombre de noyaux, puis, pour chaque atome, sa masse et ses coordonnées x, y et z. Il imprimera ensuite la matrice des distances entre paires d'atomes.

Ici, il faudra créer

- un provider pour `Natoms`, le nombre d'atomes (un entier)
- un provider pour `coord` et `mass`, les coordonnées et masses des atomes. Ce sont des tableaux de réels de dimensions `(3,Natoms)` et `(Natoms)`
- un provider pour `distance`, la matrice des distances de dimension `(Natoms,Natoms)`

Voila la sortie que vous devez obtenir:

```
$ ./test2
          0 : -> provide_distance
          0 :  -> provide_natoms
          0 :   -> natoms
 Number of atoms?
3
          0 :   <- natoms   1.0000000000000002E-003
          0 :  <- provide_natoms   1.0000000000000002E-003
          0 :  -> provide_coord
          0 :   -> coord
 For each atom: x, y, z, mass?
0. 0. 0. 40.
1. 2. 3. 10.
-1. 0. 2. 20.
          0 :   <- coord   1.0000000000000002E-003
          0 :  <- provide_coord   1.0000000000000002E-003
          0 :  -> distance
          0 :  <- distance    0.0000000000000000
          0 : <- provide_distance   2.0000000000000004E-003
          0 : -> test2
   0.0000000000000000         3.7416573867739413         2.2360679774997898
   3.7416573867739413         0.0000000000000000         3.0000000000000000
   2.2360679774997898         3.0000000000000000         0.0000000000000000
          0 : <- test2    0.0000000000000000
```

Vous pouvez vérifier que vous avez bien documenté votre code en utilisant la commande `irpman`.

```
$ irpman coord
```

vous ouvrira une "man page" qui vous renseignera sur l'entité `coord`. La liste de toutes les entités se trouve dans le fichier `irpf90_entities`.

Solution

```
1  ! Fichier test2.irp.f
2  program test2
3   integer :: i
4   do i=1,Natoms
5    print *, distance(:,i)
6   enddo
7  end program
8
9  ! Fichier atoms.irp.f
10 BEGIN_PROVIDER [ integer, Natoms ]
11  implicit none
12  BEGIN_DOC
13 ! Number of atoms
14  END_DOC
15  print *, 'Number of atoms?'
16  read(*,*) Natoms
17  ASSERT (Natoms > 0)
18 END_PROVIDER
19
20  BEGIN_PROVIDER [ double precision, coord, (3,Natoms) ]
21 &BEGIN_PROVIDER [ double precision, mass , (Natoms)   ]
22  implicit none
23  BEGIN_DOC
24 ! Atomic data, input in atomic units.
25  END_DOC
26  print *, 'For each atom: x, y, z, mass?'
27  integer :: i,j  ! On peut declarer les variables n'importe ou dans le provider
28  do i=1,Natoms
29   read(*,*) (coord(j,i), j=1,3), mass(i)
30   ASSERT (mass(i) > 0.)
31  enddo
32 END_PROVIDER
33
34 BEGIN_PROVIDER [ double precision, distance, (Natoms,Natoms) ]
35   implicit none
36   BEGIN_DOC
37 ! distance  : Distance matrix of the atoms
38   END_DOC
39   integer :: i,j,k
40   do i=1,Natoms
41    do j=1,Natoms
42     distance(j,i) = 0.
43     do k=1,3
```

```
      distance(j,i) += (coord(k,i)-coord(k,j))**2   !Operateur d'incrementation +=
     enddo
     distance(j,i) = sqrt(distance(j,i))
    enddo
  enddo
END_PROVIDER
```

2.4 Potentiel pour plusieurs particules

Exercice

Changez le provider de V_{LJ} du premier programme. Maintenant, au lieu de calculer le potentiel de Lennard-Jones pour une distance r, vous devrez claculer l'énergie potentielle totale qui est la somme des énergies potentielles par paires d'atomes:

$$V_{\mathrm{LJ}} = \sum_{i=1}^{\texttt{Natoms}} \sum_{i=j+1}^{\texttt{Natoms}} V(r_{ij}) \tag{2}$$

Les dépendances ont changé, et IRPF90 en tient compte automatiquement. Relancer le programme test pour le vérifier.

