Problem Set 5

Exercises for the lecture Fundamentals of Simulation Methods, WS 2020

Prof. Dr. Frauke Gräter, Prof. Dr. Friedrich Röpke
Tutors: Benedikt Rennekamp (benedikt.rennekamp@h-its.org), Fabian Kutzki (fabian.kutzki@h-its.org), Giovanni Leidi (giovanni.leidi@h-its.org), Siddhant Deshmukh (sdeshmukh@lsw.uni-heidelberg.de)
Offices: Heidelberg Institute for Theoretical Studies, Schloss-Wolfsbrunnenweg 35

Hand in until Wednesday, 09.12., 23:59
Tutorials on Friday, 11.12.
(Group 1, Giovanni Leidi 09:15)
(Group 2, Benedikt Rennekamp 09:15)
(Group 3, Siddhant Deshmukh 11:15)
(Group 4, Fabian Kutzki 14:15)

A simple Molecular Dynamics code

In this exercise, we construct a simple molecular dynamics code. We will provide two different versions of the same code template: `md.c` *and* `md.py`*. As a reference, we will also provide a* `postprocessing.py` *script to plot results (to interface with the Python code). Note that for this exercise, computational speed is critical. In this regard, C has usually a strong advantage. We tried to write the Python template using fast array arithmetic instead of explicit loops, so you can also try that. Even then, you might experience longer waiting times if you use Python.*

We want to simulate a simple system of $N = (N_{1d})^3$ argon atoms, interacting with a Lennard-Jones potential,

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

meaning that the total potential energy is given as

$$E_{\text{pot}} = \frac{1}{2} \sum_{i,j}^{N} V(|\vec{r}_i - \vec{r}_j|). \tag{2}$$

For argon, we will use the parameters

$$\sigma = 3.4 \times 10^{-10}\,\text{m}, \tag{3}$$
$$\epsilon = k_B \times 120\,\text{K} = 1.65 \times 10^{-21}\,\text{J}, \tag{4}$$
$$m = 6.69 \times 10^{-26}\,\text{kg}, \tag{5}$$

where σ and ϵ characterize the potential, and m is the mass of each atom.
Based on the template, you will write a computer code that integrates the equations of motion of N argon atoms, placed into a cubical box of side-length L with periodic boundary conditions. For this, solve the following problems:

1. Preparatory work [8 pt]

(a) Express all length units in terms of σ, all energies in terms of ϵ, and all masses in terms of m. In other words, introduce dimensionless distances

$$\vec{r}' = \frac{\vec{r}}{\sigma}, \tag{6}$$

dimensionless energies $E' = E/\epsilon$, etc., and rewrite all relevant equations in terms of the dimensionless quantities. What is a suitable quantity to scale the velocities?

(b) Write a function in the code template that sets up N_{1d} particles per dimension on a regular grid in a periodic box of size L (see code templates `md.c` and `md.py`). We adopt a mean particle spacing $\overline{d} = 5.0\,\sigma$ (implying that $L' = 5\,N_{1d}$ in the scaled length units). For the initial velocities, assume that we prescribe a certain kinetic temperature T in Kelvin, from which we can compute a one-dimensional velocity dispersion as

$$\sigma = \sqrt{\frac{k_B T}{m}}. \tag{7}$$

Scale this velocity dispersion to internal dimensionless units, yielding σ'. Now draw three random numbers $(\tilde{v}_x, \tilde{v}_y, \tilde{v}_z)$ for every atom from a Gaussian distribution with zero mean and a dispersion of unity, and scale them with σ' to get the initial velocities $\vec{v}' = \sigma' \tilde{\vec{v}}$. This means your initial velocities will then correspond to a Maxwellian with temperature T.

(c) Now write a function that calculates the acceleration $\vec{a}'_i$ of each particle, in the dimensionless units used by your code. For simplicity, sum over all distinct other particles in the box and always consider the nearest periodic image for each pair. Use a (quite large) cut-off radius for the potential equal to $r_{cut} = 5.0\,\sigma$, i.e. set the potential to zero for distances larger than r_{cut}.

