Project B: Lennard-Jones system

Computer Modelling

Due: 16:00 Thursday, Week 8, Semester 2

Aims

In this project, you will write code to describe an N-body system interacting through the Lennard-Jones pair potential. The code shall, eventually, be able to simulate the solid, liquid, and gas phase of Argon using periodic boundary conditions. This will be used to further investigate the equilibrium properties of all these states.

1. Background

1.1. The Lennard-Jones Interaction Potential

In the previous exercise we considered a molecule with atoms interacting through the Morse potential – which is appropriate to describe covalently bound systems. In this project, instead, we will deal with a model system interacting through the Lennard-Jones (LJ) potential. The Lennard–Jones potential is a model for weakly-interacting neutral systems. Its attractive part is the leading $\mathcal{O}\left(r^{-6}\right)$ term of van der Waals interactions. The repulsive $\mathcal{O}\left(r^{-12}\right)$ term is a numerically simple way to model core–core repulsion. The potential energy of two particles at $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ may be written as

$$U(\boldsymbol{r}_1, \boldsymbol{r}_2) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{1}$$

where $r = |\boldsymbol{r}_1 - \boldsymbol{r}_2|$. The two free parameters are ϵ, an energy, and σ, a length. It can be seen that the equilibrium distance of the Lennard–Jones potential is $r_e = \sqrt[6]{2}\,\sigma$, and that its classical binding energy is exactly ϵ.

While eq. 1 is valid for homogeneous systems (having only one kind of atom), it is possible to modify it for heterogeneous systems by realising that each *pair* interaction (e.g. Ne–Ne, or Ne–Ar) has its set of parameters.

The LJ interaction falls off very rapidly at large distances. In an attempt to save computing time, it is typical to set forces to zero beyond a certain cutoff radius[1] r_c, and $V(r > r_c) = V(r_c)$.

In this project, you may implement a cut-off radius of 3.5σ.

1.2. Units

The choice of units to use in a simulation is important - the simulation can be more accurate and easier to understand if the units give values which within a few orders of magnitude of one. In atomic simulations, using metres or seconds would lead to very awkward units. You have already dealt with problem-adapted units in exercise 3.

In a system where there is only a single type of atom, the in an atomic system we can define a set of units that makes the LJ potential simpler. The potential suggests we measure:

- Distances in units of σ,
- energy in units of ϵ
- mass such that the atom being used has mass 1.

These three choices set the value of many other unites. Concretely, the above definition sets the unit of time is $[t] = \sigma\sqrt{m/\epsilon}$, and we can measure temperature in units of energy, converting to Kelvin via the Boltzmann constant k_b when needed (and vice versa).

In these units, we have:

$$U(\boldsymbol{r}_1, \boldsymbol{r}_2) = 4\left[\frac{1}{r^{12}} - \frac{1}{r^6}\right], \tag{2}$$

with the force $\boldsymbol{F}_1$ on the particle at $\boldsymbol{r}_1$ given by:

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

and $\boldsymbol{F}_2 = -\boldsymbol{F}_1$.

Adopt these units, and thus this form of the potential and force, in this project.

[1]Just like with the timestep, whether a cutoff radius is large enough can only really be determined with careful testing. You will not be required to perform such convergence tests in this project.

1.3. N-Body simulations

When we upgrade simulations from two bodies (like the ones we simulated last semester) to arbitrary numbers, there are several important changes we have to understand.

First the force on any one particle is now the sum of the force on it from all the other particles:

$$F_i = \sum_j F_{ij} = -48 \sum_j \left[r^{-14} - \frac{1}{2} r^{-8} \right] (\boldsymbol{r}_i - \boldsymbol{r}_j) \tag{4}$$

where $r = |\boldsymbol{r_i} - \boldsymbol{r_j}|$

We have to be more careful with the potential energy, and avoid double counting, as the potential on A due to B should not be added to the potential of B due to A - this is the same energy. There are a few ways of implementing this. You can either do the complete sum and then divide by two, or by avoiding the double count in the first place:

$$U = \frac{1}{2} \sum_i \sum_{j \neq i} U_{ij} \tag{5}$$

$$= \sum_i \sum_{j > i} U_{ij} \tag{6}$$

Note that in each case we must avoid the potential of a particle with itself.

