Table of Contents

1. Introduction
2. Prepare the working environment
3. The Lennard-Jones potential
4. Describing the atoms
5. Potential for multiple particles
6. Computing the total energy
7. Computing the acceleration
8. Implementing the molecular dynamics
9. Handling files

IRPF90 Tutorial : A molecular dynamics code

Molecular dynamics models the movement of atoms according to their initial positions and velocities. In this tutorial, we will write a molecular dynamics program to illustrate how to use IRPF90. This program will read the force field parameters from an input file, as well as the initial positions of the atoms. After each little displacement of the atoms according to their velocities, the new set of coordinates will be printed into an output file such that a video animation can easily be produced with an external tool.

Here is the list of what we will have to code:

- The potential energy of a couple of atoms (Lennard-Jones potential). This will will be a very simple introduction to IRPF90.
- The potential and kinetic energy of system of *N* atoms. We will have to create arrays dimensioned by other IRP entities.
- The acceleration of the particles using finite differences for the calculation of derivatives. This part will introduce the `TOUCH` keyword.
- The Verlet algorithm to make everything move.

The first thing you will have to do is download IRPF90 from the web site: http://irpf90.ups-tlse.fr

Physical Parameters

For all this tutorial, we will use Argon atoms with the following parameters:

- mass : 39.948 g/mol
- epsilon : 0.0661 j/mol
- sigma : 0.3345 nm

The atom coordinates are given in nanometers.

Prepare the working environment

Create a new directory for the project. Inside this directory, initialize the IRPF90 environment using:

```
$ irpf90 --init
```

Two directories were created

```
$ ls
IRPF90_man  IRPF90_temp  Makefile
```

and a Makefile containing default parameters for the gfortran compiler

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

SRC=
OBJ=
LIB=

include irpf90.make

irpf90.make: $(filter-out IRPF90_temp/%, $(wildcard */*.irp.f)) $(wildcard *.irp.f) $(wil
        $(IRPF90)
```

In the Makefile, activate the asserts and the debug options by uncommenting `-a` and `-d` in the definition of the `IRPF90` variable.

The Lennard-Jones potential

Exercise

Write a program which computes the Lennard-Jones potential :

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

The user will be asked for the values of the Lennard-Jones parameters `sigma_lj` and `epsilon_lj`, as well as the `interatomic_distance`.

Create the main program in a file named `test.irp.f`, and the providers in a file named `potential.irp.f`. You don't need to modify the `Makefile`.

To compile the program, run

```
$ make
Makefile:9: irpf90.make: No such file or directory
irpf90  -a -d
IRPF90_temp/potential.irp.module.F90
IRPF90_temp/potential.irp.F90
IRPF90_temp/test.irp.module.F90
IRPF90_temp/test.irp.F90
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/test.irp.module.F90
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/potential.irp.module
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/test.irp.F90 -o IRPF
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/irp_stack.irp.F90 -o
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/potential.irp.F90 -o
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_temp/irp_touches.irp.F90
gfortran -I IRPF90_temp/  -o test IRPF90_temp/test.irp.o IRPF90_temp/test.irp.module.o IR
```

The warning `Makefile:9: irpf90.make: No such file or directory` can be ignored: the missing `irpf90.make` will be created by applying the rule in the `Makefile` that calls IRPF90.

A binary file named `test` will be created.

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

Expected Output

```
$ ./test
           0 : -> provide_v_lj
           0 :  -> provide_epsilon_lj
           0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
```

```
0.3345
           0 :   <- epsilon_lj  3.63000000000000041E-004
           0 :  <- provide_epsilon_lj  5.41999999999999947E-004
           0 :  -> provide_interatomic_distance
           0 :   -> interatomic_distance
 Inter-atomic distance?
0.3
           0 :   <- interatomic_distance  1.70499999999999992E-003
           0 :  <- provide_interatomic_distance  1.71499999999999994E-003
           0 :  -> v_lj
           0 :  <- v_lj   0.0000000000000000
           0 : <- provide_v_lj  2.34399999999999973E-003
           0 : -> test
  0.46819241808782769
           0 : <- test   0.0000000000000000
```

Solution

File `test.irp.f`

```
program test
  implicit none
  BEGIN_DOC
  ! Test program
  END_DOC
  print *,  V_lj
end
```

File `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
```

Describing the atoms

Exercise

In the same directory, create a program which reads in the standard input:

- The number of atoms
- For each atom: the mass and the *x*, *y*, *z* coordinates

The program will the print the matrix of the distances between atom pairs.

You will have to create :

- A provider for `Natoms`, the number of atoms
- A provider for `coord` and `mass`, the atom coordinates and mass. These are arrays with dimensions `(3,Natoms)` and `(Natoms)` respectively.
- A provider for `distance`, the distance matrix. Its dimension is `(Natoms,Natoms)`.

