**Project B: Phase Diagram of Argon**

**Design Document**

1. **Aims and Overview**
2. **hello**

This program aims to simulate an N-body system interacting via the Lennard-Jones (LJ) pair potential, given user-defined initial conditions. A simulation box with a finite number of particles and Periodic Boundary Conditions (PBCs) will be used. The output will be a trajectory file, with which the particles’ trajectories can be visualised using a molecular visualisation program, VMD (Visual Molecular Dynamics).1 For the equilibrium properties of the simulated system to be analysed, the program will write the system’s total, kinetic and potential energies and the particles’ mean square displacement (MSD) to a file at regular time intervals. The particles’ radial distribution function (RDF) will be visualised as a histogram.

The LJ potential is given by the sum of two interactions felt by non-ideal gases. The first is an attractive force between particles, the ‘van der Waals interaction’. This is the result of the attractive forces between permanent or temporary induced dipoles, and/or dispersion forces. The other interaction is a repulsive force. It is small in magnitude and therefore only effective at close proximity. As a result, LJ forces have an attractive and repulsive component. Which component dominates depends on the separation between the two particles in question. The LJ potential is given in equation 1, where is the hard sphere diameter and is the depth of the potential well.2

(1)

1. **Input and output**

**2.1 Units**

In this program, reduced units for length, energy, mass, temperature and time will be used. These, labelled with an asterisk, are given in equations 2 to 6 respectively. Note that the values of and depend on the system in question.

|  |  |
| --- | --- |
|  | (2) |
|  | (3) |
|  | (4) |
|  | (5) |
|  | (6) |

The LJ potential in terms of reduced units is given by equation 7.

In terms of reduced units, the force on the particle at **r1** is given by equation 8.

**2.2 Input**

The program will require an input file, param.input. This will contain the following, where relevant in reduced units:

* number of particles
* number of simulation steps
* timestep
* temperature
* particle density
* LJ cut-off distance

The format will be:

<no. particles> <no. steps> <timestep> <temp> <density> <LJ cut-off>

**2.3 Output**

A trajectory file, traj.xyz, will be created. The following format, which is compatible with visualisation with VDM, will be used:3

<number of points to plot>

<title line (in the example below it will give the timestep number)>

<particle label> <x coordinate> <y-coordinate> <z coordinate>

The final line is repeated N times for N particles.

An example is given below, for a two-particle system and m timesteps:3

2

timestep = 1

s1 x11 y11 z11

s2 x21 y21 z21

2

timestep = 2

s1 x12 y12 z12

s2 x22 y22 z22

2

timestep = m

s1 x1m y1m z1m

s2 x2m y2m z2m

Three further output files will be created, containing, at regular time intervals:

* The system’s total, kinetic and potential energies, in file energies.dat. This will have the format:

<kinetic energy> <potential energy> <total energy> <time>

* The particles’ MSD, in the file MSD.dat. It will have the format:

<MSD> <time>

* The particles’ RDF visualised as a histogram using matplotlib.

1. **Module structure**

A schematic diagram of the program’s module structure is shown in figure 1.

The simulated particles will be represented by the class Particle3D.py. The main module, particleManyBody.py, will import the class Particle3D.py and functions from the modules MDUtilities.py and pbc.py.

Particle3D.py

Main module:

particleManyBody.py

MDUtilities.py

pbc.py

Figure 1: Module structure for Computer Modelling Project B: Phase Diagram of Argon. Module descriptions can be found in sections 4 and 5.

1. **Module description**

**4.1 pbc.py**

Two functions in this file are needed; one which uses periodic boundary conditions (PBCs) to return the image of a particle in a box of user-defined dimensions. The other function considers two particles. It takes the first particle and locates the image of the second particle which is closest to it. This allows the force between the two particles to be calculated using the minimum image convention (MIC).

**4.1.1 Functions from pbc.py**

**image\_in\_cube(particle, l)**

Takes one Particle3D object as the argument, as well as the box size, l.

Updates the particle’s position to the image of the particle within a box of length l, using the numpy.mod() function.

