Molecular Dynamics Simulations

Diego Ontiveros Cruz

Molecular Modelling, December 2022

1a. Molecular Dynamics Code

In this work, the Molecular Dynamics (MD) code for simulating a system of N particles interacting through a Lennard-Jones (LJ) potential in contact with a heat bath has been developed using Fortran90. The code consists of a main program that calls the correspondent subroutines to follow the MD steps.

The scheme of the MD code is posed in Figure 1, and through this section, each part of the scheme will be explained and exemplified with the correspondent code/subroutine. The overall code, which is used for the simulation in section 1b, is available in the Appendix, or online at GitHub.

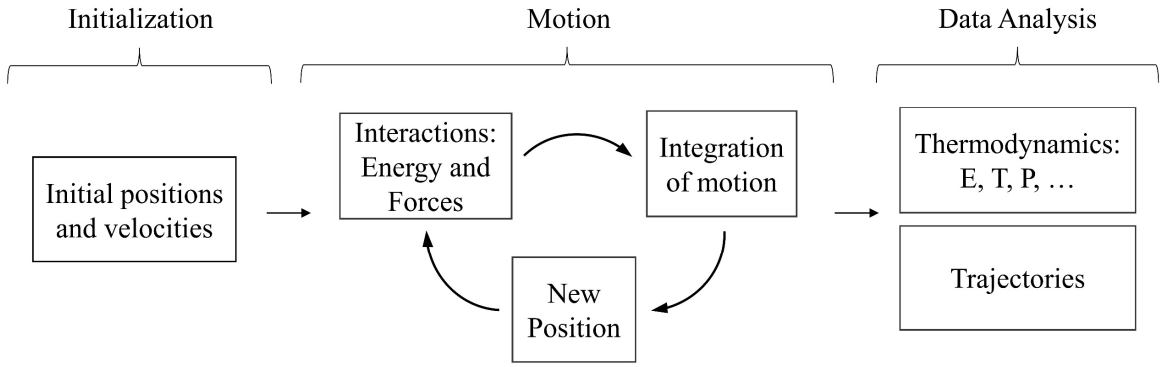


Figure 1. Simple scheme that represents the structure of a MD code.

Initialization

To start the simulations, some initial conditions must be given. This account for the positions and velocities of the particles. Depending on the problem that is studied, the initialization may be done in different ways: start from a crystalline configuration and "melt" running a NVT simulation at high temperatures, introduce random displacements in the lattice, or from random initial positions. The same is true for velocities, which can be set to zero or given a random initial value.

In our case, the code contains 3 subroutines to initialize the positions into simple cubic (*sc)*, face centered cubic (*fcc)*, or body centered cubic (*bcc)* structures, see Figure 2. All these subroutines use a unit cell matrix of the atoms positions which is repeated (looping) through the tree dimensions until the number of simulated unit cells is reached. On the other hand, the velocities can be initialized all at zero, or with a bimodal distribution of a certain temperature, see Figure 2.

The simulation box and crystalline lattice are set by the number of particles (N) and the reduced density (ρ), which are the main inputs. In that way, since the density is the number of particles in the simulation volume, $\rho = N/L^3$, the box length (L), the number of unit cells in one direction (M), and the lattice parameter (a), are automatically set with the following equations:

$$Given\ N, \rho \quad \Rightarrow \quad L = \left(\frac{N}{\rho}\right)^{\frac{1}{3}} \qquad M = \left(\frac{N}{f}\right)^{\frac{1}{3}} \qquad a = \frac{L}{M} \tag{1}$$

where f is the number of whole atoms that are in the unit cell, 1 for *sc*, 4 for *fcc* and 2 for *bcc*. Notice that, in this way, to get a complete crystal, the number of particles must be chosen a multiple of fM^3.

```
! Initial positions for SC cell
subroutine initSC(N,dens,r)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
real :: L,a
integer :: p, i,j,k, M

L = (N/dens)**(1./3.)
M = N**(1./3.)
a = L/M
p = 1
do i=0,M-1
    do j=0,M-1
        do k=0,M-1
            r(p, :) = [i,j,k]
            p=p+1
        end do
    end do
end do
r = r*a
end subroutine initSC
```

```
! Bimodal velocity distribution
subroutine initBimodal(T,v)
implicit none
real,intent(inout),dimension(:,:) :: v
real,intent(in) :: T
real :: vi
integer :: N

N = size(v, dim=1)
if (mod(N,2)/=0) then
    print*, "Number of particles (N) should be
multiple of 2."
end if
vi = sqrt(T)
v(1:N:2,:) = -vi    !All odd indices with -
v(2:N:2,:) = +vi    !All even indices with +

end subroutine initBimodal

! Half the velocities are set positive while
! the other half are set negative, to maintain
! the center of mass unmoved.
```

```
! Initial positions for FCC cell
subroutine initFCC(N,dens,r)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
real, dimension(4,3) :: ucell
real :: L,a
integer :: i,j,k, p,at,M

L = (N/dens)**(1./3.)
M = (N/4.)**(1./3.)
a = L/M
ucell = reshape((/0.,0.,0., 0.,0.5,0.5,
0.5,0.,0.5, 0.5,0.5,0./),shape(ucell),
order=(/2,1/))
do i=0,M-1
    do j=0,M-1
        do k=0,M-1
            do at=1,size(ucell, dim=1)
            r(p, :) = ucell(at,:) + (/i,j,k/)
            p=p+1
            end do
        end do
    end do
end do
r = r*a
end subroutine initFCC
```

```
! Initial positions for BCC cell
subroutine initBCC(N,dens,r)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
real, dimension(2,3) :: ucell
real :: L,a
integer :: i,j,k, p,at,M

L = (N/dens)**(1./3.)
M = (N/2.)**(1./3.)
a = L/M
ucell = reshape((/0.,0.,0., 0.5,0.5,0.5/),
shape(ucell), order=(/2,1/))
p = 1
do i=0,M-1
    do j=0,M-1
        do k=0,M-1
            do at=1,size(ucell, dim=1)
            r(p, :) = ucell(at,:) + (/i,j,k/)
            p=p+1
            end do
        end do
    end do
end do
r = r*a
end subroutine initBCC
```

Figure 2. Initialization subroutines for positions for a given a number of particles and a density and velocities, following a bimodal distribution. The initialization at v=0 subroutine has been omitted due to being only one line (v=0).

Interaction

Once the initial positions and velocities are set, the simulation begins. A certain number of steps (N_{steps}) are given and each step, or "frame", the interaction between particles is calculated in order to see how they will move the next step. The interactions consist of the Lennard-Jones potential between the particles, calculated pairwise, and the force, minus the gradient of the potential, calculated as in Equations 2 and 3, respectively. Since the energy is the sum of all pairs, it will be a scalar value, while the force is calculated component-wise, for each particle, and will be saved into a $N \times 3$ matrix.

$$V = \sum_{i=1}^{N}\sum_{j>i}^{N} 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] = \sum_{i=1}^{N}\sum_{j>i}^{N} 4\left[\left(\frac{1}{r_{ij}^{*}}\right)^{12} - \left(\frac{1}{r_{ij}^{*}}\right)^{6}\right] \tag{2}$$

$$\boldsymbol{F}(r) = -\frac{dV(r)}{dr}\hat{\boldsymbol{r}} = 4\varepsilon\left(12\frac{\sigma^{12}}{r^{13}} - 6\frac{\sigma^{6}}{r^{7}}\right)\hat{\boldsymbol{r}} = \left(\frac{48}{r^{*14}} - \frac{24}{r^{*8}}\right)\vec{\boldsymbol{r}} \tag{3}$$

The energy and the force are calculated in reduced units, which is also shown in the above equations. The reduced units will be discussed more in depth in their correspondent section, but as a summary, they are used to generalize the problem and work only with the atom's coordinates.

Since usually we may work with a big box with a very large number of particles, there are interactions between distant particles that are very small and thus can be neglected to decrease the computational cost of the simulation. Therefore, a *cuttoff* distance (r_c) is defined, which usually is smaller than $L/2$, from which particles separated a higher distance than that their contribution to the potential energy will be considered zero. Then, the new truncated potential energy can be expressed as in Equation 4. Nevertheless, this truncation creates a big discontinuity in the energy, see Figure 3a, which can lead to problems in the force computation. To overcome this problem, the potential energy is "shifted" with the potential at the cutoff distance, see Equation 5, and a smoother expression of the potential is obtained, see Figure 3b.

$$V_{trunc}(r_{ij}) = \begin{cases} V(r_{ij}), & r_{ij} \le r_c \\ 0\ , & r_{ij} > r_c \end{cases} \tag{4}$$

$$V_{trunc-shift}(r_{ij}) = \begin{cases} V(r_{ij}) - V(r_c), & r_{ij} \le r_c \\ 0\ , & r_{ij} > r_c \end{cases} \tag{5}$$