You can check that your code is well documented using the `irpman` command:

```
$ irpman coord
IRPF90 entities(l)                   coord                   IRPF90 entities(l)

Declaration
       double precision, allocatable :: coord  (3,Natoms)
       double precision, allocatable :: mass   (Natoms)

Description
       Atomic data, input in atomic units.

File
       atoms.irp.f

Needs
       natoms

Needed by
       distance

Instability factor
        50.0 %

IRPF90 entities                      coord                   IRPF90 entities(l)
```

Expected output

```
$ ./test2
           0 : -> provide_distance
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- natoms  1.59999999999999986E-004
```

```
           0 :  <- provide_natoms  3.38000000000000193E-004
           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  2.03999999999999754E-004
           0 :  <- provide_coord  3.09999999999999946E-004
           0 :  -> distance
           0 :  <- distance  1.99999999999983177E-006
           0 : <- provide_distance  7.83999999999999975E-004
           0 : -> test2
   0.0000000000000000        3.7416573867739413        2.2360679774997898
   3.7416573867739413        0.0000000000000000        3.0000000000000000
   2.2360679774997898        3.0000000000000000        0.0000000000000000
           0 : <- test2  5.90000000000000246E-005
```

Solution

File `test2.irp.f`

```
program test2
  implicit none
  BEGIN_DOC
  ! Second test : distance matrix
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *, distance(1:3,i)
  enddo
end program
```

File `atoms.irp.f`

```
BEGIN_PROVIDER [ integer, Natoms ]
  implicit none
  BEGIN_DOC
  ! Number of atoms
  END_DOC
  print *, 'Number of atoms?'
  read(*,*) Natoms
  ASSERT (Natoms > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, coord, (3,Natoms) ]
&BEGIN_PROVIDER [ double precision, mass , (Natoms)   ]
  implicit none
  BEGIN_DOC
  ! Atomic data, input in atomic units.
  END_DOC
  print *, 'For each atom: x, y, z, mass?'
  integer                        :: i,j  ! <-- Variables can be declared
                                         !     anywhere
  do i=1,Natoms
    read(*,*) (coord(j,i), j=1,3), mass(i)
    ASSERT (mass(i) > 0.d0)
```

```
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, distance, (Natoms,Natoms) ]
  implicit none
  BEGIN_DOC
  ! distance : Distance matrix
  END_DOC
  integer                        :: i,j,k
  do i=1,Natoms
    do j=1,Natoms
      distance(j,i) = 0.d0
      do k=1,3
        distance(j,i) += (coord(k,i)-coord(k,j))**2  ! <-- Note the increment
                                                     !     operator +=
      enddo
      distance(j,i) = dsqrt(distance(j,i))
    enddo
  enddo
END_PROVIDER
```

Potential for multiple particles

Exercise

Change the provider of `V_lj` of the first program. Now, instead of computing the Lennard-Jones potential of a single inter-atomic distance *r*, you will compute the total potential energy which is the sum of the potential energies due to each pair of atoms:

$$V_{\mathrm{LJ}} = \sum_{i=1}^{\mathtt{Natoms}} \sum_{j>i}^{\mathtt{Natoms}} V(r_{ij})$$

The dependencies have changed now, as your new version of `V_lj` needs the previously defined distance matrix `distance`. You can now run again the first program.

Expected output

```
$ ./test
           0 : -> provide_v_lj
           0 :  -> provide_epsilon_lj
           0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
0.3345
           0 :   <- epsilon_lj  3.06000000000000022E-003
           0 :  <- provide_epsilon_lj  3.07300000000000021E-003
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- natoms   0.0000000000000000
           0 :  <- provide_natoms   0.0000000000000000
           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  3.08900000000000017E-003
           0 : -> test
  0.39685690695535791
           0 : <- test   0.0000000000000000
```

Solution

File `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.d0
  do i=1,Natoms
    do j=i+1,Natoms
      ASSERT (distance(j,i) > 0.d0)   ! <-- Avoid a possible division by 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  ! <-- Note the *= operator
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
```

Computing the total energy

Exercise

Write a program which prints the total energy of the system.

$$E_{\mathrm{tot}} = T + V$$

`V` is the potential (Lennard-Jones here) and `T` is the kinetic energy

$$T = \frac{1}{2} \sum_{i=1}^{\mathtt{Natoms}} m_i v_i^2$$

Write the providers for the kinetic energy and for the total energy. All the velocities will be chosen to be initialized equal to zero in the `velocities` provider. Remember you already have the provider for the masses of the atoms.

Expected output

