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. Tasks

1.1. Simulation Code

Your code should build on the `Particle3D` class and the Velocity Verlet integrator already written, as well as `pbc.py` from Ex. 1. Your final code should:

- Have user-selectable simulation parameters,
- set up a user-defined arbitrary number of particles...
- ... with positions and velocities initialised by the provided module `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 appropriate observables (e.g. kinetic energy) to file, and
- write an XYZ-compatible trajectory file for the simulation. Such file can be visualised, amongst others, with VMD.

Use this code to simulate a system closely resembling Argon at appropriate densities and temperatures to describe its solid, liquid, and gas phase. 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 CLW. *If your code is far behind schedule at that stage, get in touch with the teaching staff and TA's to help you!*

1.2. 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:

- Evolution of the total energy of the system as a function of time.
- Particles' mean square displacement (MSD) as a function of time.
- Particles' average radial distribution function (RDF).

Total energies, MSD, and RDF should be written to file for further analysis. Again, detailed suggestions on how to implement these objectives are given in Section 3 below.

2. Simulating the Bulk

Before detailing the coding approach to our Lennard-Jones system, we must first revisit Exercise 1 and discuss a key approximation. Modelling condensed matter requires simulating the bulk material, with $\sim 10^{23}$ particles. This is unmanageably expensive even for the largest supercomputers, so let us go smaller. On the other end, consider a 10^3 simple-cubic arrangement of *"atoms"*. Assuming surface effects are only 1-layer deep, as many as 488 atoms belong to the surface, unacceptably many for an already expensive problem. We *need* a shortcut. A shortcut you have already coded.

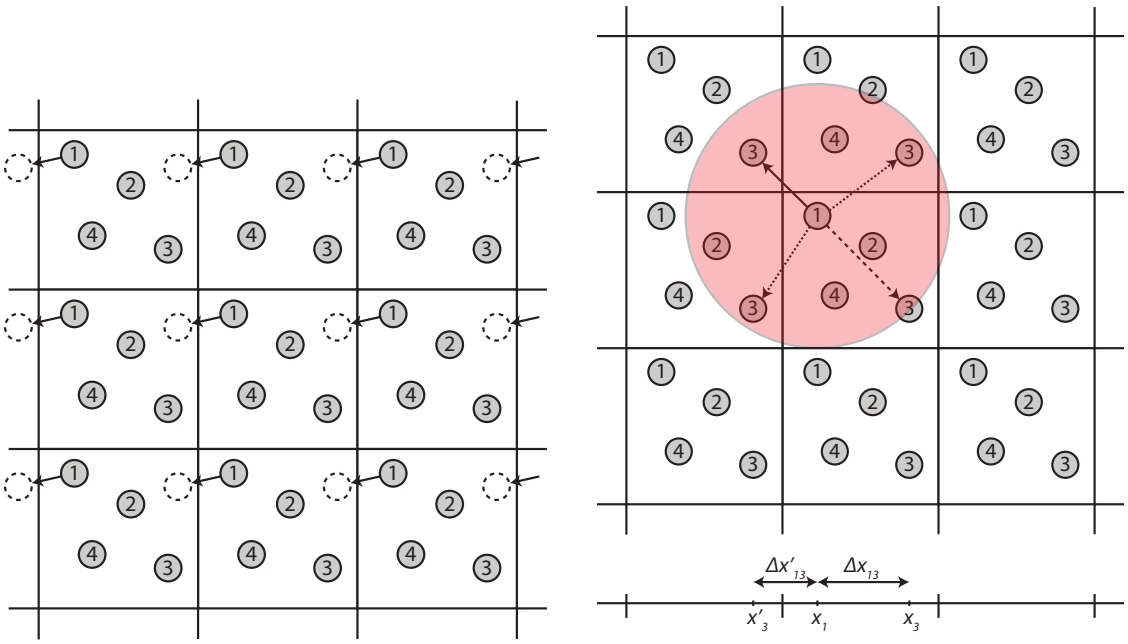
2.1. Periodic Boundary Conditions

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 (lattice vectors $\boldsymbol{a}_i$) containing a finite number of particles, and we will enforce *periodic boundary conditions* (PBC) such that a particle at $\boldsymbol{r}$ has a mirror image at $\boldsymbol{r} + \sum_i n_i \boldsymbol{a}_i$, for any integer n_i.