1.4. Boundary conditions

Modelling condensed matter requires simulating bulk materials, which have $\sim 10^{23}$ particles. This is far too expensive even for the largest supercomputers, so we *need* a shortcut.

1.4.1. PBC: Periodic Boundary Conditions

As you know from Physics of Fields and Matter, crystals are solids with perfect translational symmetry. We will assume that our system also has translational symmetry: we introduce a simulation box of size L containing a finite number of particles, and we will enforce *periodic boundary conditions* (PBC) such that each particle in the box has a mirror image in every other box.

Any particle leaving the simulation box on one face reappears through the opposite face (see Fig. 1a). Now, every particle is surrounded by infinitely many others, in every direction.

In practice we can implement this by moving every particle so that it always stays inside the box, by applying:

$$\mathrm{PBC}(\boldsymbol{x}) = \boldsymbol{x} \bmod \boldsymbol{L} \tag{7}$$

where the box size $\boldsymbol{L}$ can either be a vector or a scalar if the box is a cube.

1.4.2. MIC: Minimum Image Convention

PBC's imply any particle is surrounded by infinitely many copies of each of the others, but short-range forces like LJ allow us to ignore all interactions beyond a *cutoff radius* and just select a single nearest neighbour for each particle.

The *minimum image convention* (MIC) is a way of choosing one nearest neighbour: of all the periodic replica images of each particle, we will only consider interactions by the closest. The nearest neighbour is not necessarily the one in the same cell - as panel (b) of Fig. 1b, where the closest copy of P3 to P1 is to the top left of it, in a different cell.

MIC takes a particle location in the unit cell and returns the nearest neighbour location.

$$\mathrm{MIC}(\boldsymbol{r_{12}}) = ((\boldsymbol{r_{12}} + \boldsymbol{L}/2) \bmod \boldsymbol{L}) - \boldsymbol{L}/2 \tag{8}$$

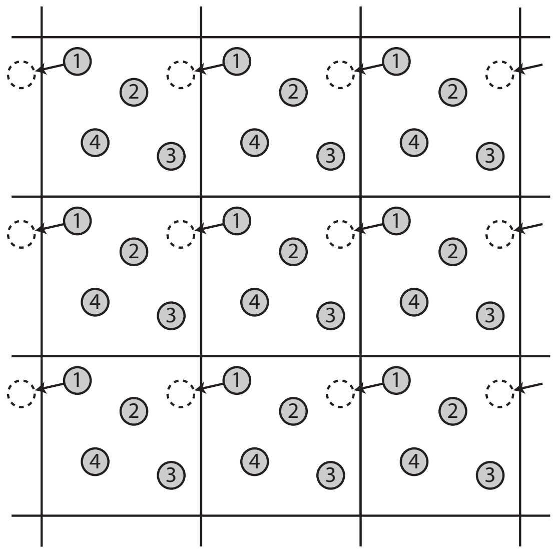
where $\boldsymbol{r_{12}} = \boldsymbol{r_1} - \boldsymbol{r_2}$. We use this function each time the force or potential between two particles is calculated.

1.5. Trajectory Files

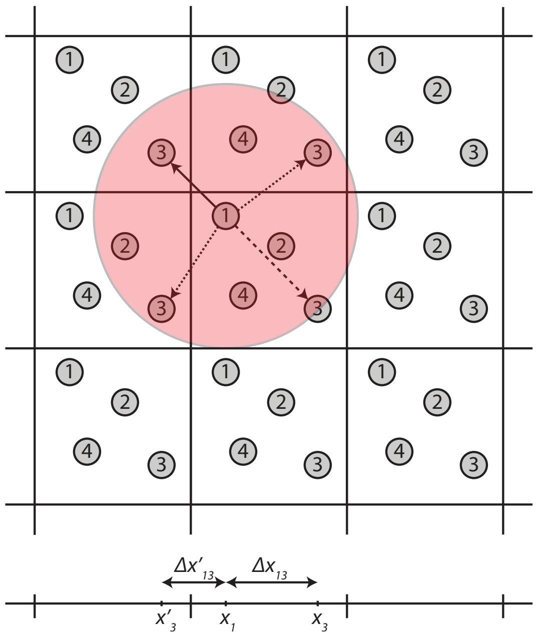
Trajectory files are often used within scientific modelling to represent a set of points (in three-dimensional Cartesian space) at a number of different steps within an ordered time series. Once the data has been stored in a trajectory file we can then visualise it in a number of ways, for example by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