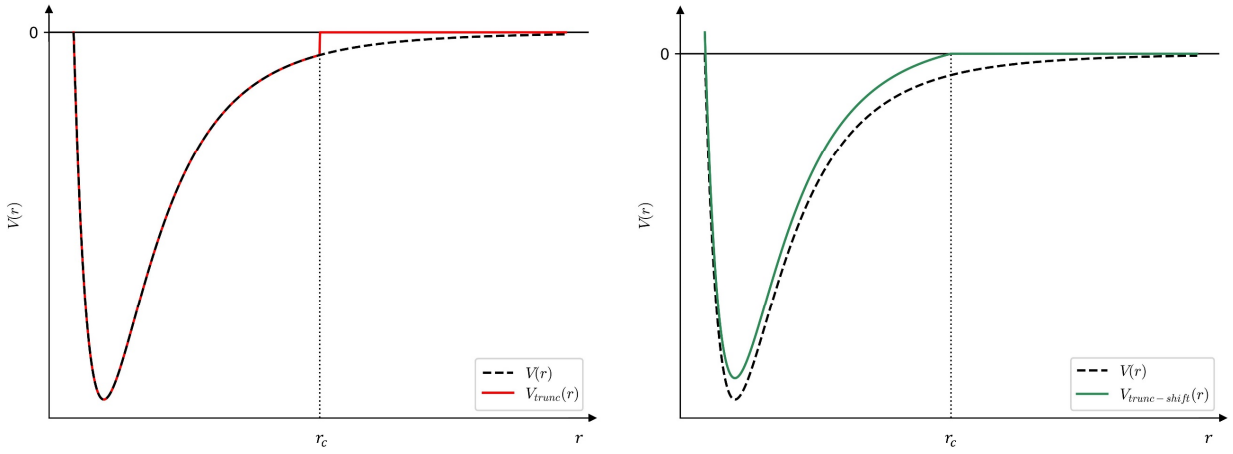
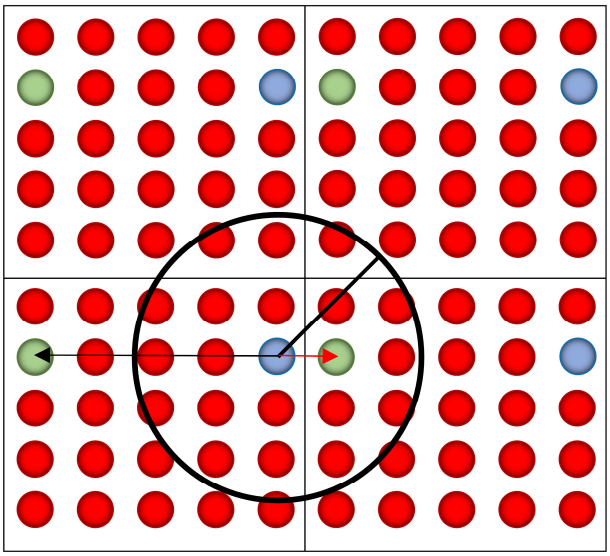


Figure 3. (a) Truncated Lennard-Jones potential and (b) shifted with $V(r_c)$.

One other thing to consider int the interaction calculation is the minimum image convention. If we apply periodic boundary conditions (PBC) to the system in order to model it as an "infinite" system, the particles will interact with the particles in the nearest box, the so-called image, and these interactions must be also computed. The minimum image convention can be taken into account considering a cutoff radius lower than $L/2$, and applying PBC to the vector between two particles so that the vector is redirected to point to the closest image of the paired particle. In Figure 4 there is a visual representation of the minimum image convention, alongside the PBC subroutine used to implement it.

Finally, we have all the steps necessary to compute the interactions. In Figure 5 is found the subroutine for computing the truncated and shifted potential energy and the force at a given timestep, considering the minimum image convention with PBC.



```
! PBC for 1 particle, MIC
subroutine pbc(rij,L)
implicit none
real, intent(inout),dimension(3) :: rij
real, intent(in) :: L
integer :: d,i

do d=1,3
    if (rij(d)>(L/2.)) then
        rij(d) = rij(d) - L
    else if (rij(d)<-(L/2.)) then
        rij(d) = rij(d) + L
    end if
end do
end subroutine pbc
```

Figure 4. Visual representation of the minimum image convention (left), where the back arrow in the drawing is the vector between two particles in the box and the red arrow indicates the "real" minimum distance considering the box image. And the PBC subroutine (right) that implements this minimum image convention in the simulation, a PBC for the distance vector to account for the interaction of the mirror image (box centered in 0).

```
! Computes truncated Lenard-Jones Energy and Forces with PBC
subroutine getForces(r,cutoff,L,addpbc,E,F)
implicit none
real, intent(in),dimension(:,:) :: r
real, intent(in) :: cutoff,L
logical, intent(in) :: addpbc
real, intent(out) :: E
real, intent(inout), dimension(:,:) :: F
real :: d2,d6,d8,d12,d14, cf2,cf6,cf12
real, dimension(3) :: rij
integer :: i,j

N = size(r, dim=1)
cf2 = cutoff*cutoff
F = 0
E = 0
do i = 1,N
    do j=i+1,N
        rij = r(i,:) - r(j,:)

        if (addpbc) then
            call pbc(rij,L)
        end if
        d2 = sum(rij**2)
        if (d2<cf2) then
            d6=d2*d2*d2; d12=d6*d6
            d8=d6*d2; d14=d8*d6
            cf6=cf2*cf2*cf2; cf12=cf6*cf6

            F(i,:) = F(i,:) + (48./d14 - 24./d8)*rij
            F(j,:) = F(j,:) - (48./d14 - 24./d8)*rij
            E = E + 4.*(1./d12 - 1./d6) - 4.*(1./cf12 - 1./cf6)
        end if
    end do
end do
end subroutine getForces
```

Figure 5. Subroutine to obtain the energy and forces in a simulation step for a given configuration. The energy is truncated to a certain cutoff, and periodic boundary conditions (PBC) are applied to consider minimum image convention.

Integration

Now that we have computed the interaction of the system, the particles will move according to this interaction. This motion is achieved integrating Newton's equations of motion with the calculated force. Since we are running a computer simulation, the motion equations need to be discretized, thus, along with the number of steps, the timestep between each frame (dt) also must be given. There exist different algorithms for integrating the Newton's equations of motion, and here three of them are discussed and used as possible integrators in the simulation code.

The first one, and the simplest, is the Euler integration, see Equation 6, which first integrates the new position of the particles and then computes the velocity from the force. Being the simplest, this method comes with some limitations, such as introducing an energy drift in the total energy, which should remain constant, or the lack of symmetry under time reversal.

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i dt + \frac{\boldsymbol{F}_i(t)}{2m_i} dt^2 \qquad \boldsymbol{v}_i(t+dt) = \boldsymbol{v}_i(t) + \frac{\boldsymbol{F}_i(t)}{m_i} dt \tag{6}$$

Later appeared the Verlet algorithm,[1] see Equation 7, which does not use the velocity for the integration of the new positions but requires the positions of the previous steps. This method conserves the total energy of the system and also allows for temporal inversion.

$$\boldsymbol{r}_i(t+dt) = 2\boldsymbol{r}_i(t) - \boldsymbol{r}_i(t-dt) + \frac{\boldsymbol{F}_i(t)}{m_i} dt^2 \qquad \boldsymbol{v}_i(t) = \frac{\boldsymbol{r}_i(t+dt) - \boldsymbol{r}_i(t-dt)}{2dt} \tag{7}$$

And finally, the Velocity Verlet algorithm,[2] Equation 8, which integrates the new positions as in the Euler method, but two force calculations are needed for velocities, at the current step and with the new positions.

$$\boldsymbol{r}_i(t+dt) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i dt + \frac{\boldsymbol{F}_i(t)}{2m_i} dt^2 \qquad \boldsymbol{v}_i(t+dt) = \boldsymbol{v}_i(t) + \frac{\boldsymbol{F}_i(t) - \boldsymbol{F}_i(t+dt)}{2m_i} dt \tag{8}$$

Now that we have the velocities at each step, the kinetic energy can be calculated:

$$K = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2 \tag{9}$$

And, at each step, the instantaneous temperature (T_{inst}), can be computed from the kinetic energy:

$$T_{inst} = \frac{2K}{k_B N_{dof}} = \frac{2K}{3k_B N} \tag{10}$$

where N_{dof} are the number of degrees of freedom, which in our case are the number of spatial dimensions for each particle ($3N$).

The subroutines for the mentioned integrators and the kinetic energy calculation are found on Figure 6.

New positions

Once the equations of motion are integrated, the particles move to a new position. As mentioned before, we are running the simulation with periodic boundary conditions, so a particle that leaves one wall of the box re-enters the box from the opposite side. In topological terms, the box with the PBC can be thought of as being mapped onto a torus. To implement this, in Figure 7 is the subroutine that is used to apply PBC to all the coordinates. With the new coordinates, the interactions are computed again, calculating the correspondent velocities and moving another step of the simulation, looping until the desired number of steps is achieved.

```
! Euler integrator
subroutine euler(r,v,dt,F,Ekin,Epot,
   cutoff,L,addpbc)
implicit none
logical, intent(in) :: addpbc
real, intent(in) :: dt, cutoff, L
real, intent(inout), dimension(:,:) :: F
real, intent(inout),dimension(:,:) :: r, v
real,intent(out) :: Ekin,Epot

call getForces(r,cutoff,L,addpbc,Epot,F)

r = r + v*dt + 0.5*F*dt*dt
v = v + F*dt
call pbcN(r,L)
call kinetc(v,Ekin)

end subroutine euler
```

```
! Kinetic Energy Calculation
subroutine kinetc(v,Ekin)
implicit none
real, intent(in),dimension(:,:) :: v
real, intent(out) :: Ekin
integer :: N,i

Ekin = 0
N = size(v, dim=1)
do i=1,N
    Ekin = Ekin + sum((v(i,:)**2)/2.)
end do

end subroutine kinetc
```

```
! Verlet integrator
subroutine verlet(r,rold,v,dt,F,Ekin,Epot,
   cutoff,L,addpbc)
implicit none
logical, intent(in) :: addpbc
real, intent(in) :: dt, cutoff, L
real, intent(inout), dimension(:,:) :: F
real, intent(inout),dimension(:,:) :: r,rold,v
real,intent(out) :: Ekin,Epot
real, dimension(:,:), allocatable :: roldtemp
allocate(roldtemp(size(r, dim=1),3))

call getForces(r,cutoff,L,addpbc,Epot,F)
roldtemp = r
r = 2*r - rold + F*dt*dt
call pbcN(r,L)
v = (r - rold)/(2*dt)
rold = roldtemp

call kinetc(v,Ekin)

deallocate(roldtemp)
end subroutine verlet
```

```
! Velocity-Verlet integrator
subroutine velocityVerlet(r,v,dt,F,Ekin,Epot,
    cutoff,L,addpbc)
implicit none
logical, intent(in) :: addpbc
real, intent(in) :: dt, cutoff, L
real, intent(inout), dimension(:,:) :: F
real, intent(inout),dimension(:,:) :: r, v
real,intent(out) :: Ekin,Epot

call getForces(r,cutoff,L,addpbc,Epot,F)
r = r + v*dt + 0.5*F* dt*dt
call pbcN(r,L)
v = v + 0.5*F*dt
call getForces(r,cutoff,L,addpbc,Epot,F)
v = v + 0.5*F*dt
call kinetc(v,Ekin)

end subroutine velocityVerlet
```