PBC effectively eliminate any surface effects, as any particle leaving the simulation box on one face reappears through the other (see Fig. 1a). Now, every particle is surrounded by infinitely many, in any direction. There is no surface[1]. Every particle is now bulk.

We can now simulate the bulk. We only need to be reasonably careful when updating particle positions, as we may need to insure particles stay inside the simulation box. In the next section we will discuss how to deal with interactions when enforcing PBC.

[1] Modelling surfaces and vacuums with PBC is possible, but an art on its own



(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.

Non examinable. Do note that enforcing PBC introduces finite size effects: Is particle i interacting with its mirror images appreciably? Is the motion of particle j affected by both i and its mirror images? Is that correct? Is my system overly prone to statistical flukes given its small number of atoms?

2.2. Summing Interactions: the Minimum Image Convention

PBC's imply any particle is surrounded by infinitely many. Our simulation cell may have N particles, but we now no longer have $N(N-1)/2$ interactions. Long-range interactions (e.g. Coulomb) may require summing all infinitely many interactions. There are ways to do this. Short-range interactions, on the other hand, may allow us to ignore all interactions beyond a *cutoff radius*. Sometimes, a nearest neighbour model is enough.

The *minimum image convention* (MIC) is a variant of nearest neighbour interactions: of all periodic images, we will only consider interactions by the closest replica. Look at Fig. 1b to consider the implications of MIC. Let us assume non-negligible interactions with particle 1 (P1) are those within the red circle. There are four images of P3 within the interaction "circle". Of these, the P1–P3 separation within the *cell* (dashed line) is not the shortest. Actually, it is the longest. Under the MIC, the we would only consider the continuous-line P1–P3 interaction. What would be the MIC P1–P2 and P1–P4 interactions?

Finally, be aware the MIC will only produce meaningful results if the simulation box is large enough so that there is, at most, only one meaningful particle pair interaction.

For this project, you will assume that the simulation box is cubic[2]. With this assumption, `pbc.py` code written for Ex. 1 should work, both for PBC and MIC. The only difference is you will have to pass vector *differences* to `mic`. Verify you acted on the given feedback.

3. Detailed Instructions

What follows is 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.

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, which enables the simulation of bulk systems. Enforcing periodic boundary conditions correctly will require care when dealing with particle interactions. It is better to **begin early, before pressure from other courses builds up**.

Do not attempt to write the entire code in one go. Good on you if you can, but single-line debugging is very likely to introduce a lot of physical or behavioural bugs that will be very hard to identify once Python can run it without errors. Instead, identify what code needs to be written to get first 2, then N particles interacting through gravity, evolve their motion in time, and then write an XYZ file.

Working as a group, split tasks based on the Design Doc, assuming your partner sticks to function naming and argument order. If you change those, inform your partner.

3.1. Dealing with N bodies

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). It would also be inflexible. We therefore need a way to automatically go through these for arbitrary N, and lists, arrays and loops are the obvious approach.

The key difference of this project with respect to the Solar System, or Exercise 3, is the need to enforce Periodic Boundary Conditions (PBC) and, most importantly, the Minimum Image Convention (MIC). All particle separations should be computed using the MIC method written for `pbc.py`.

Option 1: Represent the full solar system in a list of particles, where e.g. `particle[i]` is a Particle3D instance (P3D hereafter). This is the simplest approach. Following this

[2] If your MIC code explotis numpy well, your final simulation should work unchanged for cuboid boxes too

approach, you should extend `Particle3D` with static methods taking, at least, a list of P3D instances as arguments:

- A method that computes the total kinetic energy of a list of P3D's.
- Versions of Velocity Verlet updaters for P3D's:
  - a velocity updater for a list of P3D's.
  - a 2^{nd} order position updater for a list of P3D's, *perhaps* respecting PBC
- a method that takes a list of P3D's, and returns a MIC-compliant $[N, N, 3]$ array with the pair separations and, perhaps, also an $[N, N]$ array with the moduli.