The XYZ format (for e.g. two particles) looks like this:

```
2
Point = 1
s1 x11 y11 z11
s2 x21 y21 z21
2
Point = 2
s1 x12 y12 z12
s2 x22 y22 z22
```



(a) 2D PBC: Particle 1 moves across the simulation cell boundary and re-enters from the opposite end.



(b) 2D MIC: Particle 1's interaction with images of particle 3. Notice the distance within the box is longer than the others.

```
⋮
2
Point = m
s1 x1m y1m z1m
s2 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 comment line (in the example above it specifies the point number). Following that, the file specifies the label and the Cartesian coordinates each particle at this trajectory entry.

1.6. Observables

1.6.1. Mean Squared Displacement

The MSD is a measure of how far, at a time t, particles have moved on average from their reference position:

$$\mathrm{MSD}(t) \equiv \left\langle |\boldsymbol{r}(t) - \boldsymbol{r}_0|^2 \right\rangle = \frac{1}{N} \sum_i |\boldsymbol{r}_i(t) - \boldsymbol{r}_i(0)|^2 \tag{9}$$

The reference positions $\boldsymbol{r}_i(0)$ are the initial positions of the particles in the crystal.

The MSD shows radically different behaviour for solids and fluids. Atoms in a solid oscillate about the equilibrium positions. At intermediate temperatures, these oscillations are typically $\sim 10\%$ of the bond length. Therefore, after the equilibration period, the MSD will fluctuate around a constant value. In a simple diffusive system, on the other hand, we would expect random-walker behaviour: the MSD should be roughly linear. Figure 2 shows the MSD for the LJ system at different conditions.

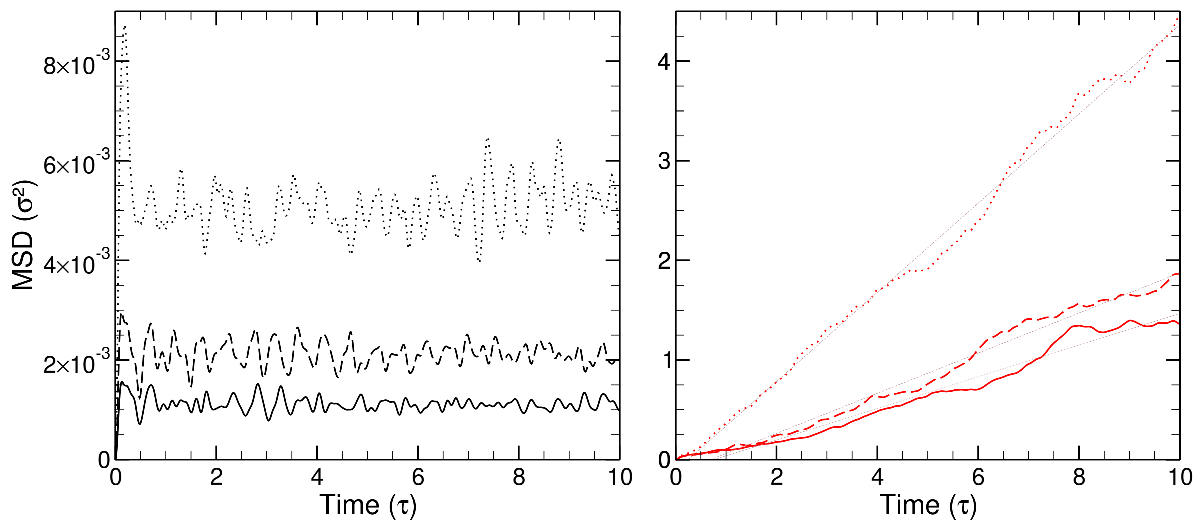


Figure 2: MSD of a LJ solid (left, black) and a fluid (right, red), each at three different temperatures. Note the different orders of magnitude. Straight lines show the average linear behaviour of the liquid MSD.

The slope of the MSD in a liquid is proportional to the diffusion coefficient D:

$$\mathrm{MSD}(t) = 6Dt \tag{10}$$

1.6.2. Radial Distribution Function

The RDF, $g(r)$, is a measure of the probability to find a particle at a given distance to a reference particle. It encodes information about the microscopic structure of a system, and how ordered it is. theoretically, the RDF is defined as

$$g(r) \equiv \frac{1}{N\rho_0(r)} \left\langle \sum_{i,j} \delta(r_{ij} - r) \right\rangle, \tag{11}$$

where N is the number of particles in the system, $\rho_0(r)$ represents the expected value of RDF for a perfectly homogeneous system, and the brackets denote a time average. For a system with particle density $\rho^\star$,:

$$\rho_0(r) = 4\pi\rho^\star r^2 \mathrm{d}r. \tag{12}$$

In practice, RDFs are estimated using a histogram of the separations r_{ij} between each pair of particles (using the MIC as described above), but the histogram must be normalized by dividing by the r^2 as in this equation.

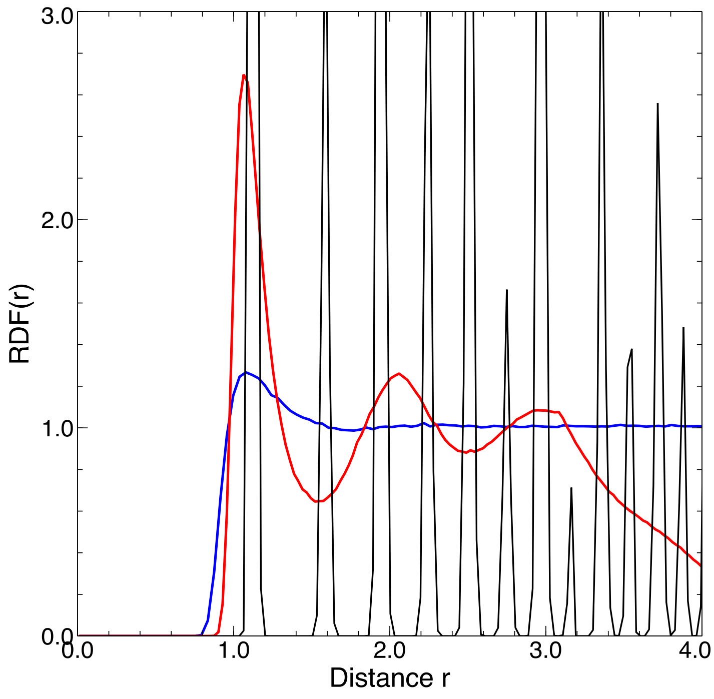


Figure 3: RDF of a LJ solid (black), fluid (red), and gas (blue). Note the erroneous drop-off of the fluid-RDF at large distances, where data collection extends beyond half the box size.

In a solid, the RDF will consist of a series of sharp peaks, due to the highly periodic setup. In a liquid, local order will lead to a well-structured RDF, while in a gas it should be almost constant - see Figure 3.

2. Coding Tasks

In this project, you will generalise your simulation program from Exercise 3 so that you can simulate an arbitrary number of classical point particles interacting via the Lennard-Jones potential inside a periodic unit cell, and compute system observables.

It is better to **begin early, before pressure from other courses builds up**.

Your code should build on the `Particle3D` class,the Velocity Verlet integrator, and the PBC and MIC functions you wrote in semester 1. Check the feedback you got on that exercise on LEARN, and fix any issues identified.

Document your code as you go along using both docstrings for functions and inline # comments explaining what you are doing.

2.1. Simulation Code