Figure 6. Euler, Verler and Velocity Verlet integrators subroutines.

```
! PBC for N particles
subroutine pbcN(r,L)
implicit none
real, intent(inout),dimension(:,:) :: r
real, intent(in) :: L
integer :: d,i

do i=1,size(r,dim=1)
    do d=1,3
        if (r(i,d)>(L)) then
            r(i,d) = r(i,d) - L
        else if (r(i,d)<0) then
            r(i,d) = r(i,d) + L
        end if
    end do
end do
end subroutine pbcN
```

Figure 7. Subroutines for implementing periodic boundary conditions (PBC) for the full positions matrix (box centered in $L/2$).

Thermostat

In the cases that we want to run simulations with a constant temperature, such as in the NVT ensemble (canonical), the temperatures must be controlled. There are different forms to maintain the temperature fixed, but the one used here is the Andersen thermostat. In the approach proposed by Andersen[3] the system is thermally coupled to a fictitious heat bath with the desired temperature. The coupling is represented by stochastic collisions that act occasionally on randomly selected particles. With a certain probability (nu) a "lucky" particle interacts with the bath and has its velocity changed to a random number from a normal distribution with $\sigma = \sqrt{k_B T/m}$. The values form the normal distribution can be gathered from the Box-Muller method.[4] The subroutines used for the Andersen thermostat and Box-Muller method are given in Figure 8.

```
subroutine thermostat(v,nu,sigma)
implicit none
real, intent(inout), dimension(:,:) :: v
real, intent(in) :: nu,sigma
real :: x1,x2,xo1,xo2
integer :: N,i,d

N = size(v,dim=1)
do i=1,N
    if (rand()<nu) then
        do d=1,3
        x1=rand();x2=rand()
        call boxmuller(sigma,x1,x2,xo1,xo2)
        v(i,d) = xo1
        end do
    end if
end do
end subroutine
```

```
subroutine boxmuller(sigma,x1,x2,xo1,xo2)
implicit none
real, intent(in) :: x1,x2, sigma
real, intent(out) :: xo1, xo2
real :: pi
pi = 4.*atan(1.)

xo1=sigma*sqrt(-2.*(log(1.-x1)))*cos(2.*pi*x2)
xo2=sigma*sqrt(-2.*(log(1.-x1)))*sin(2.*pi*x2)

end subroutine boxmuller
```

Figure 8. Subroutine for implementing the Andersen thermostat and to get random numbers that follow a standard, normal distribution, using the Box-Muller method.

Units

For systems consisting of just one type of molecule, such as the one in this simulation, it is practical to use the mass of the molecule as a fundamental unit, *i.e.*, set $m_i = 1$. Consequently, the particle momenta p_i and velocities v_i become numerically identical, as do the forces F_i and accelerations, a_i. This approach can be extended further. If the molecules interact by pair potentials of a simple form, such as the Lennard-Jones potential, they are completely specified by a few parameters, such as ε and σ, then further fundamental units of energy, length, etc. may be defined. From these definitions, units of other quantities (pressure, time, momentum, etc.) follow directly. The reduced unit representation of each variable is given in Table 1. Notice that using reduced units, the simulation can be performed for different particle types, then the conversion lies only in using the correspondent m, ε and σ of the particle.

Quantity	In Real Units	In Reduces Units
Time	t	$t^* = t\sqrt{\varepsilon/m\sigma^2}$
Distance	r	$r^* = r/\sigma$
Density	ρ	$\rho^* = \rho\sigma^3$
Temperature	T	$T^* = k_B T/\varepsilon$
Energy	E	$E^* = E/\varepsilon$
Force	F	$F^* = \sigma F/\varepsilon$
Pressure	P	$P^* = P\sigma^3/\varepsilon$

Table 1. Reduced units expressions for each desired quantity.

1b. Simulation

The code described in section 1a is used to perform a simulation of 500000 steps of an isolated system (with no contact with thermal bath) with $N = 256$ particles. The particles are initialized in a fcc cell with reduced density $\rho^* = 0.7$ and initial velocities that follow a bimodal distribution with a kinetic energy compatible with a temperature $T^* = 100$. Two different integrators are considered, the Velocity Verlet algorithm and the Euler method. The simulations are executed at different timesteps ($dt = 10^{-3}, 10^{-4}, 10^{-5}$) and always using a cutoff of $0.4L\,\sigma^{-1}$. The results are gathered and discussed in the following segments. In Figure 9 and Figure 10 are shown the energies (kinetic, E_{kin}, potential, E_{pot},and total, E), the temperature, T, and total momentum, P_T as a function of the reduced simulation time with the studied timesteps, using the Velocity Verlet and Euler integrator, respectively.

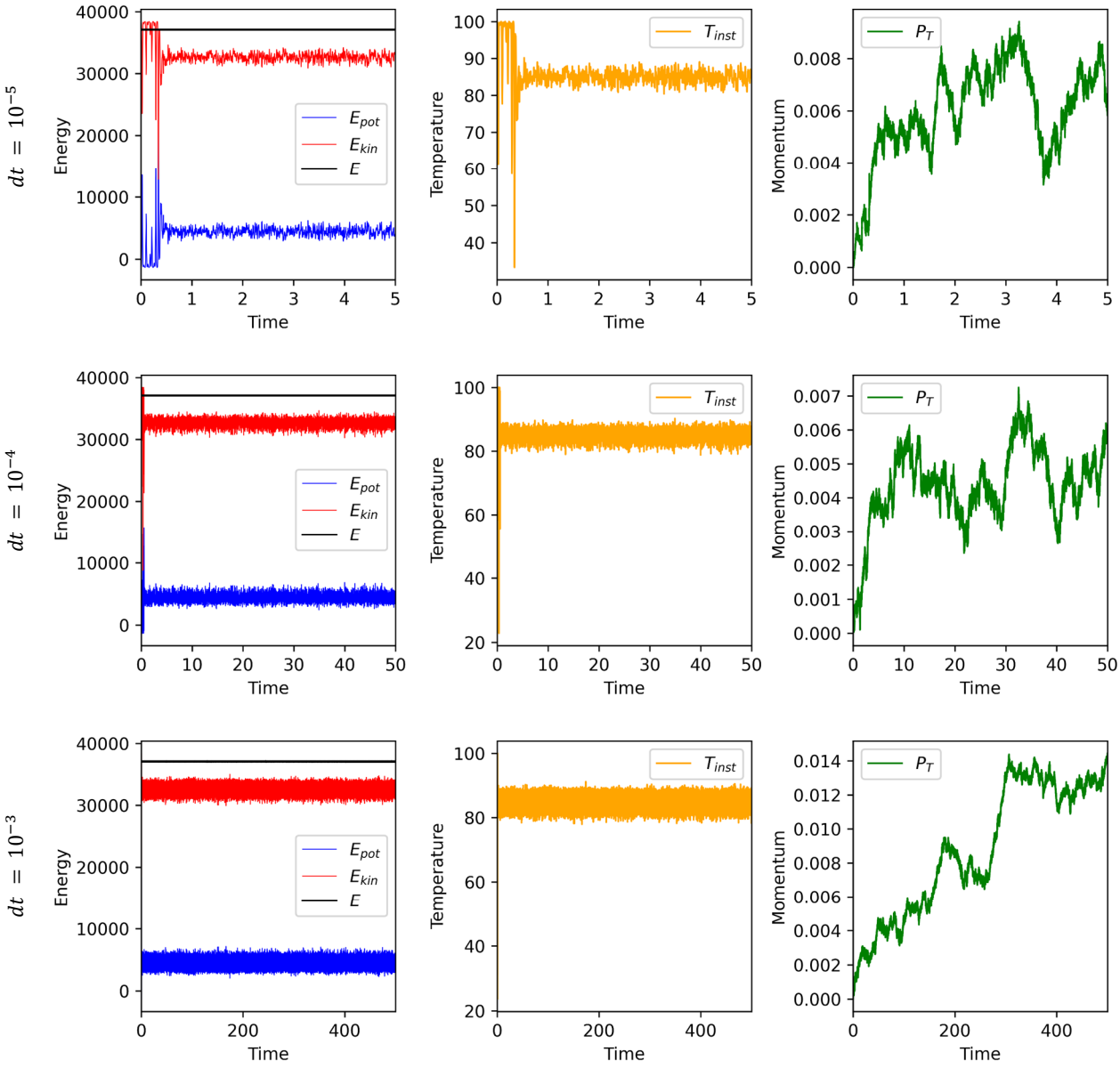


Figure 9. Thermodynamic data of the simulation runs with the Velocity Verlet integrator. First, second and third columns represent the energy contributions (E_{kin}, E_{pot}, E), instant temperature (T_{inst}), and total momentum (P_T), respectively. While rows indicate the correspondent timestep used.

