Really Simple MOLECULAR DYNAMICS *with Python*

Eric M. Furst
Department of Chemical and
Biomolecular Engineering
University of Delaware

Department of Chemical and Biomolecular Engineering
Allan P. Colburn Laboratory
150 Academy Street
University of Delaware
Newark, Delaware 19716 USA

Contents

Chapter 1

Introduction

Molecules! Chemical engineers transform molecules, taking some apart to build new ones at an almost unimaginably small scale. As *molecular engineers* it may be frustrating that we cannot *see* what these basic pieces of matter are doing. Our field is one of imagination and (mostly) indirect measurement. We can, however, build simple yet powerful models of those molecules in computers. These help us to peer into their tiny world and fast dynamics. Our models are *molecular simulations* and they serve a purpose far beyond just building intuition. They enable use to calculate thermophysical and transport properties that are fundamental to the processes of molecular transformation, including thermodynamics.

The purpose of `CHEG231MD` is to provide a simple molecular dynamics simulation code base written in fairly plain Python and standard libraries. We can use it explore the basic concepts of molecular simulations and molecular thermodynamics. It is not a high performance simulation. Those seeking modern simulation tools should use LAMMPS or other powerful packages. Still, the `CHEG231MD` performance is probably better than Alder and Wainwright's work on an IBM 704, which recorded 5000 collisions after an hour of running a simulation of just 32 particles.[1] We've come a long way in seventy years! With around 150 lines of code, `CHEG231MD` should be straightforward to understand and to modify.

[1]After Berni Alder and Thomas Wainwright's first paper [1] they reported a more general description of their methods and aims just a few years later. [2] To give a little context to these early computer calculations, the first computers started to be built in the late 1940s and early 1950s, including ENIAC at the University of Pennsylvania and MANIAC at Princeton. Enrico Fermi and his colleagues John Pasta and Stanislaw Ulam used the MANIAC I computer at Los Alamos National Laboratory to calculate the dynamics of 32 particles in a one-dimensional system.[3] Commercial stored-program computers were introduced by UNIVAC in 1951. By 1960, there were perhaps five or six-thousand computers worldwide! [4] Computers were so rare then that scientists working in the country's new, highly secret atomic weapons program even ran their code (called HIPPO) on a machine that sat in the storefront windows of the big IBM Madison Avenue showroom in New York City. [5] All of these programs were written in machine language. The FORTRAN programming language was introduced by IBM in 1958.

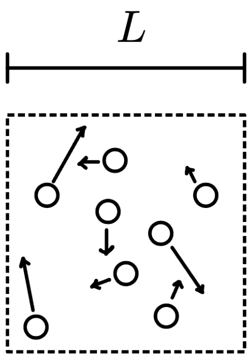

Figure 1.1: A molecular dynamics simulation numerically calculates the equations of motion for N molecules as they move and interact.

The purpose of this document is to explain a bit[2] about the internals of `CHEG231MD`, how to get it running, and how to modify it. Along the way, I will introduce basic concepts of molecular simulations. For a comprehensive treatment of the subject, see Daan Frenkel and Berend Smit's excellent book, *Understanding Molecular Simulation* and the many resources that they cite. [6]

1.1 The basic scheme

The heart of a molecular dynamics simulation is a numerical integration of Newton's equations of motion. We will start very simply. Our system will be composed of monatomic molecules with mass m. These molecules will have a diameter σ and we'll imagine them at a *number* density ρ in a volume V. We apply

$$\mathbf{F} = m\mathbf{a} \tag{1.1}$$

to each molecule, solving for its motion through space as a function of time. The forces are due to interactions with the other molecules, such that the total force on molecule i is

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{f}_{ij} \tag{1.2}$$

where $\mathbf{f}_{ij}$ is the pair interaction *force* between molecules that are separated by a distance r_{ij}. Each pair force is calculated from the pair interaction *potential*,

$$\mathbf{f}_{ij} = -\boldsymbol{\nabla} u(r_{ij}). \tag{1.3}$$

[2]No pun intended.