Solution

```
! Fichier potential.irp.f
BEGIN_PROVIDER [ double precision, V ]
  implicit none
  BEGIN_DOC
! Potential energy.
  END_DOC
  V = V_lj
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_lj ]
  implicit none
  BEGIN_DOC
! Lennard Jones potential energy.
  END_DOC
  integer :: i,j
  double precision :: sigma_over_r
  V_lj = 0.
  do i=1,Natoms
   do j=i+1,Natoms
     ASSERT (distance(j,i) > 0.)    ! On veut eviter la division par zero
     sigma_over_r = sigma_lj / distance(j,i)
     V_lj += sigma_over_r**12 - sigma_over_r**6
   enddo
  enddo
  V_lj *= 4.d0 * epsilon_lj   ! Operateur *=
END_PROVIDER
```

Sortie

```
$ ./test
            0 : -> provide_v_lj
            0 :  -> provide_epsilon_lj
            0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
            0 :   <- epsilon_lj    0.0000000000000000
            0 :  <- provide_epsilon_lj    0.0000000000000000
            0 :  -> provide_natoms
            0 :   -> natoms
 Number of atoms?
3
            0 :   <- natoms   1.0000000000000002E-003
            0 :  <- provide_natoms   1.0000000000000002E-003
            0 :  -> provide_distance
            0 :   -> provide_coord
            0 :    -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
            0 :    <- coord    0.0000000000000000
            0 :   <- provide_coord    0.0000000000000000
            0 :   -> distance
            0 :   <- distance    0.0000000000000000
            0 :  <- provide_distance    0.0000000000000000
            0 :  -> v_lj
            0 :  <- v_lj    0.0000000000000000
            0 : <- provide_v_lj   1.0000000000000002E-003
            0 : -> test
  0.39685690695535714
            0 : <- test    0.0000000000000000
```

2.5 Énergie cinétique des atomes

Exercice

Ecrivez un programme qui écrit l'énergie totale $E_{\text{tot}} = T + V$, où T est l'énergie cinétique du système. Écrivez ici le provider de l'énergie cinétique, et le provider de l'énergie totale E_{tot}. Toutes les vitesses seront choisies égales à zéro par défaut dans le provider des vitesses. Souvenez-vous que vous avez déjà le provider des masses atomiques.

$$T = \frac{1}{2} \sum_{i=1}^{\texttt{Natoms}} m_i v_i^2 \tag{3}$$

Solution

```
! Fichier test3.irp.f
program test3
  print *, E_tot
end program

! Fichier energy.irp.f
BEGIN_PROVIDER [ double precision, E_tot ]
 implicit none
 BEGIN_DOC
! Total energy of the system
 END_DOC
 E_tot = T + V
END_PROVIDER

! Fichier velocity.irp.f
BEGIN_PROVIDER [ double precision, T ]
 implicit none
 BEGIN_DOC
! Kinetic energy per atom
 END_DOC
 T = 0.d0
 integer :: i
 do i=1,Natoms
  T += mass(i) * velocity2(i)
 enddo
 T *= 0.5d0
END_PROVIDER

BEGIN_PROVIDER [ double precision, velocity2, (Natoms) ]
 implicit none
 BEGIN_DOC
! Square of the norm of the velocity per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
  velocity2(i) = 0.d0
  do k=1,3
   velocity2(i) += velocity(k,i)*velocity(k,i)
  enddo
 enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, velocity, (3,Natoms) ]
 implicit none
 BEGIN_DOC
! Velocity vector per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
```

```
  do k=1,3
   velocity(k,i) = 0.d0
  enddo
 enddo
END_PROVIDER
```