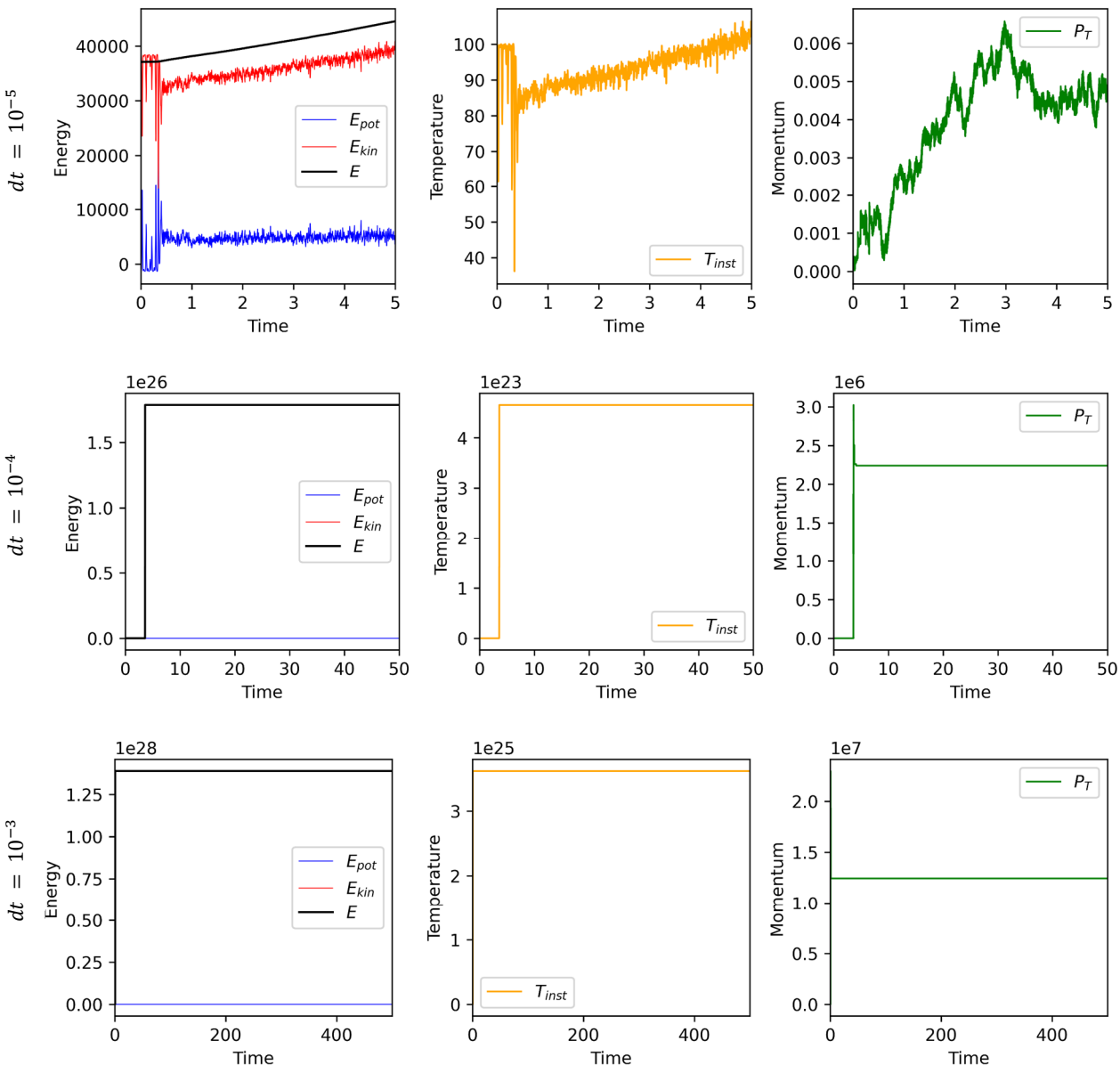


Figure 10. Thermodynamic data of the simulation runs with the Euler integrator. First, second and third columns represent the energy contributions (E_{kin}, E_{pot}, E), instant temperature (T_{inst}), and total momentum (P_T), respectively. While rows indicate the correspondent timestep used.

As presented in the Figures, The Velocity Verlet algorithm presents a much more stable approach, since with the Euler method, given the energy shift, with higher timesteps the velocity of the particles become uncontrollable and the energy rises to huge values, the system "explodes" and not even the PBC are enough to maintain the particles in the box. The Euler integrator works best with very small time-steps, *e.g.* with $dt = 10^{-5}$, as seen in the results, but even that is not enough to neglect the very clear energy shift produced.

On the other hand, the energies seem to remain conserved with the Velocity Verlet integrator. With lower timesteps, 10^{-5}, one can see at the beginning of the simulation some big shifts in the potential and kinetic energies, though the total energy keeps constant. In a relatively few number of steps, in all cases the system reaches equilibrium, and the energies and temperatures fluctuate over a constant value. The mean values of the energies, temperature and total momentum are gathered in Table 2. The averaged E and T of the system remain almost the same when changing the timestep, as expected, but the potential and kinetic energies are slightly different, where E_{pot} presents higher values at larger dt, while for E_{kin} it is the contrary.

dt	$\langle E_{pot} \rangle$	$\langle E_{kin} \rangle$	$\langle E \rangle$	$\langle T \rangle$	$\langle P_T \rangle$
10^{-5}	4241.88 ± 51.87	32843.66 ± 51.87	37085.54 ± 0.02	85.53 ± 0.14	0.006 ± 0.0001
10^{-4}	4382.93 ± 13.92	32705.83 ± 13.91	37088.76 ± 0.03	85.17 ± 0.04	0.004 ± 0.00004
10^{-3}	4416.20 ± 3.49	32652.93 ± 3.50	37069.14 ± 0.14	85.03 ± 0.01	0.009 ± 0.0001

Table 2. Block averages using 1000 blocks of 500 points for the energies, temperature and momentum gathered from the Velocity Verlet simulations at different timesteps.

In all Velocity Verlet cases, the temperature starts at $T = 100\ \varepsilon/k_B$ and quickly drops to the equilibrium temperature, around $85\ \varepsilon/k_B$. In terms of the total momentum, it seems to fluctuate and not be conserved, but the values are so close to zero that it could be attributed to numerical error. As seen in Table 2, the averaged total momentum is almost 0, which agrees with the initial conditions of initializing the velocities with a bimodal distribution.

One final thing to notice is that, even though the velocities were initialized at a bimodal distribution, once equilibrium is achieved, the velocity distribution evolves to follow a Maxwell-Boltzmann (MB) distribution centered at zero of type:

$$f(v) = \frac{1}{\sqrt{2\pi T}} e^{-\frac{v^2}{2T}} \tag{11}$$

In Figure 11, are represented the initial and final velocity distributions of the Velocity Verlet and Euler cases using $dt = 10^{-5}$, where you can clearly see the MB distribution shape. The MB distribution is fitted using the instantaneous temperature of the last frame. This evolution also occurs with the other studied timesteps in case of the Velocity Verlet integrator, but in the case of the Euler integrator, since the velocities become uncontrolled, the distribution cannot be interpreted.

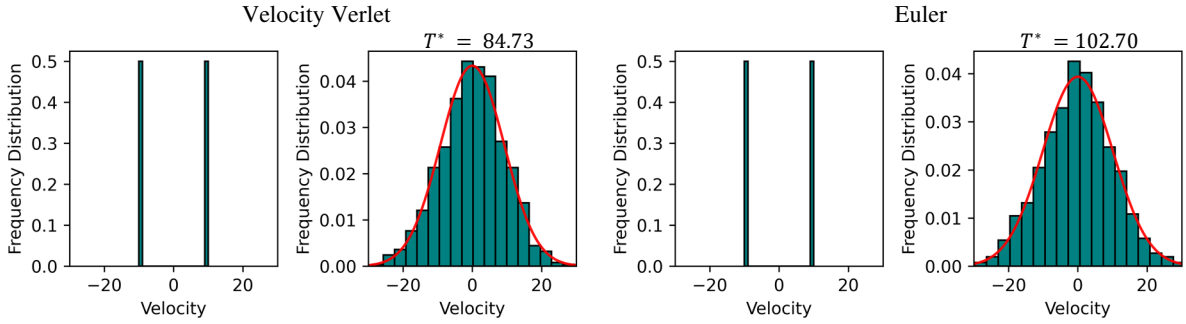


Figure 11. Velocity distributions at the initial (bimodal) and final configurations (Maxwell-Boltzmann) in the cases with $dt = 10^{-5}$. Left is using the Velocity Verlet integrator and right using the Euler method.