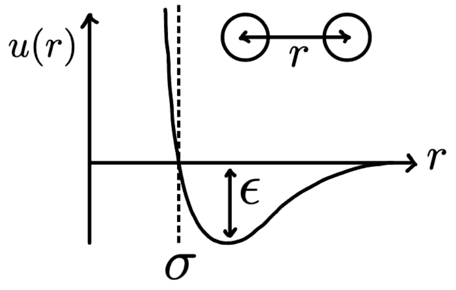

Figure 1.2: The pair potential between molecules.

The interaction potential we will use in this simulation is the Lennard-Jones or 6-12 potential,

$$u(r_{ij}) = 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{1.4}$$

where ϵ has units of energy.[3] The potential is represented with schematically in figure 1.2.

So, for each molecule i in the system, we solve the equation of motion

$$m\frac{d^2\mathbf{x}_i}{dt^2} = -\sum_{j\neq i}\boldsymbol{\nabla}u(r_{ij}). \tag{1.5}$$

These are, of course, vector quantities that must be calculated in all three dimensions. Moreover, they are $3N$ coupled differential equations as the force on each molecule at any instant in time depends on the distance to the other molecules in the simulation.

What can you do with such a calculation? At any moment, you can calculate the pressure, temperature, and different components of energy—namely, the potential energy of the intermolecular interactions and their translational kinetic energy. By specifying different densities and energies for subsequent simulation runs, one can work out values related to the mechanical and thermal equations of state, and perhaps even begin to see the onset of phase separations.[4]

[3]Sometimes u is used to represent the thermodynamic internal energy per molecule, such as when one calculates the internal energy of an ideal gas per molecule as $\frac{3}{2}kT$. Here it is a symbol for the potential energy between two molecules.

[4]For reasons beyond the scope of this document, we will not be able to simulate two phases in equilibrium, but more sophisticated calculations can!

1.2 Interaction potential

The pair interaction potential in equation 1.4 captures two of the basic properties of neutral molecules: that they have a size (and thus they repel each other) and that they exhibit a long-range[5] attraction due to *van der Waals* interactions. The $1/r^{12}$ is a steep function that approximates the repulsion between the monatomic species. The $1/r^6$ contribution captures the form of the *dispersion* interactions between molecules.

These latter attractive interactions arise from the instantaneous polarization of the molecule's electron cloud, which induce disturbances in neighboring molecules. All of that happens in timescales closer to femptoseconds, and thus, we can approximate them with a potential of this form. This concept highlights the multiple time and length scales of molecular phenomena and the process of *coarse graining* a simulation. We can approximate the quantum mechanical and fluctuating nature of the electron distribution in a real molecule with the interaction potential because the timescales for molecular motion are much longer than the timescales of their electronic transitions. Thousands of femptoseconds (or a picosecond), in fact! These ubiquitous attractions are what gives us condensed phases like liquids and their associated phase equilibria.

The 6-12 potential is convenient because it is a continuous function, and, provided we select short enough time steps in the simulation, easy to evaluate.

Because the pair potential captures both the finite volume and attraction of molecules, we should also be able to relate its parameters back to the van der Waals equation of state,

$$P = \frac{RT}{\underline{\mathrm{V}} - b} - \frac{a}{\underline{\mathrm{V}}^2}, \tag{1.6}$$

where a represents the attractive interactions, and b the finite size of the molecules on a molar basis.

Exercise 1.1: Plot the Lennard-Jones pair potential for the parameters $\epsilon/k = 120$ K and $\sigma = 0.34$ nm. *Hint: the scales are very small. You may want to plot this using dimensionless values, such as u/kT and r/σ.*

Exercise 1.2: What is the separation of the minimum interaction energy for the Lennard-Jones potential? What is the value of this minimum? At what separation does the *force* become repulsive?

1.3 Outline

This document is arranged in several sections. In the next chapter, we will dive in and run our simple MD code. After that, I will go through the code

[5] It's all relative!

and describe its inner-workings. Then we should have enough background to try solving some problems with MD simulations, including visualizing the dynamics and structure of the molecules as they move in time, driven by the interactions between them and the (kinetic) energy they start out with.