```
$ ./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  1.58999999999999853E-004
           0 :     <- provide_natoms  3.39000000000000325E-004
           0 :     -> velocity
           0 :     <- velocity  5.99999999999992900E-006
           0 :    <- provide_velocity  4.89000000000000285E-004
           0 :    -> velocity2
           0 :    <- velocity2  5.00000000000022995E-006
           0 :   <- provide_velocity2  6.57000000000000032E-004
           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  1.97999999999999825E-004
           0 :   <- provide_coord  2.89999999999999893E-004
           0 :   -> t
           0 :   <- t  9.99999999999699046E-007
           0 :  <- provide_t  1.04999999999999972E-003
           0 :  -> provide_v
           0 :   -> provide_v_lj
           0 :    -> provide_epsilon_lj
           0 :     -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
           0 :     <- epsilon_lj  2.07999999999999852E-004
```

```
            0 :    <- provide_epsilon_lj  3.02000000000000185E-004
            0 :    -> provide_distance
            0 :     -> distance
            0 :     <- distance  2.00000000000026545E-006
            0 :    <- provide_distance  3.69999999999997067E-005
            0 :    -> v_lj
            0 :    <- v_lj  1.00000000000013273E-006
            0 :   <- provide_v_lj  4.32999999999999791E-004
            0 :   -> v
            0 :   <- v  1.00000000000013273E-006
            0 :  <- provide_v  4.750000000000000595E-004
            0 :  -> e_tot
            0 :  <- e_tot  1.00000000000013273E-006
            0 : <- provide_e_tot  1.60399999999999996E-003
            0 : -> test3
  0.39685690695535791
            0 : <- test3  1.19999999999998580E-005
```

Solution

File `test3.irp.f`

```
program test3
  implicit none
  BEGIN_DOC
  ! Prints the total energy
  END_DOC
  print *,  E_tot
end program
```

File `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
```

File `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
```

Computing the acceleration

Exercise

The acceleration vector is given by

$$\{a_{x_i}\} = \left\{ -\frac{1}{m_i} \frac{\partial V}{\partial x_i} \right\}$$

where *x_i* is the *x* coordinate of atom *i* (an element of the `coord` array). Write the provider for `V_grad_numeric`, the finite-difference approximation of the derivative of the potential with respect to the coordinates:

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

It will be necessary to use the `TOUCH` keyword.

The computation of the acceleration should not depend directly on the method used to compute the gradient, so we will use `V_grad` in the provider for the `acceleration`. `V_grad` will be a simple copy of `V_grad_numeric`.

Expected output

```
$ ./test4
           0 : -> provide_acceleration
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- natoms  1.68999999999999879E-004
           0 :  <- provide_natoms  3.50999999999999750E-004
           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  2.26999999999999771E-004
           0 :  <- provide_coord  3.32000000000000264E-004
           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  1.60999999999999685E-004
           0 :      <- provide_epsilon_lj  2.52000000000000054E-004
           0 :      -> provide_distance
           0 :       -> distance
           0 :       <- distance  2.00000000000026545E-006
```

```
            0 :      <- provide_distance  4.30000000000000694E-005
            0 :      -> v_lj
            0 :      <- v_lj  2.00000000000026545E-006
            0 :     <- provide_v_lj  4.05999999999999677E-004
            0 :     -> v
            0 :     <- v  1.00000000000013273E-006
            0 :    <- provide_v  4.78999999999999825E-004
            0 :    -> provide_dstep
            0 :     -> dstep
            0 :     <- dstep  9.99999999999699046E-007
            0 :    <- provide_dstep  2.59999999999999815E-005
            0 :    -> v_grad_numeric
            0 :     -> touch_coord
            0 :     <- touch_coord  1.00000000000013273E-006
            0 :     -> provide_v
            0 :      -> provide_v_lj
            0 :       -> provide_distance
            0 :        -> distance
            0 :        <- distance  1.00000000000013273E-006
            0 :       <- provide_distance  2.20000000000003179E-005
            0 :       -> v_lj
            0 :       <- v_lj  9.99999999999265365E-007
            0 :      <- provide_v_lj  6.70000000000002191E-005
            0 :      -> v
            0 :      <- v  9.99999999999265365E-007
            0 :     <- provide_v  1.10999999999999988E-004
            0 :     -> touch_coord
            0 :     <- touch_coord  1.00000000000013273E-006
            0 :     -> provide_v
            0 :      -> provide_v_lj
            0 :       -> provide_distance
            0 :        -> distance
            0 :        <- distance  1.00000000000013273E-006
            0 :       <- provide_distance  2.20000000000003179E-005
            0 :       -> v_lj
            0 :       <- v_lj  1.00000000000013273E-006
            0 :      <- provide_v_lj  6.29999999999996882E-005
            0 :      -> v
            0 :      <- v  1.00000000000013273E-006
            0 :     <- provide_v  1.06000000000000191E-004
            0 :     -> touch_coord
----8<------------------------------------------------------------------------
            0 :     <- touch_coord  1.00000000000013273E-006
            0 :     -> provide_v
            0 :      -> provide_v_lj
            0 :       -> provide_distance
            0 :        -> distance
            0 :        <- distance  1.00000000000013273E-006
            0 :       <- provide_distance  2.20000000000003179E-005
            0 :       -> v_lj
            0 :       <- v_lj  1.00000000000013273E-006
            0 :      <- provide_v_lj  6.39999999999998209E-005
            0 :      -> v
            0 :      <- v  1.00000000000013273E-006
            0 :     <- provide_v  1.05000000000000059E-004
            0 :     -> touch_coord
            0 :     <- touch_coord  1.00000000000013273E-006
            0 :    <- v_grad_numeric  2.71300000000000013E-003
            0 :   <- provide_v_grad_numeric  3.29999999999999955E-003
            0 :   -> v_grad
            0 :   <- v_grad  1.00000000000013273E-006
            0 :  <- provide_v_grad  3.35400000000000021E-003
            0 :  -> acceleration
```