2. Analysis of Properties of a Lennard-Jones Liquid

Here, the code developed in the previous sections is modified in order to simulate a system of $N = 256$ Argon atoms that interact through a Lennard-Jones potential with $\varepsilon = 0.998\ kJ/mol$ and $\sigma = 3.4$ Å and timestep $dt = 10^{-4}$. Knowing that the atomic molar mass of Argon is $M = 40\ g/mol$ (which indicates an atomic mass of $m = M/N_A = 6.642 \cdot 10^{-23} g$) we have the necessary parameters to transform the reduced units to real units (Table 1). In this case, since ε is in kJ/mol, the Avogadro's Number (N_A) is added conveniently to the conversion factor, to obtain the observables in SI units.

Again, the system is initialized in a fcc cell but, in this case, to ensure the system is disordered, 10000 steps of "melting" the Argon at $T^* = 100$ are performed before starting the production cycles. Then, 500000 steps of equilibration are simulated at $T^* = 1.2$, both using a thermostat (Andersen's). This process is carried out for 6 different densities. The total energy and kinetic and potential contributions are presented alongside the pressure on Figure 12 as a function of the density. Moreover, on Table 3, are presented the block averaged values for different observables (energies, temperature and pressure). The first 50000 steps were discarded in order to compute the averages of the equilibrated system.

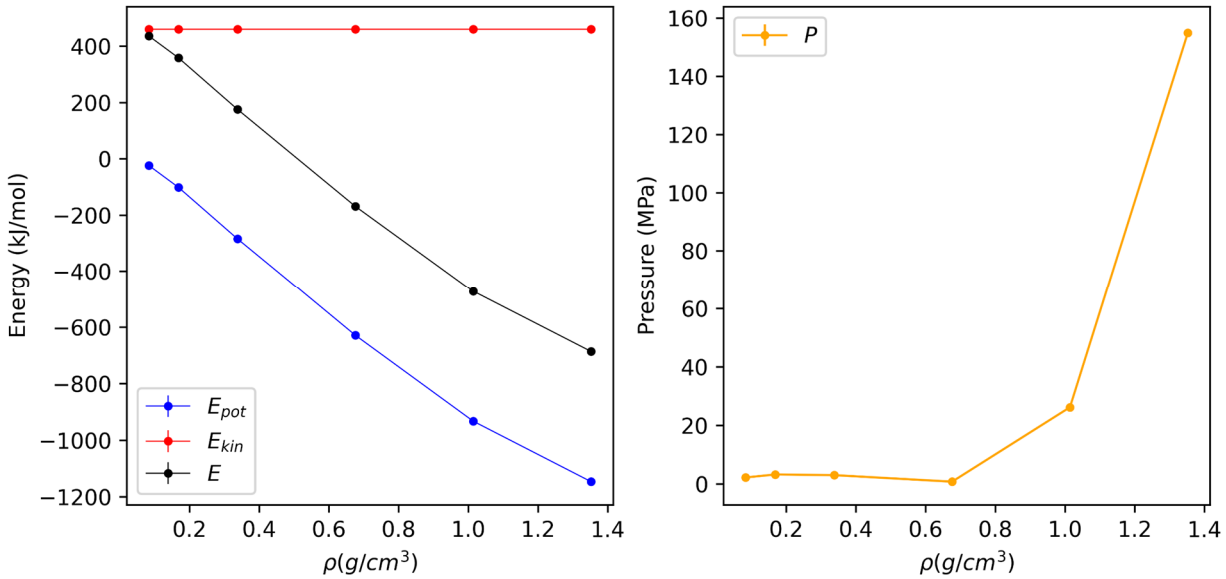


Figure 12. Energy contributions (left) and pressure (right) for the simulations at the different densities. Error bars are present but are so small they cannot be appreciated.

ρ^* (g/cm^3)	$\langle E_{pot} \rangle$ (kJ/mol)	$\langle E_{kin} \rangle$ (kJ/mol)	$\langle E \rangle$ (kJ/mol)	$\langle T \rangle$ (K)	$\langle P \rangle$ (Pa)
0.084 (0.05)	-23.9 ± 0.1	460.0 ± 0.2	426.1 ± 0.2	144.07 ± 0.05	$2.20 \cdot 10^6 \pm 2.2 \cdot 10^3$
0.169 (0.1)	-101.7 ± 0.3	460.0 ± 0.2	358.3 ± 0.3	144.07 ± 0.05	$3.23 \cdot 10^6 \pm 8.8 \cdot 10^3$
0.338 (0.2)	-284.3 ± 0.2	460.0 ± 0.2	175.7 ± 0.3	144.07 ± 0.05	$3.00 \cdot 10^6 \pm 4.1 \cdot 10^4$
0.676 (0.4)	-628.9 ± 0.3	460.0 ± 0.2	-168.9 ± 0.4	144.07 ± 0.05	$7.71 \cdot 10^5 \pm 1.4 \cdot 10^5$
1.014 (0.6)	-932.5 ± 0.7	460.0 ± 0.2	-472.5 ± 0.7	144.07 ± 0.05	$2.62 \cdot 10^7 \pm 5.0 \cdot 10^5$
1.352 (0.8)	-1146.1 ± 1.5	460.0 ± 0.2	-686.2 ± 1.5	144.07 ± 0.05	$1.55 \cdot 10^8 \pm 1.3 \cdot 10^6$

Table 3. Thermodynamic properties for the different densities. Averaged using the block average method. The values between parenthesis in the density correspond to the reduced units values.

The energy difference between densities is due mainly to potential energy contributions. One can see that the kinetic energy is almost constant in all cases, and that makes sense since it depends on the temperature, which is controlled and maintained fixed with the thermostat. On the other hand, the potential energy, and thus the total energy, decreases when increasing the density. Since with lower densities, the particles are further apart, most of the particles won't be at their equilibrium distance of the truncated LJ potential (Figure 3) and the energy will be higher. On the contrary, in the denser systems, the inter particle distances are much closer to the minimum in the LJ potential, making the potential energy lower.

The shape of the pressure curve for the different densities seem to agree with previous computational studies.[5,6] Which with lower densities, the pressure decreases (since the volume is higher and the temperature remains constant), but lower than $\rho < 0.676\ g/cm^3$ it seems to arrive at the liquid-vapour saturation limit, and it remains roughly constant.

The different densities of the system can be visualized in Figure 13, where a snapshot of the configuration at an equilibrated frame for each density are presented. Here we can clearly see the difference in separation between the particles. Moreover, if we visualize part of the trajectory once the system is equilibrated, the lower density systems show a more gas-like movement, while in higher densities the particles sort of vibrate and move as a more condensed liquid.

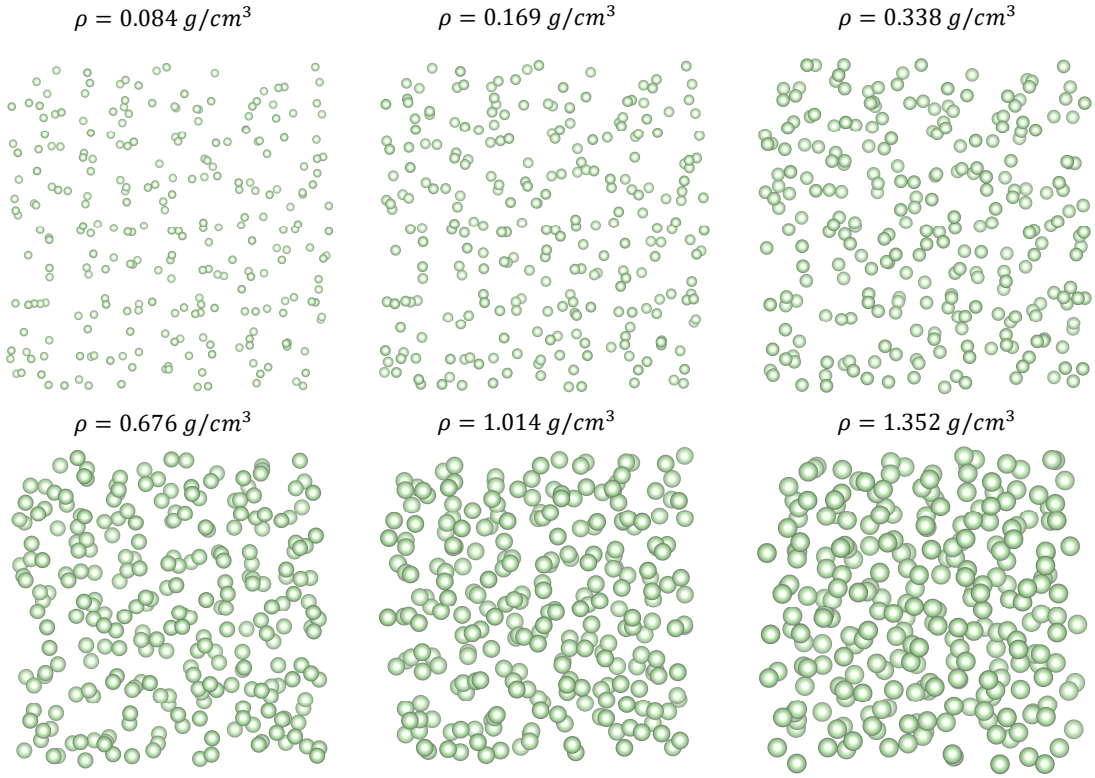


Figure 13. Snapshots of the trajectory at equilibrium for the different densities.