Chapter 2

Getting started

Get yourself set up to use `CHEG231MD`. The simulation requires that the `numpy` and `numba` packages. You will probably want to have the `matplotlib` library installed, too. All of these are available on the default `conda` channel. I run the simulation in a Jupyter notebook. I'm assuming that you will, too. Kinder and Nelson is a good reference for help getting set up with Jupyter and conda. [7]

2.1 Start a simulation

The `CHEG231MD.py` code (see appendix B for a listing) defines the `MDSimulation` class. To start a simulation, simply create an object of this class, like this line from a Jupyter notebook, which I'll call `MDsim.ipynb`:

```
import CHEG231MD as md

# create a simulation with # particles, density, and max speed
# N^(1/3), rho+, max vel+

sim = md.MDSimulation(5,0.7,5.8)
```

Here, `sim` is an object of the `MDSimulation` class. The parameters passed to the simulation—the conditions that we will run it under—are the cube root of the number of particles $N^{1/3}$, the dimensionless density ρ^*, and the maximum dimensionless velocity v^*_{max}.

In your simulation, N should vary between 125 and 1000, corresponding to $N^{1/3}$ between 5 and 10. These values will keep the simulation running at a reasonable speed. A lower number of molecules in the simulation leads to more uncertainty in the calculated values of pressure and temperature. A higher number means that we must evaluate many more pair interactions during each time step. Modern simulations use neighbor lists and other techniques to simulate much larger system sizes without getting bogged down.

A note on dimensionless variables

It is a good practice to use nondimensional and scaled variables in numerical solutions to physics and engineering problems. The reason is that a computing machine represents a number with a finite precision. Very small or very large numbers can exceed this limit, which leads to errors in the calculation.

Molecular motion occurs on very short timescales over small distances involving small masses—relative to our macroscopic world. Our choice units, the SI system, in fact biases our calculations to the macroscopic world, not the molecular world. The diameter of an argon atom is roughly 0.14 nm; hydrogen is about 0.1 nm and helium is 0.06 nm. Their masses are on the order of 10^{-23} kg. Our energy scale becomes the attractive interaction between molecules, ϵ.

If we rescale the equations of motion based on the molecular diameter, mass, and energy we can find the characteristic timescale $\sqrt{m\sigma^2/\epsilon}$ such that the dimensionless time is

$$t^* = t/\sqrt{m\sigma^2/\epsilon} \tag{2.1}$$

while the nondimensional distance between molecules becomes

$$r^* = r/\sigma. \tag{2.2}$$

We write the non-dimensional Lennard-Jones interaction interaction potential as

$$u^*(r^*) = 4\left[\left(\frac{1}{r^*}\right)^{12} - \left(\frac{1}{r^*}\right)^{6}\right] \tag{2.3}$$

where $u^* = u/\epsilon$. Likewise, the dimensionless force is

$$f^* = f/(\epsilon/\sigma) \tag{2.4}$$

and the dimensionless velocity

$$v^* = v/\sqrt{m/\epsilon}. \tag{2.5}$$

The dimensionless (number) density is the number density scaled by σ,

$$\rho^* = \rho\sigma^3. \tag{2.6}$$

With these non-dimensional units, the pressure is

$$P^* = P/(\epsilon/\sigma^3) \tag{2.7}$$

while the temperature scale is set by the depth of the attraction, such that the non-dimensional temperature is

$$T^* = T/(\epsilon/k) \tag{2.8}$$

where k is the Boltzmann constant.

Exercise 2.1: The critical point for argon is 150.687 K and 4.863 MPa. What are the dimensionless critical temperature T_c^* and pressure P_c^* using the Lennard-Jones parameters $\epsilon/k = 120$ K and $\sigma = 0.34$ nm?

Exercise 2.2: What is the characteristic timescale $\sqrt{m\sigma^2/\epsilon}$ for argon using the parameters above?

Exercise 2.3: Non-dimensionalize the equation of motion (equation 1.5) with the characteristic length σ, energy ϵ and mass m.