Modify your code to use an arbitrary number particles in a list instead of just two. Represent the full system as a list of particles, where e.g. `particles[i]` is a Particle3D instance.

You will need code to:

- compute all the separations between each pair of particles, using MIC,
- compute all the LJ forces at once from this array, using the cut-off radius,
- update all the particle velocities at once,
- update all the particle positions at once, applying the PBC after they move
- compute the total energy.
- generate results as numpy arrays of the body positions and velocities, the time, and the energy across the simulation

Computing the pair separations $(\boldsymbol{r}_i - \boldsymbol{r}_j)$ every step is the most time-consuming element of an N-body simulation. That method should exploit `numpy`, as well as symmetries.

Typing all the $N(N-1)$ ordered particle pairings to compute the force and energy is tedious and inflexible. You should therefore automatically go through the particles using loops.

2.2. Basic Test

The equilibrium separation for a pair of particles under the LJ force is $r_0 = \sqrt[6]{2}$.

Test your code by simulating:

- two particles a distance $r_0 + 0.1$ apart in the x-direction,
- moving in opposite directions with speed $v_x = \pm 0.01$,
- both inside a unit cell of size 3,
- for 1000 time steps each of size 0.01.

You should find that each of the particles oscillates harmonically, and that energy is conserved within around 0.1%.

Now move the particles so they are still the same distance apart, but one is beyond the unit cell. You should find the same behaviour if your PBC and MIC are correct.

Finally, add 1.0 to the velocity of both of the particles. You should find they move linearly across the cell then jump back to the start of it when they reach the edge. Plot MIC($\boldsymbol{r_1} - \boldsymbol{r_2}$). You should find it oscillating with a small period regardless of this motion.

2.3. Visualisation

Modify your code so that it saves the complete trajectories of all the particles in a simulation to an XYZ-format file, as described above.

Your P3D `__str__()` method should already produce a string in the correct format following Exercise 2 feedback. Using it, now code a method that writes out a complete entry for a single time-step to a trajectory file.

Use either the program VMD (see Appendix), or if it doesn't work on your system, the script `plot_xyz.py` on LEARN to view an animation of the two-particle system: run `python plot_xyz.py --3d my_file.xyz` for a basic animation, or `python plot_xyz.py --help` for more options.

2.4. Initial conditions

We have provided a file `md_utils.py` which contains two functions to simulate an FCC cell[2]:

- `set_initial_positions(rho, particles)`, which takes a number density in particles $/\sigma^3$ and a list of particles. It modifies the positions of all of those particles, and also returns the box size to use.
- `set_initial_velocities(temp, particles)`, which takes a temperature in ϵ units and a list of particles. It modifies the velocity of all of those particles randomly according to the temperature.

The list of particles should contain the number of particles that you want to simulate, already created, but their initial positions and velocities can be anything, as they will be over-written. This code assumes that the particle attributes are called `position` and `velocity`. If you called them something else like `self.pos` or `self.x` then you will have to modify the utilities code to use your names.

When you are simulating liquids and gases you can use any number of particles. For solids, though, you should ensure the lattice is completely full by making the number of particles $4n^3$ for some integer n, e.g. 4, 32, or 108.

Modify your code to create a list of particles and then pass it to these two functions before using it in the integration. For now, use hard-coded values of the number, temperature, and density, $N = 32$. $\rho = 1.0$, and $T = 0.1$.

2.5. Full runs

We can now run the code with crystalline initial conditions.