Note that, for velocity Verlet updaters to work in this way, the forces must also be *iterable*. This suggests using lists or, better, arrays.

Computing the $\mathcal{O}(N^2)$ separations every step is typically the most time-consuming element of an N-body simulation. In your simulation, the pair separations will be needed, *every step*, to compute at the forces, the energy and at the observables. Therefore, we want to save a substantial amount of time by computing the pair separations only once per step. The method computing the pair separations $(\boldsymbol{r}_i - \boldsymbol{r}_j)$ should help you do that. That method should exploit `numpy`, as well as symmetries if available. It is **critical** that the pair separations are computed observing the MIC.

Option 2: Alternatively, modify P3D so each instance represents N particles. To a point, this ditches OOP. This approach requires you to be comfortable with `numpy` broadcasting.

- Modify the mass, position and velocity of P3D to be, instead, $[N]$, $[N, 3]$ and $[N, 3]$ float arrays. Labels can be in a list.
- Modify the `__str__` method so it prints out *all* particles as per Exercise 2.
- Ensure the kinetic energy method is correct.
- Ensure broadcasting works correctly for the time integrator updaters, and *perhaps* respecting PBC
- Write an *instance* method[3] returns an $[N, N, 3]$ array with the pair separations and, perhaps, also an $[N, N]$ array their modulus.

Neither option will receive higher grades by default. Settle on one, and stick to it.

[3] Why can this be an instance method, but a class method in the first option?

3.2. The Lennard-Jones Interaction Potential

In the previous checkpoint 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 systems, including the leading order r^{-6} attractive term of van der Waals interactions, and a repulsive r^{-12} term to model core–core repulsion. The expression for the energy of two particles at $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ is

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

where $r = |\boldsymbol{r}_1 - \boldsymbol{r}_2|$.

For single-component systems, the LJ potential is actually parameter–free. We can readily introduce reduced units for length, $r^* = r/\sigma$, energy and $E^* = E/\epsilon$. For particles of mass m_0, $m^* = m/m_0$. The derived unit of time is then $t^* = t/\tau$, $\tau = \sigma\sqrt{m/\epsilon}$. Without loss of generality, temperature will be measured in ϵ. The LJ potential now reads

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

with the force 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 $\boldsymbol{F}_2 = -\boldsymbol{F}_1$. Reduced units allow us to neglect potentially very small or very large prefactors common in atomic systems, and change the scale of energies, times,... to those characteristic of the system at hand.

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[4] r_c, and $V(r > r_c) = V(r_c)$. Typical cutoff radii range from 2.5 to 3.5 σ.

Adapt the force vector and energy calculation routines (and their name, if appropriate) so they return the LJ interaction between `P3D` objects instead.

The total force method should exploit the pair separation method, and return the total force acting on each particle. Write a similar method, ideally also exploiting the pre-computed particle separations, to compute the total potential energy of the system.

Initially, you may also want to write a method that takes two particles as arguments and returns the force on one by the other. The potential energy can be computed and returned in the same method, or an auxiliary one. This may be handy in the early debugging stages, to ensure 2-body systems behave as they should.

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

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. The trajectory file can be used to perform further analysis, or to compare different runs, as well as for artistic purposes.

Part of your task consists in writing a trajectory file compatible with the XYZ file format. The P3D `__str__` magic method in Exercise 2 was defined with this in mind. We aim to visualize your trajectory file with software that can read XYZ files, such as VMD. VMD is a standard tool that is used for visualising and analysing trajectories in a variety of scientific fields ranging from biology to astrophysics.

The XYZ format for a system containing two points) 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
⋮
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.

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 timestep to a trajectory file. Note such method would benefit from fancy formatting, if your `__str__` method employed it.