For the different simulation, the mean square displacement (MSD) was computed and the results, as a function of time, are presented in Figure 14. This figure also contains the MSD from an equilibrated point, where the values present a more linear form. From the gathered MSD over time, the diffusion coefficient (D) is computed with a linear fit to the data, knowing that $MSD \propto 6Dt$. In Figure 15 are the values for each density considering the full simulation, and only the parts when the data is linear. Nonetheless, both show similar values and, in both cases, higher densities present lower diffusion coefficients.

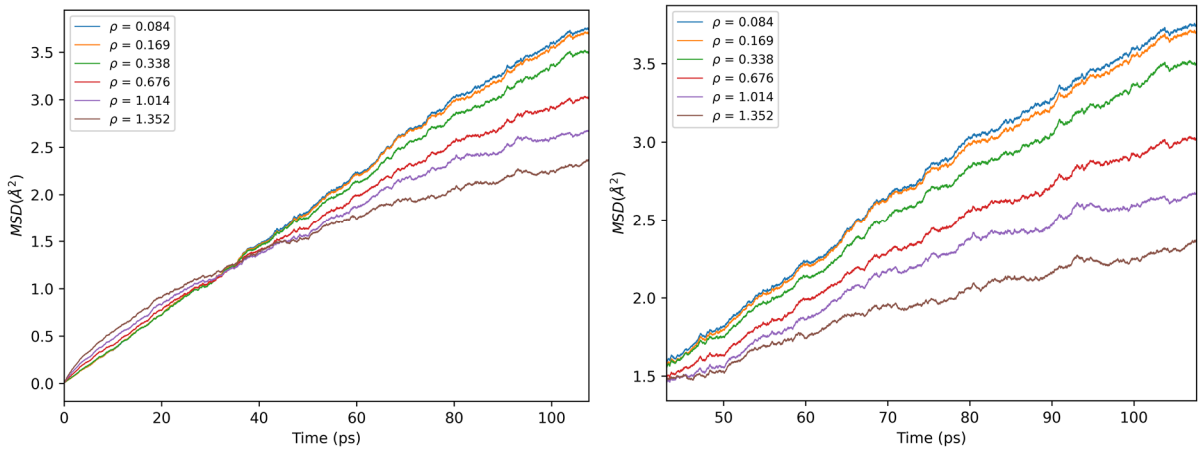


Figure 14. Plots of the Mean Square Distance (MSD, in Å) for the production cycles as the simulation evolves. On the left are present the values for the full production, while on the right is present only the part after equilibration, when the curves tend to a linear function. Densities in g/cm^3.

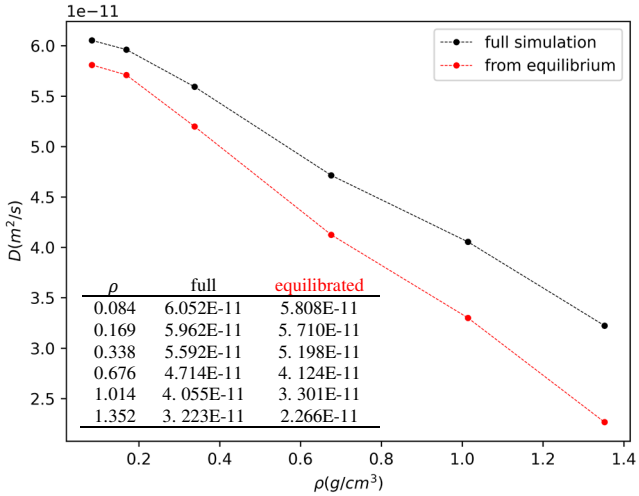


Figure 15. Diffusion coefficient for each density. Red dots indicate de values extracted discarding the non-equilibrated initial steps, and in black are the ones gathered from the full simulation. On the corner is also present the coefficients values in m^2/s.

The obtained results can be interpreted in terms of the phase diagram presented by Lin *et al.*[5], see Figure 16, which presents the different phases of Argon as a function of density and temperature. At lower densities, the system is in gas phase, and the particles are further apart, which leads to an easier auto diffusion, and therefore higher diffusion coefficients. On the contrary, with higher densities, the Argon presents liquid phase, with the particles being closer and more difficult to diffuse, leading to lower diffusion coefficients. On the other hand, according to the figure, the studied system at densities 0.338 and 0.676 g/cm^3 is in an unstable/metastable region, just below the critical point. But looking at the saved part of the trajectories, no signs of it nor separation of phases was observed. Although, with larger simulation times and saving the full trajectory, better visualizations may be achieved. Nevertheless, in those same cases, some of the instantaneous pressures obtained were negative, indicating certain unstability of the system.

Finally, a simulation of the same system at $\rho^* = 0.8$ was performed but now with higher temperature $T^* = 2\ (240.1\ K)$. The radial distribution function for both systems is calculated and shown in Figure 17. Besides the noise, which could be avoided by using more simulation frames to compute it, both present very similar shapes and in agreement to previously reported data.[7,8]

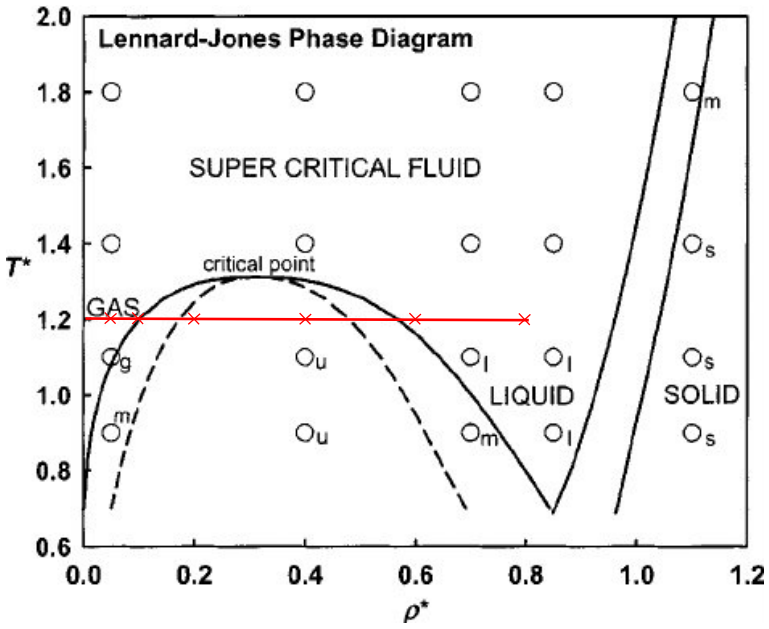


Figure 16. Phase diagram of Lennard-Jones fluid in reduced units.[5] The crossed red line indicates the study points of the present simulation. Values in reduced units.

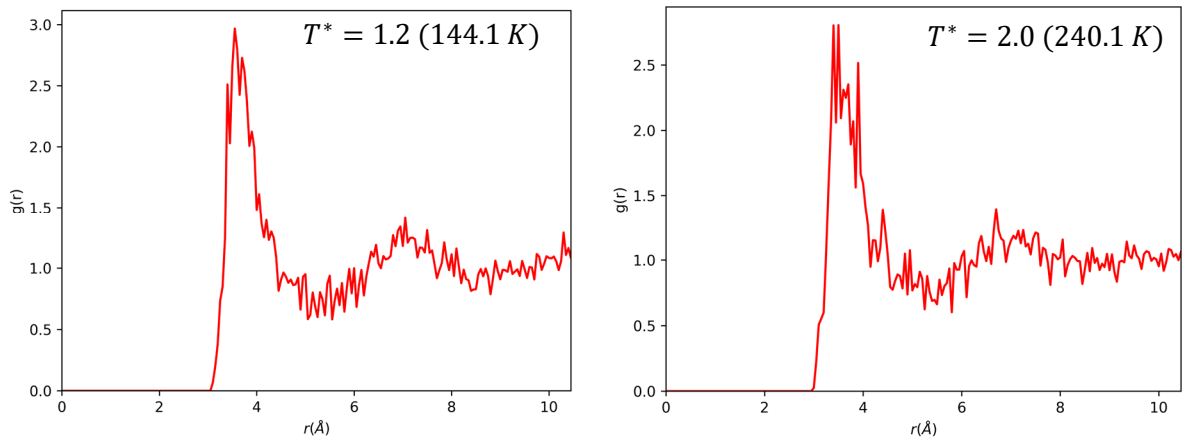


Figure 17. Radial distribution functions for $\rho = 1.352\ g/cm^3$ for $T^* = 1.2\ (144.1\ K)$ (left) and $T^* = 2.0\ (240.1\ K)$ (right). Both computed using 100 frames of the simulation at equilibrium.

Bibliography

1. L. Verlet, *Physical review online archive*, 1967, **159**, 98–103.

2 . W. Swope, H. Andersen, P. Berens and K. Wilson, *J Chem Phys*, 1982, **76**, 637–649.