Sortie

```
$ ./test3
            0 : -> provide_e_tot
            0 :  -> provide_t
            0 :   -> provide_velocity2
            0 :    -> provide_velocity
            0 :     -> provide_natoms
            0 :      -> natoms
 Number of atoms?
3
            0 :      <- natoms    0.0000000000000000
            0 :     <- provide_natoms    0.0000000000000000
            0 :     -> velocity
            0 :     <- velocity    0.0000000000000000
            0 :    <- provide_velocity    0.0000000000000000
            0 :    -> velocity2
            0 :    <- velocity2    0.0000000000000000
            0 :   <- provide_velocity2    0.0000000000000000
            0 :   -> provide_coord
            0 :    -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
            0 :    <- coord   9.98999999999999888E-004
            0 :   <- provide_coord   9.98999999999999888E-004
            0 :   -> t
            0 :   <- t    0.0000000000000000
            0 :  <- provide_t   9.98999999999999888E-004
            0 :  -> provide_v
            0 :   -> provide_v_lj
            0 :    -> provide_epsilon_lj
            0 :     -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
            0 :     <- epsilon_lj   1.00000000000000002E-003
            0 :    <- provide_epsilon_lj   1.00000000000000002E-003
            0 :    -> provide_distance
            0 :     -> distance
            0 :     <- distance    0.0000000000000000
            0 :    <- provide_distance    0.0000000000000000
            0 :    -> v_lj
```

```
          0 :    <- v_lj    0.0000000000000000
          0 :   <- provide_v_lj   1.0000000000000002E-003
          0 :   -> v
          0 :   <- v    0.0000000000000000
          0 :  <- provide_v   1.0000000000000002E-003
          0 :  -> e_tot
          0 :  <- e_tot    0.0000000000000000
          0 : <- provide_e_tot   1.9989999999999991E-003
          0 : -> test3
  0.39685690695535714
          0 : <- test3    0.0000000000000000
```

2.6 Calcul de l'accélération

Exercice

Le vecteur accélération **a** est donnée par:

$$a_{x_i} = -\frac{1}{m_i}\frac{\partial V}{\partial x_i} \tag{4}$$

où x_i est la coordonnée x de l'atome i (un élément du tableau `coord`).

Écrivez le provider de `V_grad_numeric`, la dérivée de V par rapport à x_i en différences finies:

$$\frac{\partial V}{\partial x_i} \sim \frac{V(x_i + \Delta x_i) - V(x_i - \Delta x_i)}{2\Delta x_i} \tag{5}$$

Il sera nécessaire d'utiliser le mot-clé `TOUCH`. Le calcul de l'accélération ne doit pas dépendre de la méthode de calcul du gradient du potentiel. Nous utiliserons l'entité `V_grad` dans le provider de l'accélération, qui est une simple copie des valeurs de `V_grad_numeric` dans `V_grad`.

Solution

```
! Fichier test4.irp.f
program test4
 implicit none
 integer :: i
 do i=1,Natoms
  print *, acceleration(:,i)
 enddo
end program

! Fichier potential.irp.f
BEGIN_PROVIDER [ double precision, dstep ]
 implicit none
 BEGIN_DOC
! Finite difference step
 END_DOC
 dstep = 1.d-4
END_PROVIDER
```

```
BEGIN_PROVIDER [ double precision, V_grad_numeric, (3,Natoms) ]
 implicit none
 BEGIN_DOC
! Numerical gradient of the potential
 END_DOC
 integer :: i, k
 do i=1,Natoms
  do k=1,3
   coord(k,i) += dstep    ! On deplace la coordonnee x_i en x_i + delta
   TOUCH coord            ! On informe IRPF90 que coord a ete modifie
   V_grad_numeric(k,i) = V       ! V est ici V(x_i + delta)
   coord(k,i) -= 2.d0*dstep      ! On deplace la coordonnee x_i en x_i - delta
   TOUCH coord            ! On informe IRPF90 que coord a ete modifie
   V_grad_numeric(k,i) -= V ! V est ici V(x_i - delta)
   V_grad_numeric(k,i) *= .5d0/dstep
   coord(k,i) += dstep    ! On remet x_i a sa position d'origine
                          ! Il n'est pas necessaire de re-toucher coord puisque:
                          ! - au prochain tour de boucle, il est re-touche
                          ! - a la sortie de la boucle, il est 'soft-touche'
  enddo
 enddo
 SOFT_TOUCH coord    ! Ne re-provide pas les entites courantes. Ici, V
                     ! ne serait pas re-calcule. Economise du temps de calcul,
                     ! mais ne peut etre utilise que lorsque plus rien n'est
                     ! utilise apres.
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_grad, (3,Natoms) ]
 implicit none
 BEGIN_DOC
! Gradient of the potential
 END_DOC
 integer :: i,k
 do i=1,Natoms
  do k=1,3
   V_grad(k,i) = V_grad_numeric(k,i)
  enddo
 enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, acceleration, (3,Natoms) ]
 implicit none
 BEGIN_DOC
! Acceleration = - grad(V)/m
 END_DOC
 integer :: i, k
 do i=1,Natoms
  do k=1,3
   acceleration(k,i) = -V_grad(k,i)/mass(i)
  enddo
 enddo
```

```
END_PROVIDER
```

