Computer Modelling: Lennard-Jones PBC Argon Simulation

Authors: Ethan van Woerkom and Christos Kourris

Date: 24 January 2019

Overview:

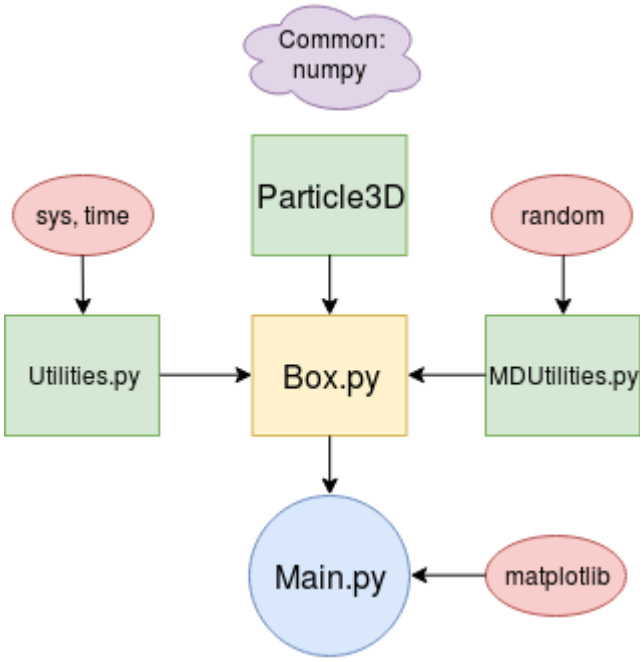
The program will model the behavior of Argon at different initial conditions using a Lennard-Jones potential in an N-Body simulation, with the Velocity Verlet integration scheme. The minimum image convention and periodic boundary conditions will be applied to the box of simulation. The Lennard Jones potential is given by:

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

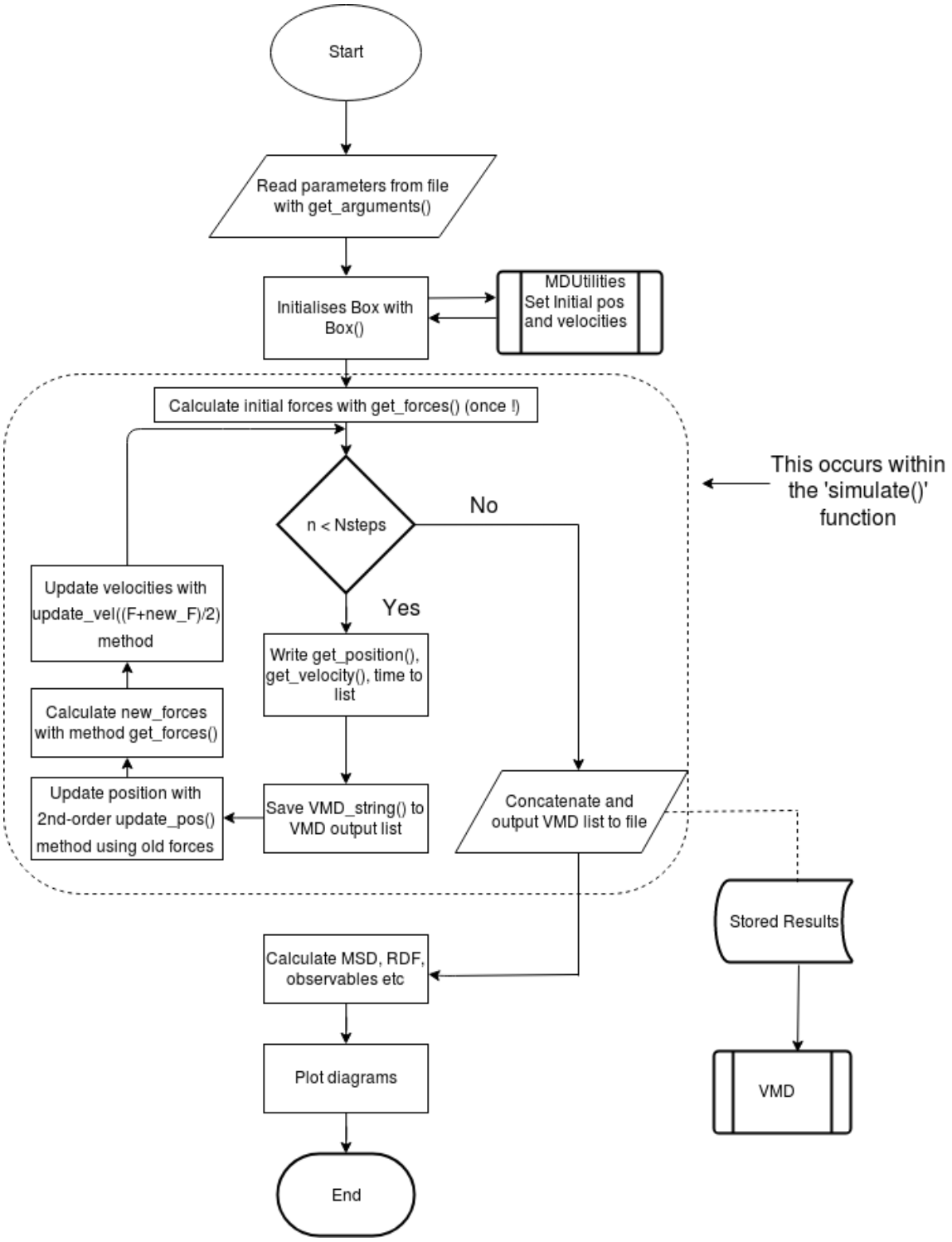
This document will explain the structure of the program by describing the different modules, classes and functions explicitly that will be used as well as their dependence. The code will be written in Python and the results will be analyzed in VMD.

