Project B
Lennard-Jones System

Computer Modelling 2016-17

Due: 5pm Thursday, Week 9, Semester 2

1 Aims

In this project, you will write code to describe N-body systems interacting through the Lennard-Jones pair potential. You will use this to simulate, using periodic boundary conditions, a Lennard-Jones solid, fluid, and gas, and investigate equilibrium properties of the simulated systems.

2 Tasks

2.1 Simulation Code

Your code builds on the `Particle3D` class you wrote in the previous exercise. It should

- Read simulation parameters from an input file,
- Set up a list of `Particle3D` objects to hold the particles,
- Initialise particle positions and velocities by interfacing with the routines provided in `MDUtilities.py`,
- Simulate the evolution of the system using the velocity Verlet time integration algorithm, while obeying the minimum image convention and periodic boundary conditions,
- Write a trajectory file for the simulation that can be visualised using VMD.

Use this code to simulate Lennard-Jones systems at various densities and temperatures. Detailed suggestions on how to implement these objectives are given in Section 3 below. As a rough indication, this part of the code should be finished after ILW.

2.2 Molecular Dynamics Observables

Your code should include functionality to analyse the simulation in terms of equilibrium particle properties. The following quantities should be determined by your code:

- Fluctuations of the total energy of the system.
- Particle radial distribution function.
- Particle root mean square displacement.

Again, detailed suggestions on how to implement these objectives are given in Section 3 below.

3 Detailed Instructions

You are now going to generalise your simulation program so that you can simulate an arbitrary number of particles. You are also going to modify the force calculation in our code to simulate an atomic system rather than a gravitational-based system. This is often known as molecular dynamics (MD). One of the major differences between gravitational-based simulations and molecular dynamics is that, often, in an atomic-scale simulation we are simulating a periodically repeating system. We first need to modify our position-based routines to take account of this periodicity.

The largest cost in a simulation of this type is in computing all the pairwise interactions between particles. Typing all the $N(N-1)$ ordered particle pairings to compute the force and energy is clearly going to get tedious even for just three particles (where there would be 6 interactions). Ponder the cost of this operation for an entire protein or for a galaxy of objects where the number of particles required could be on the order of millions. We will use lists and loops to repeat the force calculation for all pairs of particles rather than try to write each interaction individually.

The instructions below are a suggestion of how to gradually build your project code by incrementally adding complexity and constantly testing it. You do not have to follow this suggested route, but beware that TA's and lecturers can best help you with issues and debugging if you do not stray too far from it. *Do not attempt to write the entire code in one go. Good on you if you can, but very likely you will introduce a lot of bugs that will be very hard to find.*

3.1 The Lennard-Jones Interaction Potential

In the previous checkpoint we considered a single particle orbiting a massive point located at the origin. Now we need to generalise the force vector and energy calculation routines so they can calculate the force vector arising from the interaction between two `Particle3D` objects and the potential energy from the interaction between two `Particle3D` objects.

We will simulate an ideal gas using one of the most common approximations to the pairwise interaction between non-polar particles – the Lennard-Jones (LJ) potential. The form of the potential for idealised particles is:

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

For particles of mass m we can introduce reduced units for length, $r^* = r/\sigma$, energy, $E^* = E/\epsilon$, and mass, $m^* = 1$. From those, we can derive units of temperature, $T^* = \epsilon/k_B$, and time, $t^* = \sigma\sqrt{m/\epsilon}$. Reduced units allow us to neglect potentially very small or very large prefactors common in atomic systems (Argon model parameters, for instance, are $\epsilon/k_B = 119.8\text{K}$, $\sigma = 3.405 \times 10^{-10}\text{m}$, $m = 0.03994\text{kg/mol}$). In reduced units, the LJ potential reads

$$U(\boldsymbol{r}) = 4 \left[\frac{1}{r^{12}} - \frac{1}{r^{6}} \right] \tag{2}$$

with the force given by:

$$\boldsymbol{F}_1(\boldsymbol{r}_{12}) = -\nabla_2 U(\boldsymbol{r}_{12}) = 48 \left[\frac{1}{r^{14}} - \frac{1}{2r^{8}} \right] (\boldsymbol{r}_1 - \boldsymbol{r}_2) \tag{3}$$

You notice that the LJ force falls off very rapidly at large distances. To avoid unnecessary calculations, the force and potential energy of a pair of LJ particles can be set to zero, if their separations is larger than a certain cutoff radius. In reduced units, cutoff radii $r_c = 2.5 \cdots 3.5$ are commonly used.

Create two methods in that take two `Particle3D` objects as their arguments and return the force vector (for the force method) or the potential energy (from the energy method), and take into account a user-defined cutoff distance.