Sortie

```
$ ./test4
           0 : -> provide_acceleration
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- natoms    0.0000000000000000
           0 :  <- provide_natoms    0.0000000000000000
           0 :  -> provide_coord
           0 :   -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
           0 :   <- coord    0.0000000000000000
           0 :  <- provide_coord    0.0000000000000000
           0 :  -> provide_v_grad
           0 :   -> provide_v_grad_numeric
           0 :    -> provide_v
           0 :     -> provide_v_lj
           0 :      -> provide_epsilon_lj
           0 :       -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
           0 :       <- epsilon_lj   1.0000000000000002E-003
           0 :      <- provide_epsilon_lj   1.0000000000000002E-003
           0 :      -> provide_distance
           0 :       -> distance
           0 :       <- distance    0.0000000000000000
           0 :      <- provide_distance    0.0000000000000000
           0 :      -> v_lj
           0 :      <- v_lj    0.0000000000000000
           0 :     <- provide_v_lj   1.0000000000000002E-003
           0 :     -> v
           0 :     <- v    0.0000000000000000
           0 :    <- provide_v   1.0000000000000002E-003
           0 :    -> provide_dstep
           0 :     -> dstep
           0 :     <- dstep    0.0000000000000000
           0 :    <- provide_dstep    0.0000000000000000
           0 :    -> v_grad_numeric
           0 :     -> touch_coord
           0 :     <- touch_coord    0.0000000000000000
           0 :     -> provide_v
           0 :      -> provide_v_lj
```

```
          0 :        -> provide_distance
          0 :         -> distance
          0 :         <- distance    0.0000000000000000
          0 :        <- provide_distance    0.0000000000000000
          0 :        -> v_lj
          0 :        <- v_lj    0.0000000000000000
          0 :       <- provide_v_lj   9.98999999999999888E-004
          0 :       -> v
          0 :       <- v    0.0000000000000000
          0 :      <- provide_v   9.98999999999999888E-004
.....................................................
          0 :      -> touch_coord
          0 :      <- touch_coord    0.0000000000000000
          0 :     <- v_grad_numeric   1.09980000000000008E-002
          0 :    <- provide_v_grad_numeric   1.19980000000000017E-002
          0 :    -> v_grad
          0 :    <- v_grad    0.0000000000000000
          0 :   <- provide_v_grad   1.19980000000000017E-002
          0 :   -> acceleration
          0 :   <- acceleration    0.0000000000000000
          0 : <- provide_acceleration   1.19980000000000017E-002
          0 : -> test4
 -1.21434697003541814E-003  -2.42873782738128874E-003   -2.8852483886702140
  3.77225707531847476E-004   7.54451431647651383E-004    1.4421824477391931
  3.06597036647815457E-004   6.13223309390657279E-004   5.88995461200022218E-004
          0 : <- test4    0.0000000000000000
```

2.7 Dynamique des atomes

Exercice
L'algorithme de Verlet est le suivant:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^n \Delta t + \mathbf{a}^n \frac{\Delta t^2}{2} \quad (6)$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{1}{2}(\mathbf{a}^n + \mathbf{a}^{n+1})\Delta t \quad (7)$$

où n est l'indice du pas courant, $\mathbf{r}$ est le vecteur des positions, $\mathbf{v}$ est le vecteur des vitesses, $\mathbf{a}$ est le vecteur des accélérations, et Δt est un pas de temps.