**image\_closest\_to\_particle(particle1, particle2 , l)**

Takes two Particle3D objects as arguments, as well as the box size, l.

Returns the image of the particle2 which is closest to particle1, in accordance with the MIC. The method used is given in equation 9.

(9)

**4.2 Particle3D.py**

In this class an instance represents a particle in 3D space.

**4.2.1 Properties**

The properties of a Particle3D instance are listed in table 1.

|  |  |  |
| --- | --- | --- |
| Name | Type | Notes |
| name | string | Label of the particle |
| position | NumPy array | Position as 3D coord (x1, x2, x3) |
| velocity | NumPy array | Velocity vector (v1, v2, v3) |
| mass | float | Mass of the particle |

Table 1: The properties of a Particle3D instance, represented by the Particle3D.py module.

**4.2.2 Instance methods**

**\_\_init\_\_(self, name, position, velocity, mass)**

Initialises a Particle3D instance using the constructor. The details of these arguments are given in table 1.

**\_\_str\_\_(self)**

Defines the output format. Returns a string giving the name and position of the particle.

For particle called 'tester' at position = (2.0, 0.5, 1.0) this will print as:

tester 2.0 0.5 1.0

**kinetic\_energy(self)**

Returns the kinetic energy of the particle, using equation 10.

**leap\_velocity(self, dt, f)**

A first-order velocity update. The arguments are explained in table 2.

|  |  |  |
| --- | --- | --- |
| Name | Type | Notes |
| dt | float | User-defined timestep |
| f | NumPy array | Force vector acting on particle, (f1, f2, f3) |

Table 2: Arguments used in instance methods leap\_velocity() and leap\_pos2nd().

The particle velocity is updated based on the previous velocity and the arguments dt and f. Equation 11 is used.

**leap\_pos2nd(self, dt, f)**

A second-order position update. The arguments are detailed in table 2.

Both velocity and acceleration are used to find a more accurate position (compared to a first-order velocity update). The particle’s position is updated based on the particle’s previous position, its mass and velocity, and arguments dt and f. This is shown in equation 12.

**4.2.3 Static Methods**

**particle\_separation(particle1, particle2)**

Takes two Particle3D objects as arguments.

Returns the modulus of the separation of the two particles. The NumPy linear algebra normalisation function is used, acting on the vector representing the distance between the particles.

**4.3 MDUtilities.py**

This module is responsible for setting the initial conditions of the particles. There are two functions in this module. Both require a list of Particle3D objects as an input. The temperature and another parameter, rho, are also used. rho is the particle density within the simulation box.

**4.3.1 Functions from MDUtilities**

**set\_initial\_positions(rho, particles)**

Takes the arguments rho and ‘particles’, a list of Particle3D objects.

The function updates the position vectors of each Particle3D object. The size of the simulation box is determined, and the particles are spaced evenly in a lattice arrangement within the box. An error message will be produced if the number of particles is inconsistent with the number of particles required to fill a fcc lattice completely.

**set\_initial\_velocities(temp, particles)**

Takes the arguments temperature and ‘particles’, a list of Particle3D objects.

The function updates the velocity vectors for each Particle3D object. Each particle is assigned a random initial velocity, between -0.5 and 0.5 in each dimension, using the random number generating function. This random velocity is then scaled, using the Boltzmann factor and the given temperature of the particles. This effectively simulates the random motion of particles under the given conditions.

1. **Main code description: particleManyBody.py**

**5.1 Functions used in main code**

**force\_lj(particle1, particle2)**

The arguments are two Particle3D objects.

Returns the Lennard-Jones force on particle 1, using equation 8. The force felt by the other particle in this interaction is equal and opposite.

**pot\_energy\_lj(particle1, particle2)**

The arguments are two Particle3D objects.

Returns the energy of the particles due to the Lennard-Jones potential, calculated using equation 7.

**MSD(particles)**

The argument is the list of all the Particle3D objects in the simulation.

Returns the average MSD over all particles, using equation 13. N is the number of particles, **ri**(t) is the position of particle i and **ri0** is the initial position of particle i.