3.2 Particle List Methods

You should now add a set of methods that can operate on lists of particles. Using these you can easily extend simulations to an arbitrary number of particles. You should add functionality that:

- Updates the velocity for all particles in a list. Remember the total force vector for a particle is the sum of the forces due to all the other particles. You should formulate an algorithm to compute this sum for a single particle and then consider how you might modify this so it operates on each particle in turn.
- Updates the position for all the particles in an array.
- Calculates the total energy due to atom pair interactions and particle kinetic energies. Take care here to avoid double-counting interactions.

3.3 Visualization

Trajectory files are often used within scientific modelling to represent a set of points (in three-dimensional Cartesian space) at a number of different steps within an ordered time series. Once the data has been stored in a trajectory file we can then visualise it in a number of ways, for example by animating it or by plotting all the points simultaneously.

VMD is a standard tool that is used for visualising trajectories in a variety of scientific fields ranging from biology to astrophysics.

The format of the trajectory file for VMD (for a system containing two points) looks like this:

```
2
Point = 1
s11 x11 y11 z11
s21 x21 y21 z21
2
Point = 2
s12 x12 y12 z12
s22 x22 y22 z22
⋮
2
Point = m
s1m x1m y1m z1m
s2m x2m y2m z2m
```

You can see that the file consists of `m` repeating units (where `m` is the number of steps in the trajectory) and that each unit consists of two header lines: the first specifies the number of points to plot (this should be the same for each unit) and a title line (in the example above it specifies the point number). Following the header lines are the lines specifying the Cartesian coordinates for the particles (one for each point), which consist of: the particle label (some text) and the x, y, and z coordinates for that particle at this trajectory entry.

You should already have a `__str__()` method in your `Particle3D` class that produces a String in the correct format for a VMD trajectory file. Write a method to write out an entry for a single timestep to a VMD trajectory file. While you can write out trajectory information at every single timestep, often this leads to very large trajectory files. Think about how you can produce file output every n-th timestep, and whether n can be passed to your code together with other simulation parameters.

You should use VMD to visualise the trajectories for the simulations you run. You can view a trajectory in VMD using something like:

```
[user@cplab001 ~]$ vmd myTrajFile.xyz
```

You should experiment with visual representations in VMD (Graphics → Representations menu item) to find different ways of representing the time series. In particular:

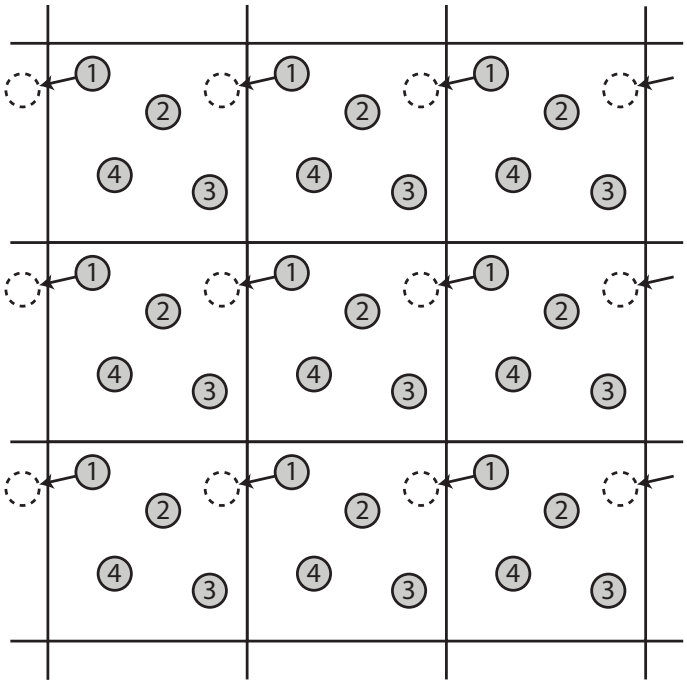

Figure 1: 2D Periodic Boundary Conditions: Particle 1 moves across the simulation cell boundary and re-enters from the opposite end.

- Use the Points drawing method to visualise the trajectories as particles moving in time. Can you speed up and slow down the animation?
- Use the Points drawing method to create a static representation of the entire trajectory. You will need to use the "Trajectory" tab of the representations window to set the trajectory range (lower and upper limit of steps to display) and stride (increment between displayed steps) to generate a meaningful representation.