2.2 Run the simulation

Return to the Jupyter notebook. Now that our simulation is initialized, we can run it. We will calculate the average pressure and temperature from the simulation. We start by creating Python lists to store the pressure, temperature, and time step so we can plot these later. We also initialize the average pressure and temperature variables. Type in the code block:

```
# initialize variables and lists
pres=[]
temp=[]
time=[]
Pavg = 0
Tavg = 0
```

Followed by:

```
# Run time steps of the simulation
for timestep in range(1,1001):

    # advance simulation one timestep
    sim.move()

    # only average after an equilibration time
    if timestep > 100:
        # production stage
        Pavg += (sim.P-Pavg)/(timestep+1)
        Tavg += (sim.T-Tavg)/(timestep+1)
    else:
        # equilibration stage
        Pavg = sim.P
        Tavg = sim.T

    # print timestep, ke, pe, total e, Tavg, Pavg
    if timestep%50==0: # only print every 50 timesteps
        # timestep, <ke>, <pe>, <e>, T, P, Tavg, Pavg
        print("%4d␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f" %
              (timestep,sim.ke/sim.N,sim.pe/sim.N,
                (sim.ke+sim.pe)/sim.N,sim.T,sim.P,Tavg,Pavg))

    pres.append(sim.P)
    temp.append(sim.T)
    time.append(timestep)
```

When you execute this block, you should see text like the following. You won't get exactly the same numbers, but something similar.

```
  50   3.198  -4.207  -1.009   2.132   3.365   2.132   3.365
 100   3.167  -4.177  -1.010   2.111   3.393   2.111   3.393
 150   3.214  -4.224  -1.010   2.143   3.246   2.116   3.386
 200   3.242  -4.252  -1.010   2.161   3.264   2.126   3.361
 250   3.259  -4.270  -1.011   2.173   3.184   2.130   3.346
 300   3.181  -4.192  -1.011   2.121   3.395   2.129   3.361
 350   3.235  -4.245  -1.010   2.157   3.281   2.126   3.379
 400   3.041  -4.050  -1.010   2.027   3.997   2.126   3.377
 450   3.262  -4.272  -1.009   2.175   3.201   2.127   3.385
 500   3.124  -4.133  -1.009   2.083   3.613   2.129   3.377
 550   3.172  -4.182  -1.010   2.115   3.407   2.130   3.370
 600   3.188  -4.198  -1.010   2.126   3.367   2.129   3.370
 650   3.087  -4.096  -1.009   2.058   3.814   2.127   3.381
 700   3.130  -4.140  -1.009   2.087   3.708   2.126   3.400
 750   3.180  -4.190  -1.010   2.120   3.446   2.126   3.398
 800   3.170  -4.179  -1.009   2.113   3.377   2.127   3.387
 850   3.141  -4.151  -1.010   2.094   3.619   2.126   3.391
 900   3.130  -4.139  -1.010   2.086   3.616   2.127   3.387
 950   3.224  -4.233  -1.009   2.149   3.311   2.126   3.394
1000   3.206  -4.215  -1.009   2.137   3.283   2.127   3.390
```

The columns are the time step, instantaneous kinetic energy, potential energy, total energy, temperature, and pressure, and the temperature and pressure averaged to that point. Notice how the fourth and seventh columns become fairly steady, while the other numbers fluctuate.

2.3 Analyze the results

At the end of the run, we have an average dimensionless temperature $T^* = 2.127$ (more on dimensionless values later) and a dimensionless pressure $P^* = 3.390$. Because we stored these values in the lists `pres` and `temp` in our notebook, we can calculate the average and standard deviation, skipping the first hundred or so time steps. (The reason for this will become apparent in a moment.)

```
# A different way to calculate the average pressure and temperature
# with standard deviations

import numpy as np

print("Pressure:␣␣␣␣{:.2f}+/-{:.2f}"
      .format(np.mean(pres[200:]),np.std(pres[200:])))
```

```
print("Temperature:␣{:.2f}+/-{:.2f}"
      .format(np.mean(temp[200:]),np.std(temp[200:])))
```

```
Pressure:    3.40+/-0.23
Temperature: 2.13+/-0.04
```

Make a plot! Add this code cell to your notebook:

```
import matplotlib.pyplot as plt

# Plot pressure and temperature
fig, ax = plt.subplots()
ax.plot(time,pres,label="P*")
ax.plot(time,temp,label="T*")
ax.set_xlabel("time␣step")
ax.legend(frameon=False)
ax.set_xscale('linear')
plt.title("Dimensionless␣instantaneous␣pressure␣and␣temperature")

plt.show()
```

When you run it, do you see a plot similar to figure 2.1?