On the CPLab machines, opening an XYZ trajectory with VMD is straightforward:

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

More information on VMD is included in Sec. 3.8

3.4. Initial Conditions

You are provided an auxiliary python file, `mdutilities.py`, to help you set up appropriate initial conditions for particle positions and velocities.

The routine `set_initial_positions()` takes two parameters: `rho`, the reduced particle density, and `particles`, a list of `Particle3D` objects. The routine generates a cubic fcc supercell[5] large enough to accommodate the required number of particles, and places the particles there. It returns the simulation box as a `numpy` array and a boolean stating whether the cell is full. As a sanity check, this method will print the nearest neighbour distance, and a warning if there are vacancies in the cell. This is fine for simulations of fluids, but if you want to simulate a solid, you should consider creating a fully filled box.

Particle velocities are initialised by `set_initial_velocities()`. It takes two parameters[6]: `temp`, the reduced temperature, and `particles`, a list of `Particle3D` objects. It randomly assigns particle velocities $\boldsymbol{v}_i$, but such that the Centre of Mass is fixed, and $E_k = \frac{3}{2}NT$. Be aware these initial conditions may be reasonable, but are not representative of equilibrium.

You may need to adapt `mdutilities` so it can access the positions and velocities as defined in your Particle3D class. If you chose to generalise `Particle3D` to N particles, you will also need to change the position of array indices.

3.5. N-body Simulation Code

We aim to simulate the motion of many particles. This requires setting simulation parameters, the initial conditions for N particles, and writing some data to file (the trajectory at the very least). This requires *you* to answer the following design questions:

- Am I sticking to reduced units?
- How am I obtaining the simulation parameters (such as dt)?
- How will I use `mdutilities`?
- How am I choosing the output file name(s)?

One potential way to approach this[7] would be reading in the simulation parameters from one file with two lines, one for the system setup (T, $\rho\ldots$) and another for the time integration (dt, steps…). Your program, `foo.py`[8], could then take three command line arguments, i.e.:

```
[user@cplab001 ~]$ python foo.py setup.dat data.dat traj.xyz
```

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

[6] Additionally, `set_initial_velocities()` may also take `seed`, a float setting the random seed. This ensures the generated velocities are identical across runs, and is a useful debugging/comparing feature.

[7] certainly not the only one, feel free to choose

[8] Choose something more appropriate than `foo.py`.

where `setup.dat` contains the critical simulation parameters, `data.dat` contains all the data you should write to file (see Sec. 3.7); and `traj.xyz` would be the name of the output trajectory file (with an `.xyz` to allow for seamless use with VMD).

With the above questions settled, we are in a good position to complete `foo.py`, for which the Ex. 3 velocity Verlet integrator should be a good starting point. Remember it should eventually fulfil all functionality requirements in Sec. 1. The main simulation extensions should require little more than using the new N-body methods described above for P3D together with `pbc.py`, the full force method, and the XYZ writer.

This completed program is now conceptually not far off many of the large software packages that exist for performing this type of simulation, such as GULP or LAMMPS.

3.6. Initial test runs

Before implementing the more advanced functionality, you should verify the code produces reasonable physics. By now, your code should:

- run by Python without syntax errors,
- take user-defined simulation parameters,
- use `mdutilities` to set up a simulation box,
- evolve the system through time using velocity Verlet, and
- print the particles' trajectory in an XYZ file.

This should be achieved by CLW, giving you enough time to debug the physics and implement the advanced observables. You should try running your code with the following parameters:

- Solid: 32 particles, $\rho = 1.0$, $T = 0.1\epsilon$, $dt = 0.01$, 1000 steps.
- Gas: 30 particles, $\rho = 0.05$, $T = 1.0\epsilon$, $dt = 0.005$, 2000 steps.

If you are computing the energy (you should!), is it conserved? Do particles in the solid oscillate about their initial positions? Does the gas "explode"? It may not be obvious that your system is behaving adequately; do not hesitate to ask a TA/lecturer for help.