2.5.1. Solid Argon

Run the code with 32 particles, $\rho = 1.0$, $T = 0.1$, $dt = 0.01$ (in reduced units) for 1000 steps.

Ensure energy is still conserved, and use VMD or `plot_xyz.py` with the `--3d` flag to animate the XYZ file. The particles should oscillate somewhat randomly around their initial starting positions, though note that because of PBC some may jump from one side of the box to the other.

[2]The crystal structure of most noble gases is fcc.

2.5.2. Gaseous Argon

Run the code with 30 particles, $\rho = 0.05$, $T = 1.0$, $dt = 0.005$ (in reduced units) for 2000 steps.

Again, ensure energy is still conserved, and view the simulation again. The particles should move far around the box, bouncing off each other occasionally.

2.6. User input

Add a user input system for selecting:

- the simulation parameters δt and N_{step}
- the number, temperature, and density of particles in reduced units,
- the name of output files.

It's up to you how you do this, but the more convenient it is for the user the better - having to manually type something out every time you want to run is sub-optimal.

2.7. Observables

Add functions to estimate these quantities:

- The kinetic, potential, and total energies of the system
- Particles' mean square displacement (MSD) as a function of time.
- Particles' average radial distribution function (RDF).

and save all of them to files for later analysis.

These functions should operate on the arrays you generated in section 2.1, rather than trying to do the calculation during the integration.

The total energy in the system should be conserved. Long term drifts are a sign of either a wrong integrator or a too-large timestep. However, there will be an initial drift in the kinetic & potential energies (the *"equilibration phase"*), followed by fluctuations.

Your MSD method should average the displacements (using the MIC) over all particles at regular intervals in the data. You should think about what is the maximum displacement you can usefully measure in this periodic simulation.

You can average the RDF over a number of time steps, with gaps between, starting after the system has settled down to equilibrium.

Make plots of the RDF and MSD for your solid and gas simulations, and try to find a temperature where the argon is a liquid.

2.8. Performance

Early optimization is the root of all evil. Make sure your code runs before trying this section.

Having said that, the typical errors slowing student codes down are computing expensive quantities over and over again, and improper usage of `numpy`. The worst offender is by far computing $\boldsymbol{r}_i - \boldsymbol{r}_j$ multiple times per step.

If needed, modify your code to only compute pair separations once per step. That alone should result in acceptable performance: a recent Intel Core i5 should run all the simulations quickly enough.

3. Submission

3.1. Anonymising

Your submitted code will be anonymised and then uploaded for a plagiarism check using `moss`.

To enable this, ensure none of your files include your exam number (e.g. B0000). Also ensure your names and matriculation numbers appear only on the headers of the python files, and not elsewhere.

3.2. What to submit

Collect in a directory:

- any required simulation input files
- your python file(s)
- a readme telling us how to run your code and what the format and units of the

input and output files are

- the XYZ trajectory file for a representative simulation rum

Submissions larger than 50 MB will not be accepted. Hence, make informed choices on how often to print positions 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.

3.3. Compressing your submission directory

Windows Right-click on the folder, and under the "Send-to" menu, select "Compressed (zipped) folder". Rename the file that is created to include your university ID.

Mac Ctrl-click on the folder and click "Compress name_of_folder". Rename the file that is created to include your university ID.

Linux Run this in a terminal (without the $):