Écrire une subroutine qui implémente l'algorithme de Verlet: À une itération n

- Calculer les coordonnées au pas $n+1$
- Calculer la composante de la vitesse qui dépend de la position au pas n
- Faire le `TOUCH` des coordonnées et des vitesses
- Ajouter aux vitesses la partie qui dépend du pas $n+1$
- Faire le `TOUCH` des vitesses

Solution

```
program test5

! Fichier verlet.irp.f
BEGIN_PROVIDER [ integer, Nsteps ]
 implicit none
 BEGIN_DOC
! Number of steps for the dynamics
 END_DOC
 print *, 'Nsteps?'
 read(*,*) Nsteps
 ASSERT (Nsteps > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, tstep ]
&BEGIN_PROVIDER [ double precision, tstep2 ]
 implicit none
 BEGIN_DOC
! Time step for the dynamics
 END_DOC
 print *, 'Time step?'
 read(*,*) tstep
 ASSERT (tstep > 0.)
 tstep2 = tstep*tstep
END_PROVIDER

subroutine verlet
 implicit none
 integer :: is, i, k
 do is=1,Nsteps
! call print_data(is)      ! A de-commenter pour l'exercice suivant
  do i=1,Natoms
   do k=1,3
    coord(k,i) += tstep*velocity(k,i) + 0.5*tstep2*acceleration(k,i)
    velocity(k,i) += 0.5*tstep*acceleration(k,i)
   enddo
  enddo
  TOUCH coord velocity
  do i=1,Natoms
   do k=1,3
    velocity(k,i) += 0.5*tstep*acceleration(k,i)
   enddo
  enddo
  TOUCH velocity
 enddo
end subroutine
```

Sortie (sans debug)

```
$ ./test5
 Number of atoms?
3
 For each atom: x, y, z, mass?
0 0 0 40
0 0 .5 40
.1 .2 -.5 40
    0.0000000000000000          0.0000000000000000          0.0000000000000000
    0.0000000000000000          0.0000000000000000          0.29999999999999999
   0.10000000000000001         0.20000000000000001         -0.29999999999999999
 Epsilon?
.0661
 Sigma?
.3345
 Nsteps?
1000
 Time step?
.2
  -4.85173626539635722E-002  -9.70435730911516636E-002   0.18819318566003412
  -1.11022168342603082E-002  -2.22085308639154572E-002   0.62064345350606354
   0.15961957948937169         0.31925210395487302        -0.80883663916789905
```

2.8 Gestion des fichiers

Exercice
Ajoutez une impression dans un fichier des coordonnées des atomes après chaque déplacement. Pour cela, vous devez décommenter la ligne de l'exercice précédent et écrire la subroutine `print_data` et les providers associés au fichier.

Solution

```
! Fichier files.irp.f
integer function getUnitAndOpen(f,mode)
! Trouve un numero d'unite et ouvre le fichier
  implicit none
  character*(*)       :: f
  character*(128)     :: new_f
  integer             :: iunit
  logical             :: is_open, exists
  character           :: mode

  is_open = .True.
  iunit = 10
  new_f = f
  do while (is_open)
    inquire(unit=iunit,opened=is_open)
    if (.not.is_open) then
      getUnitAndOpen = iunit
    endif
    iunit = iunit+1
```

```
  enddo
  if (mode.eq.'r') then
    inquire(file=f,exist=exists)
    if (.not.exists) then
      open(unit=getUnitAndOpen,file=f,status='NEW',action='WRITE')
      close(unit=getUnitAndOpen)
    endif
    open(unit=getUnitAndOpen,file=f,status='OLD',action='READ')
  else if (mode.eq.'w') then
    open(unit=getUnitAndOpen,file=new_f,status='UNKNOWN',action='WRITE')
  else if (mode.eq.'a') then
    open(unit=getUnitAndOpen,file=new_f,status='UNKNOWN', &
    action='WRITE',position='APPEND')
  else if (mode.eq.'x') then
    open(unit=getUnitAndOpen,file=new_f)
  endif
end function getUnitAndOpen

BEGIN_PROVIDER [ integer, output ]
   BEGIN_DOC
!   File unit corresponding to the output file.
   END_DOC
   integer :: getUnitAndOpen
   output = getUnitAndOpen('output','w')
END_PROVIDER

subroutine print_data(is)
 implicit none
 integer, intent(in) :: is
 write(output,*) Natoms
 write(output,'(I8, 3(2X, E15.8))') is, V, T, E_tot
 integer :: i
 do i=1,Natoms
  write(output,'(A,3(2X,F15.8))') 'Ar', coord(:,i)
 enddo
end subroutine print_data
```