You could also experiment with the `PBCTools` plugin for VMD (http://www.ks.uiuc.edu/Research/vmd/plugins/pbctools/), which allows you to relay information about periodic boundary conditions to VMD (which is not included in `.xyz` files). For instance, you can set box size and display its edges by typing

```
vmd > pbc set {5.0 5.0 5.0} -all
vmd > pbc box
```

3.4 Periodic Boundary Conditions

To simulate the condensed phase we need to simulate the bulk material – this would lead to an unmanageable number of simulated particles (1 mole = 6.023×10^{23}) even for the largest supercomputer. An approach used to overcome this limitation is known

as Periodic Boundary Conditions (PBC). We introduce a simulation box containing a finite number of particles set up such that a particle that moves out of the box re-enters from the opposite side. Formally, this can be thought of as having an infinite number of repeating simulation boxes placed in periodically repeating manner – when a particle leaves the box its image re-enters the box from one of the copies (see Figure 1 for a diagram of 2-dimensional PBC).

Although this algorithm makes it possible to simulate the bulk condensed phase it does introduce some complications when updating the positions of particles and computing the separation between particles. At the point at which you update the particle positions you need to account for particles that may have moved past the box boundaries. You should produce an algorithm that modifies the updated position to the correct value if it ends up outwith the box boundaries. The origin is at the bottom left-hand corner of the box. Remember that particles can move out of the box in two directions – your algorithm should be able to cope with either situation.

3.5 Minimum Image Convention

PBC's also mean that when computing the separation between two particles we must take account of the fact that a particle's image may be closer than the actual particle in the box. By convention, we compute the interaction based on the closest image of the particle – this is called the *minimum image convention*.

Formulate and implement an algorithm to use the minimum image convention whenever distances between particles are used in your code. Note that if a particle is more than half a box length away in a particular dimension then an image will be closer.

3.6 Initial Conditions

You are provided an auxiliary python file, `MDUtilities.py`, that provides functionality to set up appropriate initial conditions for particle positions and velocities. The routine `setInitialPositions()` takes two parameters: `rho`, the reduced particle density, and `particles`, a list of `Particle3D` objects. It returns the simulation box as Numpy array. It will also place the particles on a face-centered cubic lattice inside the box. This is the crystal structure of, e.g., solid Argon. The method will print a warning if the number of particles is not consistent with a fully filled lattice inside the box, i.e., if it can not be written as $4N^3$. This is fine for simulations of fluids and gases, but if you want to simulate a solid, you should consider creating a fully filled box.

Initial velocities are set by the method `setInitialVelocities()`. It also takes two parameters: `tem[`, the reduced temperature, and `particles`, a list of `Particle3D` objects. It will set the particle velocities $\boldsymbol{v}_i$ to random vectors, but with an overall normalization to a Boltzmann-Maxwell distribution according to the reduced temperature.

3.7 N-body Simulation Code

Write a program to simulate many-body systems. This can be based on the tester code you wrote for Exercise 3. It should contain all the functionality requested in section 2.1.

As we may now also have many particles in a simulation it makes sense to read in the simulation parameters (number of particles, number of simulation steps, time step, reduced temperature and density, LJ cutoff distance) from a file. This means you program should take two command line arguments at this stage, i.e.:

```
[user@cplab001 ~]$ python ParticleManyBody.py param.input
   traj.xyz
```

where `particle.input` is the file containing the particle details; and `traj.xyz` is the name of the output file containing the trajectory (this must have a `.xyz` extension to work correctly with VMD, see section 3.3).

3.8 LJ Simulations

Test that your code works by running simulations of LJ systems. Typical conditions that will result in a solid (fluid, gas) phase are reduced densities $\rho = 1.0(0.5, 0.1)$ and temperatures $T = 0.1(0.5, 5.0)$, respectively. If you study the resulting trajectories in VMD, you should see clear differences between the solid phase (where all particles move about their lattice positions) and the other phases (where particles move freely through the box). Check that PBC are working correctly, by visually tracking particles leaving and re-entering the simulation box.

To quantify the particle properties, equilibrium properties on particle behaviour and correlations are useful. These will in particular help to distinguish fluid-like from gas-like behaviour.

3.9 Adding Observables

You should now add functionality to your code to obtain system-wide information about particles and their correlations. In particular, your code should produce output on

- Total, potental, and kinetic energy of the entire system,
- Average particle mean squared displacements (MSD) as function of time,
- Particle radial distribution functions (RDF).

Total energy in the system should be conserved – there are no dissipative forces or couplings to external heat sources/sinks in these simulations. However, there will be an initial drift in the total energy (the 'equilibration phase'), and the distribution of the total energy between kinetic and potential energy will fluctuate. Note that your system needs to be in equilibrium for unbiased measurements of the observables below.