```
$ tar -czvf project-YOUR-UNI-ID.tar.gz
   project_directory_name
```

and verify the command ran without errors. Check that all the files you want are included by running

```
$ tar -tf project-YOUR-UNI-ID.tar.gz
```

3.4. Submitting

Submit the compressed package through the course LEARN page, by **16:00 on Thursday, week 8 of semester 2**. Successful submissions are emailed a receipt.

4. Marking Scheme

This assignment counts for 25% of your total course mark.

1. Code compiles and works with inputs provided [**5**]
2. Input and output formats sensible and appropriate [**5**]
3. The simulation is physically correct [**10**]

4. Code has all required functionality to measure observables: list methods & `numpy`, correct file I/O, total energy tracking, MSD, and RDF.[**20**]

5. Code layout, naming conventions, and comments are clear and logical [**10**]

Total: 50 marks.

A. Plotting with VMD

Operating systems: VMD works well under Windows and Linux. It used to work well on Macs until Mojave (10.14), but Catalina (10.15) broke it. Running VMD on Catalina requires a special test build and heavy circumventing binary notarization, but can be done. Running VMD under Big Sur may not be doable at all.

Plotting trajectories with VMD in the CPLab is straightforward, provided one is aware of a few potential pitfalls:

1. **Order of magnitude:** VMD is designed with *molecules* in mind, and expects figures near unity (10^6 is fine). If the scale is very large, though, VMD may crash. The units we chose in this project should work.

2. **Visualization:** Particles are assigned to an element based on their labels, and then given a radius proportional to the covalent radius of that element. Do not expect to see anything if your trajectory coordinates are in km.

3. **File size:** It is possible to write a trajectory file of 108 particles for a million time steps, but such trajectory file would be unwieldy. 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 by opening the XYZ file you saved within it.

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

- 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
```