3. H. Andersen, *J Chem Phys*, 1980, **72**, 2384–2393.

4. G. Box and M. Muller, *The Annals of mathematical statistics*, 1958, **29**, 610–611.

5. S. T. Lin, M. Blanco and W. A. Goddard, *Journal of Chemical Physics*, 2003, **119**(22), 11792–11805.

6. A. B. Kaplun and A. B. Meshalkin, *High Temperature*, 2003, **41**(3), 319–326.

7. J. L. Yarnell, M. J. Katz, R. G. Wenzel and S. H. Koenig, *Physical Review A*, 1973, 7, 2130–2144.

8. K. Ghosh and C. v. Krishnamurthy, *Physical Review E*, 2018, **97**(1), 012131**.**

Appendix

Fortran90 Code used for obtaining the results of the report. This is an upgraded version of the one mentioned in section 1a test, which includes extra subroutines and equilibration used for section 2 calculations. Both codes, with the correspondent Python scripts to analyse the data will be present on the attached file and on this GitHub repository: https://github.com/diegonti/MMC/tree/main/MoMo/B2-MD.

```
program main

    ! Variable declaration
    implicit none
    real, parameter :: pi = 4.*atan(1.)
    real, parameter :: Na = 6.0221408e23
    real, parameter :: kb = 1.380649e-23
    real :: L,dens,a, E, cutoff, dt, Ekin,Epot, T,s,nu,Tinst,P,pt_mod,msd
    real :: ru_time,ru_dens,ru_dist,ru_temp,ru_E,ru_press,ru_vel,ru_mom, eps,sigma,MM,mass
    real, allocatable,dimension(:,:) :: r,rold,v,F
    real, dimension(3) :: pt_vec
    character(len=10) :: cell
    character(len=25) :: fileTraj,fileData,file1,file2,fileV1,fileV2
    integer :: M, N, i,j,k, Nsteps, frames_to_RDF
    logical :: addpbc

    ! Initial parameters/inputs                        (216)         (256)             (250)
    N = 256         ! Choose the right number, N=M**3 for sc, N=4M**3 for fcc, N=2M**3 for bcc
    dens = 0.8      ! (0.05; 0.1; 0.2; 0.4; 0.6; 0.8)
    cell = "fcc"    ! Specify unit cell (sc, fcc, bcc)

    sigma = 3.4     ! Amstrongs
    eps = 0.998     ! kJ/mol
    MM = 40         ! g/mol
    mass = MM/Na    ! g

    L = (N/dens)**(1./3.)
    if (cell=="sc") then; M = (N)**(1./3.)
    else if (cell=="fcc") then; M = (N/4.)**(1./3.)
    else if (cell=="bcc") then; M = (N/2.)**(1./3.)
    end if

    a = L/M              ! Lattice parameter
    dt = 0.0001          ! timestep
    cutoff = 0.4*L       ! Cutoff for interactions
    nu = 0.1             ! Thermostat nu
    addpbc = .true.      ! To add or not PBC
    frames_to_RDF = 1000 ! Frames to save positions to compute later RDF

    ! Name of files the data is outputed
    file1 = "initial.xyz"                   ! Initial coordinates
    file2 = "final.xyz"                     ! Final coordinates
    fileV1 = "initialV.dat"                 ! Initial velocities
    fileV2 = "finalV.dat"                   ! Final velocities
    fileData = "thermodynamics_raw.dat"     ! Observables data
    fileTraj = "trajectory.xyz"             ! Saved trajectories

    ! Conversion factors between reduced and real units
    ru_time = sqrt(mass*Na*(sigma*1e-10)**2 / (eps*1e6))    ! t in seconds
    ru_dist = sigma                                         ! distance in Ang
    ru_dens = 1e24 * MM / (Na*sigma**3)                     ! density in g/mL
    ru_E = eps                                              ! energy in kJ/mol
    ru_press = 1e3*eps/((sigma*1e-10)**3 * Na)              ! pressure in Pa
    ru_temp = 1e3*eps/(kb*Na)                               ! temperature in K
    ru_vel = ru_dist/ru_time *1e-10                         ! velocity in Ang/s
    ru_mom = m*ru_vel

    ! Print parameters
```

```
    print*, "Input parameters (reduced units)"
    print*, "Unit Cell:           ", cell
    print*, "Number of particles: ", N
    print*, "Density:             ", dens
    print*, "Box length:          ", L
    print*, "Lattice parameter:   ", a
    print*

    print*, "Input parameters (real units)"
    print*, "Unit Cell:           ", cell
    print*, "Number of particles: ", N
    print*, "Density:             ", dens*ru_dens
    print*, "Box length:          ", L*ru_dist
    print*, "Lattice parameter:   ", a*ru_dist
    print*


    allocate(r(N,3))
    allocate(v(N,3))
    allocate(F(N,3))
    allocate(rold(N,3))

    ! Opening output files
    open(1,file=file1,status="replace")
    open(2,file=file2,status="replace")
    open(3,file=fileData,status="replace")
    write(3,"(8a15)") "# Time","Epot","Ekin","E","Momentum","Tinst","Pressure","MSD"
    open(4,file=fileV1,status="replace")
    open(5,file=fileV2,status="replace")
    open(8,file=fileTraj,status="replace")

    ! Initializing positions (r) and velocities (v)
    call initializePositions(N,dens,r,cell)
    v = 0

    call writePOS(r*ru_dist,1)     ! Saves initial geometry
    call writePOS(v*ru_vel,4)      ! Saves initial velocities


    ! Main MD loop to disorder the configuration !!!!!!!!!!!!!!!!!!
    Nsteps = 10000      ! Number of disordering steps
    T = 100             ! Reduced temperature (melting)
    Tinst = T
    s = sqrt(T)         ! Sigma (themrostat)
    print*, "Starting Melting at T = ",T*ru_temp
    do i=1,Nsteps
        call velocityVerlet(r,v,dt,F,Ekin,Epot,P,cutoff,dens,Tinst,addpbc)
        call thermostat(v,nu,s)
    end do


    ! Main MD production loop !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    Nsteps = 500000     ! Number of simulation steps
    T = 1.2             ! Reduced temperature
    Tinst = T
    s = sqrt(T)         ! Sigma (thermostat)
    rold = r            ! Saving initial coordinates (MSD)

    print*, "Starting Production Cycles at T = ",T*ru_temp
    write(*,"(a)",advance='no') "Completed (%): "
    do i=1,Nsteps
        call velocityVerlet(r,v,dt,F,Ekin,Epot,P,cutoff,dens,Tinst,addpbc)
        call thermostat(v,nu,s)

        call momentum(v,pt_vec,pt_mod)
        Tinst = 2.*Ekin/(3.*N)
```

```
        call getMSD(r,rold,L,msd)

        write(3,*) i*dt*ru_time,Epot*ru_E,Ekin*ru_E, (Epot+Ekin)*ru_E, pt_mod*ru_mom,
Tinst*ru_temp,P*ru_press,msd*ru_dist**2

        ! Saves last snapshots of trajectoy (to compute RDF correctly)
        if ((i>=(Nsteps-frames_to_RDF)) .and. (i<=Nsteps)) then
            call writePOS(r*ru_dist,8)
        end if

        if (mod(i,Nsteps/10)==0) then
            write(*,'(1x,i0)',advance='no') (100*i)/Nsteps
        end if
    end do
    print*,

    call writePOS(r*ru_dist,2)  ! Saves final position
    call writePOS(v*ru_vel,5)  ! Saves final velocities


    deallocate(r)
    deallocate(v)
    deallocate(F)
    deallocate(rold)


    contains
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!      Subroutines      !!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


        subroutine boxmuller(S, X1,X2, XOUT1, XOUT2)
        implicit none
        real, intent(in) :: x1,x2, s
        real, intent(out) :: xout1, xout2
        real pi
        pi = 4.*atan(1.)

        XOUT1=s*sqrt(-2.*(log(1.-x1)))*cos(2.*pi*x2)
        XOUT2=s*sqrt(-2.*(log(1.-x1)))*sin(2.*pi*x2)

        end subroutine boxmuller

        ! Andersen Thermostat
        subroutine thermostat(v,nu,s)
        implicit none
        real, intent(inout),dimension(:,:) :: v
        real, intent(in) :: nu,s
        real :: x1,x2,xout1,xout2
        integer :: N,i,d

        N = size(v,dim=1)
        do i=1,N
            if (rand()<nu) then
                do d=1,3
                    x1=rand();x2=rand()
                    call boxmuller(s,x1,x2,xout1,xout2)
                    v(i,d) = xout1
                end do
            end if
        end do
        end subroutine


        ! To choose and run the specified initial position
```

```
subroutine initializePositions(N,dens,r,cell)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
character(len=10), intent(in) :: cell

if ((index(cell,"fcc")==1) .OR. (index(cell,"FCC")==1))  then
    call initFCC(N,dens,r)
else if((index(cell,"sc")==1) .OR. (index(cell,"SC")==1)) then
    call initSC(N,dens,r)
else if ((index(cell,"bcc")==1) .OR. (index(cell,"BCC")==1))  then
    call initBCC(N,dens,r)
end if

end subroutine initializePositions

! Initial position and velocities of two particles
subroutine init2(L,r,v)
implicit none
real, intent(in) :: L
real, intent(inout),dimension(:,:) :: r,v

r = reshape( (/L/2.-0.75,0.,0., L/2+0.75,0.,0./),shape(r), order=(/2,1/))
v = reshape( (/0.,0.,0., 0.,0.,0./),shape(v), order=(/2,1/))

end subroutine

! Initial positions for SC cell
subroutine initSC(N,dens,r)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
real :: L,a
integer :: p, i,j,k, M

L = (N/dens)**(1./3.)
M = N**(1./3.)
a = L/M
p = 1
do i=0,M-1
    do j=0,M-1
        do k=0,M-1
            r(p, :) = [i,j,k]
            p=p+1