Operation and code description:

The different modules that will be used and their interaction is diagrammatically shown in the chart below. The main function and simulation functions are schematically described in the flowchart on the next page.



Detailed Flowchart:



Unit Discussion:

In molecular dynamics and this simulation in particular we use reduced units in which $\varepsilon = 1$, $\sigma = 1$ and $m = 1$, where ε and σ are the Lennard Jones parameters for the substance being simulated, Argon in this case. Using dimensional analysis we can derive the units of the other relevant quantities as:

- Energy: $E^* = \varepsilon$
- Distance: $L^* = \sigma$
- Temperature: $T^* = \varepsilon/k_B$
- Pressure $P^* = \varepsilon/\sigma^3$
- Time: $\tau^* = \sigma\sqrt{(m/\varepsilon)}$
- Number density: $\rho^* = 1/\sigma^3$

In the reduced units, the Lennard Jones potential and the force take the form:

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

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

Running Program

The program is executed using the following syntax:

```
python Main.py parameters.txt vmdoutput.xyz
```

"parameters.txt" contains the simulation parameters. "vmdoutput.xyz" is where trajectory data will be written.

Example parameters.txt:

```
Nparticles
108
Reduced Density
1
LJ Cutoff Distance
2.5
Reduced Temperature
1
Reduced timestep
0.01
Simulation steps
10000
```

Program Modules:

- `Main.py`
  Contains the main function, conducts simulations, graphing and analysis

- `Particle3D.py`
  Contains the Particle3D class and methods.

- `Box.py`
  Contains the Box class and methods.

- `Utilities.py`
  Contains all other functions (integrators,pbc checks)

- `MDUtilities.py`
  Contains auxiliary set_initial_positions and set_initial_velocities methods that initialize the Box object.

Explanation of basic structure of code

In this program we have chosen to use a high degree of modularity and object-orientation. For this reason the state of our simulation is not held in a list of particle data, coupled with numerous other variables, but everything is grouped together in a ‘Box’ class.

The Box class holds particle data using a list of Particle3D objects, box dimensions and simulation parameters. Finally, all interactions with the simulation are done with and controlled by new methods implemented in the Box class. The simulation function is responsible for the simulation.

The program is executed from the Main.py file containing the main() function, which is explained in the flowchart. It reads data from the parameter file using a utility function, outsources initialisation and simulation to the corresponding functions. The only significant code in the main funciton, is for the analysis of the results.

Visualisation of the simulation will be done by applying VMD to the VMD output file.

Git is used for version control.

Functions

- **Utilities.py**

```
get_arguments()
     returns:
     param = [N, rho, LJ_cutoff, T, dt, nsteps], VMDfile
```

The program is called with two filenames as input. The first one contains the simulation parameters. The second one is the name of the file to which the VMD output will be written. This function parses those inputs and returns a list with the contents of the first file and a string with the name of the second file.

```
LJ_Potential(vector, cutoff)
     returns: scalarpotential
```

This function calculates the Lennard-Jones potential for a particle with the given vector as separation, using a given cutoff distance. It returns a scalar.

```
LJ_Force(vector, cutoff)
     returns: force
```

This function calculates the Lennard-Jones force vector for a particle separated by the vector in distance, using a given cutoff distance. It returns an narray.

```
Total_PE(positions, cutoff)
     returns: energy
```