PBC/MIC plus new forces are the usual source of error at this stage. Rather than hacking randomly, you may want to use a LJ dimer as simple test. This requires overriding `mdutilities`, and instead generate manually a list of two particles separated by roughly $\sqrt[6]{2}\,\sigma$. Do they oscillate harmonically? Is energy conserved? Place the particles elsewhere in the unit cell, so interaction is via MIC. Do they still oscillate normally? Is energy also conserved? Is behaviour still physical if you add the same velocity $\boldsymbol{v}$ to both particles?

3.7. 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 compute

- the total, potental, and kinetic energy of the entire system as a function of itme,
- the mean squared displacements (MSD) as function of time,
- and the radial distribution functions (RDF).

Without dissipative forces or heat sinks/sources, 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. These fluctuations after equilibration can be linked to equilibrium properties via statistical mechanics.

Simulations with realistic parameters (number of steps, number of particles...) can take a substantial time. Therefore, all relevant observables should be written to an aptly-named file so further analysis does not require re-running lengthy simulations.

3.7.1. Mean Squared Displacement

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

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

You should 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 at regular intervals. If your position updater enforces PBC, you will need to use MIC to evaluate the displacements (*Think why*).

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.

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

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

When analysing the MSD, you should consider what is the largest displacement you can model in a system with enforced periodic boundary conditions.

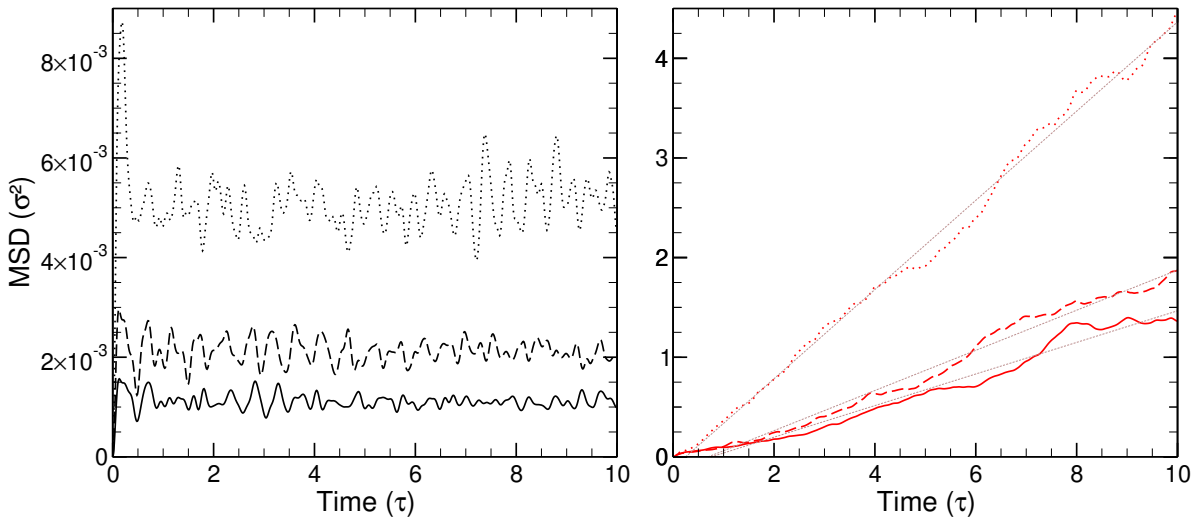


Figure 2.: MSD of a LJ solid (left, black) and a fluid (right, red), each at three different temperatures. Note the different orders of magnitude. Straight lines show the average linear behaviour of the liquid MSD.

3.7.2. Radial Distribution Function

The RDF (sometimes also called Pair Distribution Function), $g(r)$, 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 the microscopic structure of a system, and how ordered it is. the RDF is defined as

$$g(r) \equiv \frac{1}{N\rho_0(r)} \left\langle \sum_{i,j} \delta(r_{ij} - r) \right\rangle, \tag{6}$$

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

From the above definition, if $g(r) = 0$, we would never find two particles separated by a distance r. Conversely, if $g(r) = 1$, then, on average, we expect to find, in a shell of radius r and width $\mathrm{d}r$, as many particles as in the homogeneous system.