```
           0 :  <- acceleration  1.00000000000013273E-006
           0 : <- provide_acceleration  4.20500000000000040E-003
           0 : -> test4
 -1.21434697006317371E-003 -2.42873782740904431E-003  -2.8852483886706581
  3.77225707531847476E-004  7.54451431647651383E-004   1.4421824477394567
  3.06597036647815457E-004  6.13223309409161033E-004  5.88995461163014712E-004
           0 : <- test4  8.89999999999996697E-005
```

Solution

File `test4.irp.f`

```
program test4
  implicit none
  BEGIN_DOC
  ! Program testing the acceleration
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *, acceleration(:,i)
  enddo
end program
```

File `potential.irp.f`, add

```
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   ! Move coordinate x_i to x_i + delta
      TOUCH coord mass      ! Tell IRPF90 that coord has been changed
      V_grad_numeric(k,i) = V      ! V is here V(x_i + delta)
      coord(k,i) -= 2.d0*dstep     ! Move coordinate x_i to x_i - delta
      TOUCH coord mass      ! Tell IRPF90 that coord has been changed
      V_grad_numeric(k,i) -= V      ! V is here V(x_i - delta)
      V_grad_numeric(k,i) *= .5d0/dstep
      coord(k,i) += dstep   ! Put back x_i to its initial position
                            ! It is not necessary to re-touch coord since
                            ! - at the next loop iteration it will be touched
                            ! - out of the loop, it is soft-touched
    enddo
  enddo
  SOFT_TOUCH coord mass  ! Does not re-provide the current entities. Here, V will
                         ! not be re-computed. This reduced the CPU time, but is
                         ! dangerous.
```

```
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
```

Implementing the molecular dynamics

Exercise

The Verlet algorithm is the following

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

where *n* is the index of the current step, **r** is the position vector, **v** is the velocity vector, **a** is the acceleration vector and Δt is the time step.

Write a subroutine which implements the Verlet algorithm. To do this, at each iteration :

- Compute the coordinates at step *n*+1
- Compute the component of the velocity which depends on the position at step *n*
- `TOUCH` the coordinates and the velocities
- Add to the velocities the part which depends on step *n*+1
- `TOUCH` the velocities

For this exercise, remove the debug option in the `Makefile`.

Expected output

```
$ ./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       0.50000000000000000
  0.10000000000000001       0.20000000000000001      -0.50000000000000000
 Epsilon?
0.0661
 Sigma?
0.3345
 Nsteps?
1000
 Time step?
0.2
 -4.8517362265511752E-002 -9.7043572312663528E-002  0.18819318396255702
 -1.1102216641681017E-002 -2.2208530476353932E-002  0.62064345108566521
  0.15961957890929021       0.31925210279233324      -0.80883663504845249
```

Solution

File `test5.irp.f`

```
program test5
  implicit none
  BEGIN_DOC
  ! Program testing the verlet algorithm
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *,  coord(1:3,i)
  enddo
  call verlet
  do i=1,Natoms
    print *,  coord(1:3,i)
  enddo
end
```

File `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?'
 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 mass 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
```

8. Handling files

Exercise

Add a call to a printing subroutine at each step of the dynamics. The printing subroutine will print th coordinates of the atoms in a file. To do this, uncomment the call statement in the previous exercise and write the `print_data` subroutine as well as the associated providers.

Solution

File `files.irp.f`

```
integer function getUnitAndOpen(f,mode)
  implicit none
  BEGIN_DOC
  ! Finds an available unit number and opens the file
  END_DOC
  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 ]
  implicit none
  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
```