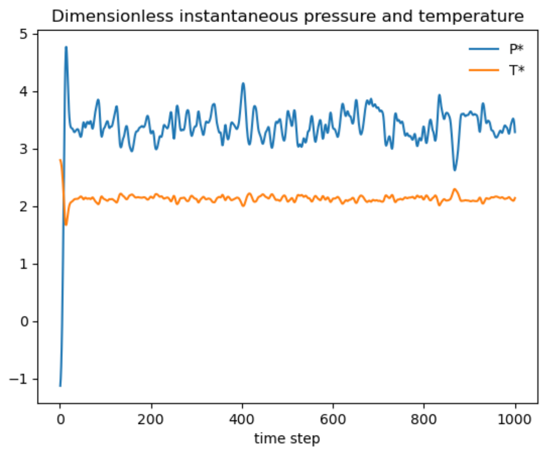

Figure 2.1: Plotted output from the Jupyter notebook.

It is clearer now how the pressure and temperature each fluctuate around an average value. Those steady values represent our simulation's equilibrium. Also notice the initial jump in the traces of the pressure and temperature. The starting point of the simulation is not at equilibrium. The simulation equilibrates during the initial period of its run.

Exercise 2.4: Run the simulation under the same conditions multiple times. How does $\langle P^* \rangle$ and $\langle T^* \rangle$ vary? How does the average total energy vary?

Exercise 2.5: Plot the pressure and temperature but focus on the first 100 time steps. How do these properties behave at the beginning of the simulation?

Exercise 2.6: Examine the dependence of $\langle P^* \rangle$ and $\langle T^* \rangle$ as you change N from $N^{1/3} = 4$ to 10 (that is, N from 64 to 1000 molecules).

Chapter 3

How it works

In this chapter, we will go through the simulation code step-by-step to learn how it works.

3.1 Initialization

Look at `CHEG231MD.py` and you should see how the simulation object initializes. This is the block of code that begins with

```
    def __init__(self, nn, rho, vmax):
```

The main thing it needs to do is start the simulation in a known state; that is, to set the initial particle locations and velocities. In this simulation:

- particles start on a cubic lattice
- the velocities are assigned a random fraction of the maximum
- these random velocities are corrected to eliminate net flow

What follows are a few of the code blocks in `CHEG231MD.py`.

3.1.1 Volume and simulation length

We imagine the molecules are in a box with sides $L \times L \times L$. In fact, the simulation size depends on the density and number of particles that we choose. Then $V = N/\rho$ and $L = V^{1/3}$. So, first, variables like the number of particles and the box dimensions are set based on ρ^* and $N^{1/3}$:

```
        # initial variables
        self.N = nn**3  # total number of particles
        self.rho = rho
        self.vol = self.N/rho # volume of the simulation
        self.L = self.vol**(1/3) # size of the simulation
        self.dL = self.L/nn # initial cubic array lattice size
```

Notice that the last variable, `dL`, sets the spacing for the initial cubic array.

Next, arrays for the particle positions and acceleration are created and other state variables initialized:

```
        # initialize position and acceleration arrays
        self.x = np.zeros((self.N,3)) # init N by 3 array of positions
        self.xnew = np.zeros((self.N,3)) # init N by 3 array of
            positions
        self.a = np.zeros((self.N,3)) # init N by 3 array of
            accelerations

        # state of the simulation
        self.pe = 0 # total potential energy
        self.ke = 0 # total kinetic energy
        self.vir = 0 # virial coef
        self.T = 1 # temperature
        self.P = 0 # pressure
```

3.1.2 Initial positions

A loop sets the initial particle positions on a cubic lattice:

```
        # create a cubic array of particles
        particle = 0
        for i in range(0, nn):
            for j in range(0,nn):
                for k in range(0,nn):
                    self.x[particle,X]=(i+0.5)*self.dL
                    self.x[particle,Y]=(j+0.5)*self.dL
                    self.x[particle,Z]=(k+0.5)*self.dL
                    particle += 1
```

It may seem strange to start the simulation off on a lattice, but this highly non-equilibrium structure will rapidly relax through the initial time steps of the simulation.

3.1.3 Time step

One of the key parameters in the initialization is the simulation time step.

```
        self.dt = 0.005 # dimensionless time step (0.005 is common)
        self.dt2 = self.dt*self.dt
```

A typical value is `dt = 0.005`. The time step is chosen to avoid wasting calculation time on very tiny displacements (fractions of σ). Yet `dt` must be small enough that unphysical displacements, like molecules overlapping, can be avoided.