        end do
    end do
end do
r = r*a
end subroutine initSC

! Initial positions for FCC cell
subroutine initFCC(N,dens,r)
implicit none
real, intent(inout),dimension(:,:) :: r
integer, intent(in) :: N
real, intent(in) :: dens
real, dimension(4,3) :: ucell
real :: L,a
integer :: i,j,k, p,at,M

L = (N/dens)**(1./3.)
M = (N/4.)**(1./3.)
a = L/M
```

```
        ucell = reshape( (/0.,0.,0., 0.,0.5,0.5,  0.5,0.,0.5,  0.5,0.5,0./),shape(ucell),
order=(/2,1/))
        p = 1
        do i=0,M-1
            do j=0,M-1
                do k=0,M-1
                    do at=1,size(ucell, dim=1)
                        r(p, :) = ucell(at,:) + (/i,j,k/)
                        p=p+1
                    end do
                end do
            end do
        end do
        r = r*a

        end subroutine initFCC

        ! Initial positions for BCC cell
        subroutine initBCC(N,dens,r)
        implicit none
        real, intent(inout),dimension(:,:) :: r
        integer, intent(in) :: N
        real, intent(in) :: dens
        real, dimension(2,3) :: ucell
        real :: L,a
        integer :: i,j,k, p,at,M

        L = (N/dens)**(1./3.)
        M = (N/2.)**(1./3.)
        a = L/M

        ucell = reshape( (/0.,0.,0., 0.5,0.5,0.5/),shape(ucell), order=(/2,1/))
        p = 1
        do i=0,M-1
            do j=0,M-1
                do k=0,M-1
                    do at=1,size(ucell, dim=1)
                        r(p, :) = ucell(at,:) + (/i,j,k/)
                        p=p+1
                    end do
                end do
            end do
        end do
        r = r*a
        end subroutine initBCC

        ! Initial velocities as Bimodal distribution
        subroutine initBimodal(T,v)
        implicit none
        real,intent(inout),dimension(:,:) :: v
        real,intent(in) :: T
        real :: vi
        integer :: N
        N = size(v, dim=1)
        v=0
        if (mod(N,2)/=0) then
            print*, "Number of particles (N) should be multiple of 2."
        end if
        vi = sqrt(T)
        v(1:N:2,:) = -vi
        v(2:N:2,:) = +vi

        end subroutine initBimodal

        ! Initialize velocities at zero
        subroutine initZero(v)
        implicit none
```

```
        real,intent(inout),dimension(:,:) :: v
        v = 0
        end subroutine initZero


        ! Writes current positions in the specified file (indexed)
        subroutine writePOS(r,fileN)
        implicit none
        real, intent(in),dimension(:,:) :: r
        integer, intent(in) :: fileN
        integer :: i, N

        N = size(r, dim=1)

        write(fileN,*) N
        write(fileN,*)
        do i= 1,N
            write(fileN,*) "Ar", r(i,:)
        end do

        end subroutine writePOS


        ! Computes truncated Lenard-Jones Energy and Forces with PBC
        subroutine getForces(r,cutoff,dens,Tinst,addpbc,E,F,P)
        implicit none
        real, intent(in),dimension(:,:) :: r
        real, intent(in) :: cutoff,dens,Tinst
        logical, intent(in) :: addpbc
        real, intent(out) :: E,P
        real, intent(inout), dimension(:,:) :: F
        real :: d2,d6,d8,d12,d14, cf2,cf6,cf12, L,V
        real, dimension(3) :: rij,Fij
        integer :: N,i,j

        N = size(r, dim=1)
        L = (N/dens)**(1./3.)
        cf2 = cutoff*cutoff
        F = 0
        E = 0
        P = (dens*Tinst)
        V = L**3
        do i = 1,N
            do j=i+1,N
                rij = r(i,:) - r(j,:)

                if (addpbc) then
                    call pbc(rij,L)
                end if


                d2 = sum(rij**2)
                if (d2<cf2) then
                    d6=d2*d2*d2; d12=d6*d6
                    d8=d6*d2; d14=d8*d6
                    cf6=cf2*cf2*cf2; cf12=cf6*cf6

                    Fij = (48./d14 - 24./d8)*rij

                    ! F(i) = -F(j)
                    F(i,:) = F(i,:) + Fij
                    F(j,:) = F(j,:) - Fij

                    !Energy: E = E(r) - E(cf) --> smoother truncation
                    E = E + 4.*(1./d12 - 1./d6) - 4.*(1./cf12 - 1./cf6)
```

```
                ! Pressure
                P = P + (1./(3.*V))*(dot_product(rij,Fij))

            end if
        end do
    end do
    end subroutine getForces


    ! Periodic Boundary Conditions for 1 particle
    subroutine pbc(rij,L)
    implicit none
    real, intent(inout),dimension(3) :: rij
    real, intent(in) :: L
    integer :: d,i

    do d=1,3
        if (rij(d)>(L/2.)) then
            rij(d) = rij(d) - L
        else if (rij(d)<-(L/2.)) then
            rij(d) = rij(d) + L
        end if
    end do

    end subroutine pbc

    ! do pbc pero <L <0 ??

    ! Periodic Boundary Conditions for a matrix
    subroutine pbcN(r,L)
    implicit none
    real, intent(inout),dimension(:,:) :: r
    real, intent(in) :: L
    integer :: d,i

    do i=1,size(r,dim=1)
        do d=1,3
            if (r(i,d)>(L)) then
                r(i,d) = r(i,d) - L
            else if (r(i,d)<0) then
                r(i,d) = r(i,d) + L
            end if
        end do
    end do

    end subroutine pbcN

    ! Implementation of the Verlet integrator
    subroutine verlet(r,rold,v,dt,F,Ekin,Epot,P,cutoff,dens,Tinst,addpbc)
    implicit none
    logical, intent(in) :: addpbc
    real, intent(in) :: dt, cutoff, dens,Tinst
    real, intent(inout), dimension(:,:) :: F
    real, intent(inout),dimension(:,:) :: r, rold,v
    real, intent(out) :: Ekin,Epot,P
    real, dimension(:,:), allocatable :: roldtemp

    allocate(roldtemp(size(r, dim=1),3))

    call getForces(r,cutoff,dens,Tinst,addpbc,Epot,F,P)
    roldtemp = r
    r = 2*r - rold + F*dt*dt
    call pbcN(r,L)
    v = (r - rold)/(2*dt)
    rold = roldtemp
```

```
        call kinetc(v,Ekin)

        deallocate(roldtemp)

        end subroutine verlet

        ! Implementation of the Velocity-Verlet integrator
        subroutine velocityVerlet(r,v,dt,F,Ekin,Epot,P,cutoff,dens,Tinst,addpbc)
        implicit none
        logical, intent(in) :: addpbc
        real, intent(in) :: dt, cutoff, dens,Tinst
        real, intent(inout), dimension(:,:) :: F
        real, intent(inout),dimension(:,:) :: r, v
        real,intent(out) :: Ekin,Epot,P

        call getForces(r,cutoff,dens,Tinst,addpbc,Epot,F,P)
        r = r + v*dt + 0.5*F* dt*dt
        call pbcN(r,L)
        v = v + 0.5*F*dt
        call getForces(r,cutoff,dens,Tinst,addpbc,Epot,F,P)
        v = v + 0.5*F*dt
        call kinetc(v,Ekin)

        end subroutine velocityVerlet


        ! Implementation of the Euler integrator
        subroutine euler(r,v,dt,F,Ekin,Epot,P,cutoff,dens,Tinst,addpbc)
        implicit none
        logical, intent(in) :: addpbc
        real, intent(in) :: dt, cutoff, dens,Tinst
        real, intent(inout), dimension(:,:) :: F
        real, intent(inout),dimension(:,:) :: r, v
        real,intent(out) :: Ekin,Epot,P

        call getForces(r,cutoff,dens,Tinst,addpbc,Epot,F,P)

        r = r + v*dt + 0.5*F*dt*dt
        v = v + F*dt
        call pbcN(r,L)
        call kinetc(v,Ekin)

        end subroutine euler

        ! Kinetic Energy Calculation
        subroutine kinetc(v,Ekin)
        implicit none
        real, intent(in),dimension(:,:) :: v
        real, intent(out) :: Ekin
        integer :: N,i

        Ekin = 0
        N = size(v, dim=1)
        do i=1,N
            Ekin = Ekin + sum((v(i,:)**2)/2.)
        end do

        end subroutine kinetc

        ! Compute the total momentum
        subroutine momentum(v,pt_vec,pt_mod)
        implicit none
        real, intent(in),dimension(:,:) :: v
        real, intent(out),dimension(3) :: pt_vec
        real, intent(out) :: pt_mod
        integer :: N,i
```

```
        N = size(v, dim=1)

        pt_vec = 0
        do i=1,N
            pt_vec = pt_vec + v(i,:)
        end do
        pt_mod = sqrt(sum(pt_vec**2))

        end subroutine momentum

        subroutine getMSD(r,rold,L,msd)
        implicit none
        real, intent(in) :: L
        real, intent(in),dimension(:,:) :: r,rold
        real,intent(out) :: msd
        real, dimension(3) :: rij
        integer :: N, i
        N = size(r, dim=1)
        msd = 0.
        do i=1,N
            rij = r(i,:) - rold(i,:)
            call pbc(rij,L)
            msd = msd + sum(rij**2)
        end do
        msd = msd/N
        end subroutine


        ! Test function to print a variable array (r,v,f,...)
        subroutine test(var)
        implicit none
        real, intent(in), dimension(:,:) :: var
        integer :: i

        do i=1,size(var,dim=1)
            print*, var(i,:)
        end do

        end subroutine test


end program main
```