The RDF average is usually determined by regularly calculating (reusing) all pair separations r_{ij} (adhering to the MIC in our case) and binning them into a histogram with defined width Δr. You should write a method that performs this data collection at regular intervals. You should then normalize the histogram data, noting our discretization implies $\mathrm{d}r \to \Delta r$. 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 3.

It should be emphasised the RDF is a useful concept: a system's symmetry and its

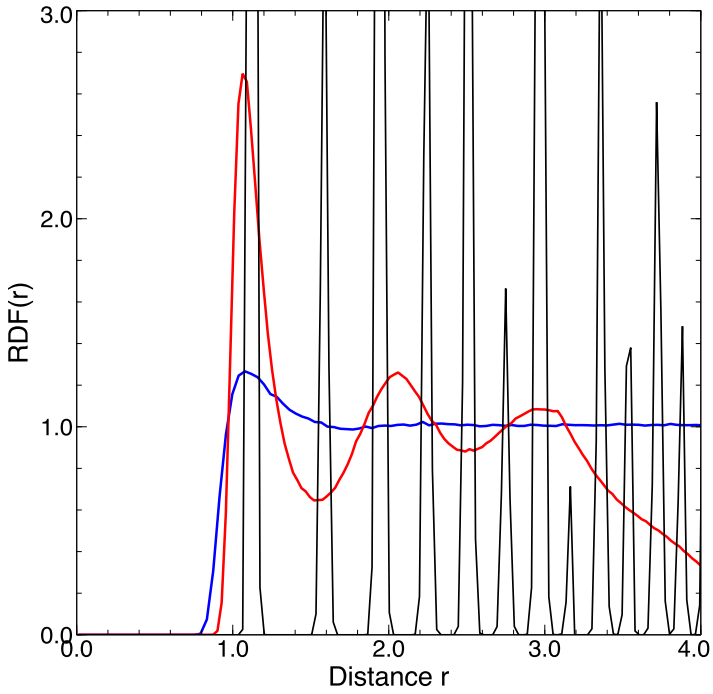


Figure 3.: RDF of a LJ solid (black), fluid (red), and gas (blue). Note the erroneous drop-off of the fluid-RDF at large distances, where data collection extends beyond half the box size.

RDF determine its neutron and X-ray scattering, for example. If interactions are known, the RDF is also linked to the external pressure via the Virial theorem.

3.8. Plotting with VMD

Compatibility notice: 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. Versions 1.9.2 and older used to crash with Solar System trajectories in metres. Consider writing the trajectory coordinates in AU, 10^{11} m or similar, even if the code works internally in SI.

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 all 12 bodies every 6 hours for 1000 years, 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 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:

- 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
```

3.9. Performance

Early optimization is the root of all evil. Make sure your code runs. Then make sure your code runs correctly. Only then think about performance.

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. Try to write your code such that you only compute pair separations once per step. That alone should result in acceptable performance: a recent Intel Core i5 should run the a 32-particle (solid) simulation for 1000 steps in well under 30 seconds.

4. Submission

Your submitted code will be anonymised and then uploaded for a plagiarism check using `moss`. None of your files should include your exam number (e.g. B0000). Please make

sure your names and (maybe) matriculation numbers appear only on the headers of the python files, and not elsewhere.

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-b`, you can use the `tar` command by running:

```
[user@cplab001 ~]$ tar -zcvf project-${USER}.tar project-b
```

Verify the command ran without errors. Now, in a different folder, decompress the tarball to ensure the compressed package is not empty and contains the intended files. The documentation of your simulation program should describe how to run it and what the format and units of the input and output files are.

Submissions larger than 50 MB will not be accepted. 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?). One of your group members should submit the compressed package (e.g., `project.tar.gz`) through the course LEARN page, by **16:00 on Thursday, week 8 of semester 2**. Successful submissions are emailed a receipt.

5. 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. LJ Simulations of Argon

The LJ potential, initially popular because of its numerical simplicity, has the leading attractive van der Waals term. Even though its repulsive r^{-12} term is too soft, LJ turns out to be a reasonable model of the interactions of noble gases, of which the closest match is Argon. Some of the pioneering works of computational condensed matter physics consisted in a comparison of the properties of Ar with those of of a LJ system. We are now going to follow some of the steps of those physicists.

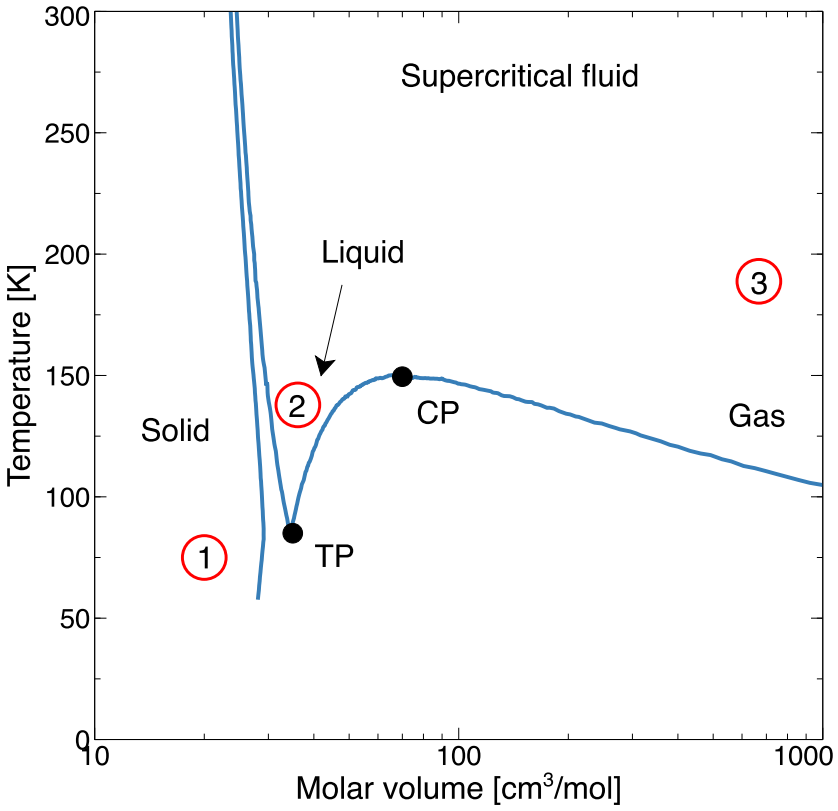


Figure 4.: Phase diagram of Argon. Solid, liquid, gas, and supercritical regions are labelled, as are triple (TP) and critical point (CP). Numbers 1-3 indicate regions suitable for simulations' initial conditions.

The phase diagram of Ar is show in Fig. 4, together with three regions you may want to explore. First, however, we need to convert the Volume–Temperature regions in Fig. 4 to numbers your code understands. Remember the LJ units are completely defined by the potential parameters[9]. A common choice of LJ parameters for Argon is $\epsilon/k_B = 119.8\text{K}$, $\sigma = 3.405 \times 10^{-10}\text{m}$, and $m = 0.03994\text{kg/mol}$.

[9]The critical point in a LJ system is $T_c = 1.326\epsilon$, $\rho_c = 0.316\sigma^{-3}$, and can be used to uniquely set the LJ parameters.

Set up simulations in the solid (1), liquid (2) and gas (3) regions of phase space, with appropriate initial temperatures T^* and densities ρ^*. For the solid state, choose atom numbers that result in a fully occupied face-centered cubic lattice, while for the fluid and gas you should choose atom numbers with a certain number of vacancies, to avoid being trapped in a metastable state.

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. Near the melting point, the MSD is very useful to identify melting.