Be careful! If you are importing the simulation into a Jupyter notebook and you change the time step value in `MDCHEG231.py`, you may need to restart the Jupyter kernel for it to take effect.

3.1.4 Initial velocities

The equations of motion are second-order in position, so we need two sets of initial conditions. We randomly assign velocities based on a maximum value. Because the randomly assigned velocities do not preclude the chance that there could be a bulk flow (a non-zero velocity on average in the simulation box), we calculate a velocity of the center of mass in the simulation and subtract this from each particle velocity.

```
       # assign random velocities from uniform distribution (units?)
        self.v = vmax*np.random.rand(self.N,3) # init N x 3 array of
            velocities

        # normallize the velocities s/t net velocity is zero
        vcm = np.sum(self.v,axis=0)/self.N # 1x3 array of <vx>, <vy>, <
            vz>
        for particle in range(0,self.N):
            self.v[particle,:] -= vcm

        # set the initial accelerations
        self.accel()
```

At the end of the velocity initialization, we calculate the forces acting between the molecules, so we will discuss that calculation next before describing the integration scheme.

3.2 Force calculation

The method `self.accel()` in the `MDsimulation` class calculates the forces between the molecules.

```
    # Non-JIT-compiled class method
    def accel(self):
        """
        Acceleration calculated using F = ma
        """
        self.a = np.zeros((self.N, 3))  # Zero out all accelerations
        self.pe = 0
        self.vir = 0

        # Call the JIT-compiled function to calculate acceleration
        self.pe, self.vir = accel_jit(self.N, self.x, self.a, self.L)
```

The method passes its parameters to a separate `accel.jit` function, which uses the `numba` package to perform just-in-time compilation of the code. Separating the code into the method and a compiled function speeds up the simulation considerably.

```
# JIT-compiled function to calculate acceleration
# Note: numba works by "pass in reference" with numpy arrays,
# so self.a values are modified and used in self.move()
# in addition to returning the values of potential energy and
# the virial factor
@jit(nopython=True, parallel=False)
def accel_jit(N, x, a, L):
    pe = 0
    vir = 0

    for i in range(0, N - 1):
        for j in range(i + 1, N):
            r2 = 0
            dx = x[i, :] - x[j, :]  # 1x3 array
            # Apply minimum image convention over periodic boundary
                conditions
            for k in range(3):  # Iterate over X, Y, Z
                if abs(dx[k]) > 0.5 * L:
                    dx[k] -= np.sign(dx[k]) * L
                r2 += dx[k] * dx[k]

            # Calculate the force, potential, and virial contribution
            forceri, potential, virij = LJ(r2)
            a[i, :] += forceri * dx
            a[j, :] -= forceri * dx
            pe += potential
            vir += virij

    return pe, vir
```

`accel.jit` calls another compiled function `LJ` that evaluates the Lennard-Jones potential

```
# JIT-compiled function to calculate interaction
@jit(nopython=True)
def LJ(r2):
    """
    Compute force and potential from LJ interaction
    r2 is r^2, forceri returns force/r
    """
    r2i = 1/r2
    r6i = r2i**3
    potential = 4*(r6i*r6i-r6i)
    virij = 48*(r6i*r6i-0.5*r6i)
    forceri = virij*r2i
    return forceri, potential, virij
```

Because the pairwise interaction (or forces) between all particles is evaluated, the function also returns the contributions to the potential energy and pressure, which we will discuss below, along with details that handle the simulation boundary conditions.

Exercise 3.1: After you have the simulation running, try moving the code in the `accel_jit` function into the `self.accel()` method and see how slow it gets!

3.3 Verlet algorithm

As we learned in the introduction, the heart of a molecular dynamics simulation is a numerical integration of Newton's equations of motion. At each time step in the simulation, the forces between molecules are calculated from the interaction potential and the positions and velocities updated.