Note that the cut-off radius does not affect the simulation in the provided code. In a real simulation, the cut-off radius is used – how would you have to change the setup of the simulation to make use of this quantity?
(For the Python users)
Here's a simple example of how you can calculate the acceleration in case of a simple potential $V(r) = r^2$:

```
def get_acceleration(self):

    for i in range(self.N):

      dx,dy,dz = dx_vec(i,self)

      r2 = (dx**2+dy**2+dz**2)
      self.F[:,:] = 0.0
```

```
k_int = (r2>0.) & (r2<self.r_cut**2)
r     = r2[k_int]**0.5
self.F[k_int,i_x] = -2.*r*(dx[k_int]/r)
self.F[k_int,i_y] = -2.*r*(dy[k_int]/r)
self.F[k_int,i_z] = -2.*r*(dz[k_int]/r)
self.a[i,:] = numpy.sum(self.F,axis=0)
```

2. Implementation and application [12 pt]

(a) Use the Leapfrog time integration scheme to advance the particles. To this end, prepare a function that 'kicks' the particles with their stored accelerations for a given time interval Δt. Also, produce a function that 'drifts' the particles with constant velocity over a given time interval Δt. After the particles have been moved, map them back periodically into the principal box in case they have left it.

(For the Python users)
You can take inspiration from the `rk1` implementation:

```
def do_rk1(self):

  #get acceleration
  self.get_acceleration()

  #get residuals
  self.R[:,i_x:i_z+1] = self.vel[:,:]*dt
  self.R[:,i_u:i_w+1] = self.a[:,:]*dt

  #evolve coordinates
  self.evolve_coords()

  #evolve velocities
  self.vel[:,:]       = self.vel[:,:] + self.R[:,i_u:i_w+1]
```

(b) In the force calculation, add a computation of the total potential seen by each particle due to its neighbors. Also, write a routine that computes the total kinetic energy and total potential energy of the system, as well as the instantaneous kinetic temperature. Call this function at the beginning of each time step, and output the mean kinetic energy per particle, mean potential energy per particle, and kinetic temperature to a file.

(Fot the Python users) The class `particles` already has an attribute for the potential energy (`self.V`). You just have to add the potential to this variable for each particle:

```
def get_acceleration(self):
```

```
self.V = 0.

for i in range(self.N):

    ....
    V = ....
    #remember that variable V here is an array
    #while self.V is a scalar!
    self.V += ...
```

(c) Run your code with a timestep $\Delta t' = 0.01$ in internal units (corresponding to $\Delta t = \sigma(m/\epsilon)^{1/2}\Delta t'$) for 20,000 steps using $N_{1d} = 5$ (i.e. $N = 125$ atoms) and $T_{init} = 80\,\mathrm{K}$. Calculate the autocorrelation of the absolute values of the velocities. Confirm that the total energy is conserved well by plotting the time evolution of the relative error.

The running time for the Python code should be on the order of $20 - 25$ minutes and the disk space is roughly 10 MB. Keep the output frequency equal to 10.

(d) Run your molecular dynamics simulations for the target temperature 400 K and calculate the autocorrelation of the absolute values of the velocities. How does this compare to the autocorrelation computed at 80 K? How long does it take for the autocorrelation to approach $1/e$ in each case? Explain the results.

Estimate from the results *for the last* 10,000 *steps* (to reduce the influence of the initial transient phase) the molar heat capacity at fixed volume, C_v, at the temperature $T \simeq 400\,\mathrm{K}$ (and the given density). Compare this result with the heat capacity of $C_v = \frac{3}{2}R$ expected for a monoatomic ideal gas, where R is the gas constant. Interpret your result.

Compare the time evolutions of the mean kinetic temperature (defined as the temperature derived from the mean kinetic energy per particle).

For Python users: post-processing, you can use the 'postprocessing.py' file for time evolutions, distributions and scatter plots.