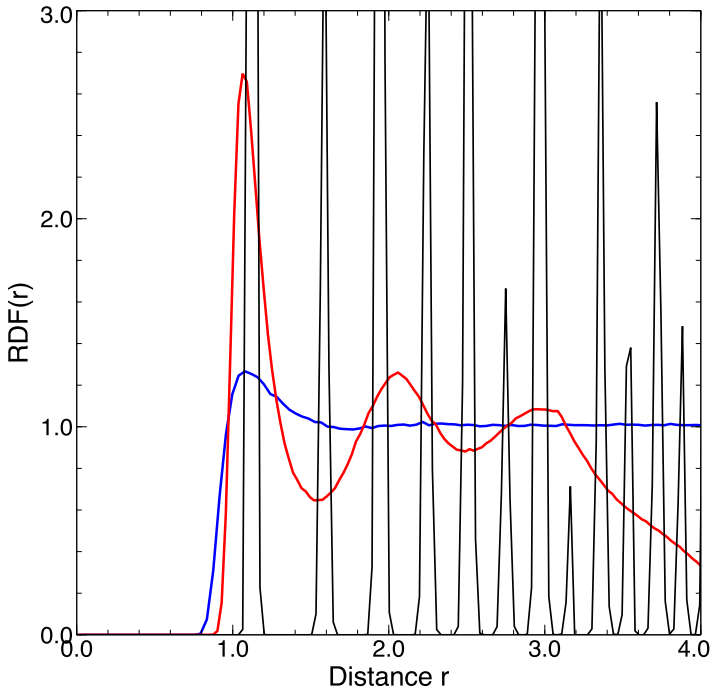

Figure 2: RDF of a LJ solid (black), fluid (red), and gas (blue). Note the erroneous drop-off of the fluid-RDF at large distances, where data collection extends beyond half the box size.

3.9.1 Mean Squared Displacement

The MSD is a measure of how a particle deviates over time from a reference position.

$$\mathrm{MSD}(t) \equiv \langle |\boldsymbol{r}(t) - \boldsymbol{r}_0|^2 \rangle = \frac{1}{N} \sum_i |\boldsymbol{r}_i(t) - \boldsymbol{r}_{i0}|^2 \tag{4}$$

If you simulate a fully filled box, you can choose the reference positions $\boldsymbol{r}_{i0}$ to be the initial positions of the particles on the face-centered cubic lattice. You should write a method that determines and averages the MSD over all particles (adhering to the minimum image convention) at regular intervals.

In a solid, the MSD will be a constant. In a simple diffusive system and at long times, on the other hand, the MSD is proportional to the diffusion constant D:

$$\mathrm{MSD}(t) = 6Dt \tag{5}$$

3.9.2 Radial Distribution Function

The RDF is a measure of the probability to find a particle at a given distance to a reference particle. It is a pair-correlation function that reveals information about how

ordered the simulated system is. It is defined as

$$\mathrm{RDF}(r) \equiv \frac{1}{N\rho}\langle\sum_{i,j}\delta(r_{ij}-r)\rangle \tag{6}$$

This average is usually determined by calculating all distances r_{ij} between pairs of particles (adhering to the minimum image convention) and binning them into a histogram. You should write a method that performs this data collection at regular intervals. You should then normalize the histogram data, noting that the average number of particles expected at a distance r is $4\pi r^2\rho dr$, where ρ is the number density and dr the spacing between histogram bins. You should write a second method that performs this normalization.

In a solid, the RDF will consist of a series of sharp peaks, due to the highly periodic setup. In a liquid, local order (such as fixed coordination numbers) will lead to a well-structured RDF, while in a gas it should be almost constant - see Figure 2.

4 Submission

Package the subdirectory that contains `Particle3D.py`, your simulation program, input files, and the resulting trajectory file for a representative simulation run. For instance, if your subdirectory is called `project`, you can use the `tar` and `gzip` commands by running:

```
[user@cplab001 ~]$ tar cvf project.tar project
[user@cplab001 ~]$ gzip project.tar
```

In the documentation of your simulation program, describe how to run it and what the format and units of the input and output files are.

We will not accept submissions that are larger than 50MB total. Hence, make informed choices on how often to print trajectory information to file (is every single time step necessary?), how many digits are relevant (use formatted outputs), and restrict the overall simulation time to reasonable values (what is a reasonable time to accumulate MSD and RDF data?). Submit the compressed package (e.g., `project.tar.gz`) through the course LEARN page, by **5pm on Thursday, week 9 of semester 2**. Only one submission per group is necessary.

5 Marking Scheme

This assignment counts for 20% of your total course mark.

1. Code compiles and works with inputs provided [4]
2. Input and output formats sensible and appropriate [4]

3. Code has all required functionality: LJ potential, list methods, file I/O, periodic boundary conditions, minimum image convention, velocity Verlet integrator, total energy tracking, mean squared displacements, radial distribution function. [20]

4. Code layout, naming conventions, and comments are clear and logical [12]

Total: 40 marks.