3 Bonus

3.1 Documentation du programme

```
program get_doc

 integer :: iargc
 character*(32) :: arg
 integer :: i, j

 !-----------
 ! Commande : ./get_doc
 ! Imprime la liste des entites IRP
```

```
 !-----------
 if (iargc() == 0) then
  print *, 'Liste des entites IRP'
  do j=1,size(entities)
   print *, entities(j)
  enddo
  return
 endif

 !-----------
 ! Commande : ./get_doc  titi  toto  momo
 ! Imprime la documentation des entites IRP titi, toto et momo
 !-----------
 do i=1,iargc()
   call getarg(i,arg)

!-----------
! Script Python, execute a la compilation, qui trouve le nom de toutes les
! entites IRP. Si la variable est dans la ligne de commande, on imprime sa
! documentation. Le shell est dans le programme principal.
!-----------

BEGIN_SHELL [ /usr/bin/python ]
import os
entities = []
for filename in os.listdir('.'):    # On boucle sur tous les noms de fichiers
  if filename.endswith('.irp.f'):   #  Si le nom du fichier finit par .irp.f
    file = open(filename,'r')       #   On l'ouvre
    for line in file:               #   Pour chacune de ses lignes
      if line.strip().lower().startswith('begin_provider'):
                                    #    Si la ligne commence par
                                    #     begin_provider (sans casse)
        name = line.split(',')[1].split(']')[0].strip()
                                    #    On decoupe la ligne pour en extraire
                                    #     le nom de l'entite
        entities.append(name)       #    Et on la met dans la liste 'entities'
    file.close()                    #   On ferme le fichier

for e in entities:
  print "  if (arg == '%s') then"%(e,)
  print "    print *, %s_doc"%(e,)
  print "  endif"
END_SHELL

 enddo
end

!-----------------
! Script qui cree les provider necessaires. Ce shell n'est pas dans
! une subroutine ou un provider.
!-----------------
```

```
BEGIN_SHELL [ /usr/bin/python ]

import os
doc = {}
for filename in os.listdir('.'):    # On boucle sur tous les noms de fichiers
  if filename.endswith('.irp.f'):   #  Si le nom du fichier finit par .irp.f
    file = open(filename,'r')       #   On l'ouvre
    inside_doc = False              #   On n'est pas encore dans la doc
    for line in file:               #   Pour chacune des lignes
      if line.strip().lower().startswith('begin_provider'):
                                    #    Si la ligne commence par
                                    #     begin_provider (sans casse)
        name = line.split(',')[1].split(']')[0].strip()
                                    #     On decoupe la ligne pour en extraire
                                    #      le nom de l'entite
        doc[name] = ""              #     La doc est initialisee vide
      elif line.strip().lower().startswith('begin_doc'):
                                    #    Si la ligne commence par
                                    #     begin_doc (sans casse)
        inside_doc = True           #     on est dans la doc
      elif line.strip().lower().startswith('end_doc'):
                                    #    Si la ligne commence par
                                    #     end_doc (sans casse)
        inside_doc = False          #     on n'est plus dans la doc
      elif inside_doc:              #    Sinon si on est dans la doc
        doc[name] += line[1:].strip()+" "
                                    #     on ajoute a la doc de l'entite
                                    #      la ligne courante.
    file.close()                    #   On ferme le fichier

lenmax = 0
for e in doc.keys():
  lenmax = max(len(e),lenmax)

# On cree le provider de entities, la liste de toutes les entites
# ---------------------------------------------------------------
print "BEGIN_PROVIDER [ character*(%d), entities, (%d) ]"%(lenmax,len(doc))
print " BEGIN_DOC"
print "! List of IRP entities"
print " END_DOC"
for i,e in enumerate(doc.keys()):
  print "entities(%d) = '%s'"%(i+1, e)
print "END_PROVIDER"

# On cree les providers de chacune des entites
# --------------------------------------------
for e in doc.keys():
  print "BEGIN_PROVIDER [ character*(%d), %s_doc ]"%(len(doc[e]),e)
  print " BEGIN_DOC"
  print "! Documentation of variable %s"%(e,)
  print " END_DOC"
```

```
    print " %s_doc = '%s '"%(e,doc[e])
    print "END_PROVIDER"

END_SHELL
```