This function returns the total calculated potential energy for a set of system positions given by an [N, 3]-dimensional narray, using a given LJ cutoff.

```
Total_KE(velocities)
     returns: energy
```

This function returns the total calculated kinetic energy for an [N,3] dimensional narray of system velocities.

```
Bin_particles(pos, bins, boxdim)
     returns: bin_entries
```

Given a [N,3]-dimensional narray of system positions, this will calculate the distance between all pairs of particles (respecting PBC and MIC) and bin these using the given bins into an entries array. Finally this array will be divided by the number of particles in the system, to give the average #neighbours for each distance range from a particle in the system.

```
RDF(bins, bin_entries, num_density)
     returns: radial_density_histogram_entries
```
Given a bins and bin_entires narray produced by Bin_particles, this will calculate the radial density function histogram using the given bins by performing a normalisation of dividing each bin by its approximate size equal to $4\pi r^2 \rho dr$, with ρ=num_density.

```
MSD(pos, start, length)
     returns: mean_square_displacement
```
Given a [T, N,3]-dimensional narray of system positions indexed by time, this will calculate the mean square displacement for the system from time start to start+length exclusive relative to the given start time.

- **MDUtilities.py**

```
set_initial_positions(rho, particles)
     return np.array([box_size, box_size, box_size])
```
Sets up initial positions of a Particle3D list in FCC configuration, then returns corresponding box dimensions in narray.

```
set_initial_velocities(temp, particles)
     return None
```
Initialises a Particle3D list to a given temperature with random velocities.

Classes

class Particle3D:

Variables:

```
label (str)
pos (3-dim narray)
vel (3-dim narray)
mass (float)
```

Methods:

```
__init__(self, label, pos, vel, mass)
     returns: None
```
Initializes a Particle3D object by giving it a label, position, velocity and mass.

`__str__(self)`
`returns: VMD_string`
Returns a string with the Particle's data in the form:
<label> <x> <y> <z>

`kinetic_energy(self)`
`returns: energy`
*Returns the kinetic energy of the particle calculated by the velocity narray: K = ½*m*(vx^2 + vy^2 + vz^2)*

`leap_velocity(self, dt, force)`
`returns: None`
Updates the velocity of the particle using the previous velocity up to second order: v(t+dt) = v(t) + (a(t) + a(t+dt))/2

`leap_position(self, dt, force)`
`returns: None`
Updates the position of the particle using the previous position and velocity up to second order:
*x(t+dt) = x(t) + v(t)*dt + ½ a(t)*dt^2*

`sep(p1, p2) (staticmethod)`
`returns: separation_vector`
Returns the separation vector between the position vectors of two particles as an narray.

`pbc_sep(p1, p2, boxdim) (staticmethod)`
`returns: mic_separation_vector`
Returns the separation vectors of two particle objects based on the minimum image convention and the periodic boundary conditions, as an narray.

Class Box:

Variables:

`particles (list of Particle3D)`

List containing information about all particles.

`boxdim (float)`

Dimension of box.

`LJ_cutoff (float)`

Lennard-Jones cutoff distance.

Methods:

`__init__(self, N, LJ_cutoff, rho, T)`

`returns: None`

Initialises simulation box with given parameters using function from MDUtilities.py to set particle positions and velocities.

`update_vel(self, forces, dt)`

`returns: None`

*Conducts first order velocity update given an narray of forces on all particles, using v = v + F/m*dt*

`update_pos(self, forces, dt)`

`returns: None`

*Conducts second order position update given an narray of forces on all particles, using x = x + v*dt + 0.5*F/m*dt^2.*

`get_positions(self)`

`returns: particles_positions`

Returns [N,3]-dim narray of positions of all particles.

`get_velocities(self)`

`returns: particles_velocities`

Returns [N,3]-dim narray of velocities of all particles.

`get_forces(self)`

`returns: particle_forces`

Returns [N,3]-dim narray of forces on all particles.

```
VMD_string(self, time)
     returns: VMD_output_string
```
Produces a string in the VMD format giving the state of the current system, with Point = T. Uses the Particle3D.__str__ method.

```
enforce_pbc(self)
     returns: None
```
Enforces period boundary conditions and moves any particle that has strayed outside the box back into the box according to pbc.

```
simulate(self, outputfile, nsteps, dt)
     returns: positions, velocities, times
```
Runs a Verlet n-body simulation on the initialised box for nsteps with timestep dt, and returns [nsteps,N,3]-dim position and velocity narrays and a nsteps-length time narray. After running, print simulation timelength.