**RDF\_collection(particles)**

The argument is the array of Particle3D objects.

Returns the distances between each pair of particles, placed into bins of width dr. This can be done using the matplotlib histogram function. Distance between the particles is measured by the modulus of the separation, as shown in equation 14.

**RDF\_normal(binned\_particles)**

The argument of this function is an array, the array produced by the RDF\_collection function.

It returns a normalised array of the distances found in the previous function. It does this by dividing the number of particle pairs in each interval by Nwhere is the number of timesteps taken in the whole of RDF\_collection and is number density. This is shown in equation 15.

Produces a graph of the results using matplotlib.

**5.2 Description of main code**

The main code will print out a trajectory file for an N-body system interacting via the LJ pair potential, which can be visualised using VMD. In order to do this, the main code will go through the following steps:

1. Import NumPy library, sys module, matplotlib, the Particle3D class and the functions in modules pbc.py and MDUtilities.py.
2. Read in the input parameters from the input file, param.input. Assign variables to these numerical values.
3. Create a list of Particle3D objects, with arbitrary positions and velocities. The list should have length equal to the user-defined number of particles, N.
4. Apply the **set\_initial\_positions()**and **set\_initial\_velocities()** functions to the list created in step 3.
5. Print the number of particles and a header line to the trajectory file traj.xyz. Print the initial positions of all the particles (in the list of Particle3D objects) to the trajectory file using the function \_\_str\_\_()**.**
6. For one particular particle in the list of Particle3D objects, apply image\_closest\_to\_particle() function N-1 times. The arguments will be the particle in question, and in turn, each of the other particles present. Each time, using the static method particle\_separation(), calculate the particle separation. If this is greater than the LJ cut-off distance (user-defined), this interaction can be ignored. If not, calculate the force on this particle using the function force\_lj(). Add up all the relevant forces to obtain the total force on the particle. Using this force, use the leap\_velocity() function to update the particle’s velocity. Using this same force, and the function leap\_pos2nd(), update the particle’s position.
7. Repeat step 6 for the N-1 other particles in the list of Particle3D objects.
8. Use image\_in\_cube()function to find the image of each particle in the simulation box. The function will update each particle’s position to the position of this image. Effectively this allows the particles leaving the simulation box to re-enter it on the opposite side.
9. Repeat steps six to nine M-1 times, where M is the user-defined number of simulation steps. Every X number of timesteps (where X is a hard-coded number of time-steps), print the number of particles and a header line to the trajectory file traj.xyz. Print the positions of all the particles (in the list of Particle3D objects) to the trajectory file using the function \_\_str\_\_(self)**.**

To print the system’s equilibrium properties to an output file every X timesteps, the following steps should be taken every time the particle positions are printed to the trajectory file:

1. Calculate the kinetic energy of each particle using kinetic\_energy() instance method from the Particle3D class. Sum these to get the total kinetic energy of the system.   
   Calculate the potential energy of each particle-particle interaction using the pot\_energy\_lj() function, and remembering not to double-count. Sum these to get the total potential energy of the system.   
   Calculate the total potential energy by summing the total kinetic and potential energies.   
   Print these three total energies, as well as the time elapsed, to the file energies.dat in the format described in section 2.3.
2. Calculate the MSD using the function MSD(). Print this and the time elapsed to the file MSD.dat, using the format described in section 2.3
3. Calculate the RDF using the RDF\_collection() and RDF\_normal() functions. Produces a normalised histogram of the RDF for a range of distances.

To run the code, the following command line should be entered into the terminal:

$ python3 ParticleManyBody.py param.input traj.xyz

1. **References**
2. Theoretical and Computational Biophysics Group, UIUC. *Visual Molecular Dynamics*. Available at: http://www.ks.uiuc.edu/Research/vmd/. (Accessed: 18th January 2020)
3. Blundell, S. & Blundell, Katherine M, 2010. *Concepts in thermal physics* 2nd ed., Oxford: Oxford University Press.
4. University of Edinburgh, 2018. *Computer Modelling Project B Instructions: Phase Diagram of Argon.*