We use the Verlet velocity algorithm, which is a predictor-corrector method of numerical integration. Each time step Δt updates the position of every molecule from its previous location, $\mathbf{x}_i(t)$,

$$\mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{\mathbf{F}_i(t)}{2m}(\Delta t)^2. \tag{3.1}$$

The velocity is updated at the same time with

$$\mathbf{v}_i(t+\Delta t) = \frac{\mathbf{F}_i(t+\Delta t) + \mathbf{F}_i(t)}{2m}\Delta t. \tag{3.2}$$

Once the positions and velocities are calculated, the values of the average kinetic energy, temperature, and pressure are updated. Below the Verlet velocity algorithm is shown in the `move` method form `CHEG231MD`.

```
    def move(self):
        """
        Verlet velocity algorithm
        """
        # find new positions
        self.xnew = self.x + self.v*self.dt + self.a*self.dt2/2.

        # apply periodic boundary conditions
        beyond_L = self.xnew[:,:] > self.L # N x 3 Boolean array
        self.xnew[beyond_L] -= self.L
        beyond_zero = self.xnew[:,:] < 0
        self.xnew[beyond_zero] += self.L

        self.x = self.xnew   # update positions
        self.v = self.v + self.a*self.dt/2 #half update velocity

        # call accelerate
        self.accel()

        # finish velocity update
        self.v = self.v + self.a*self.dt/2
```

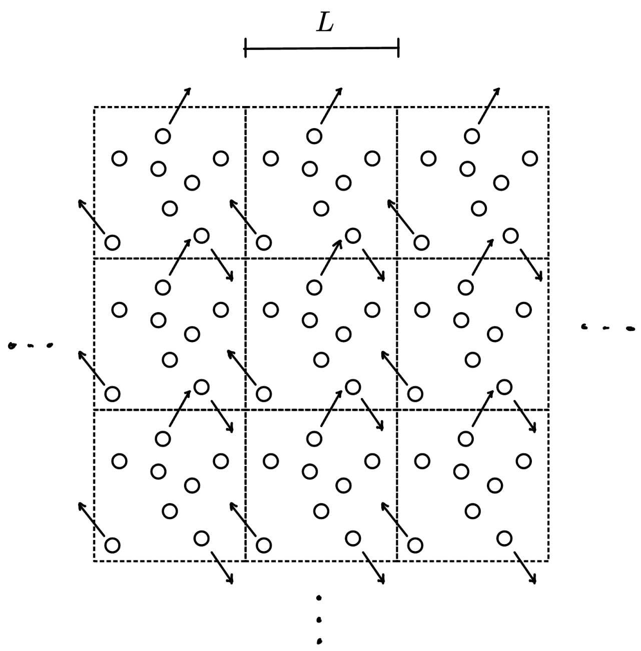

Figure 3.1: A representation of the periodic boundary conditions for a simulation with $N = 8$ molecules. The central cell represents the simulation box. Molecules passing through the boundary appear on the opposite side.

3.4 Boundary conditions

Our simulation uses periodic boundary conditions, which are represented in figure 3.1. This will minimize surface effects and enable us to simulate a much larger system with fewer molecules. When a molecule leaves the simulation cell on one side, it appears on the other side. Each molecule sees the "bulk" material from its reference frame.

3.4.1 Movement with periodic boundaries

After new locations are calculated, the code checks whether the molecule moved beyond the boundary. If it did, then the location is updated to appear on the opposite side.

```
        # find new positions
        self.xnew = self.x + self.v*self.dt + self.a*self.dt2/2.

        # apply periodic boundary conditions
        beyond_L = self.xnew[:,:] > self.L # N x 3 Boolean array
        self.xnew[beyond_L] -= self.L
        beyond_zero = self.xnew[:,:] < 0
```

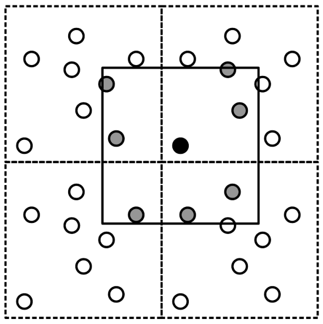

Figure 3.2: A nearest image calculation for a simulation with $N = 8$. The interactions are calculated for molecule i (black) with the closest representations of the other molecules, shown in gray.

```
        self.xnew[beyond_zero] += self.L
```

3.4.2 Nearest image interactions

Using periodic boundary conditions, we have to be careful not to calculate interactions with a molecule and its image, or multiple interactions with the same neighbor. In the code, you will notice that we calculate interactions with the "nearest image" of another molecule, which might require calculating an interaction across the periodic boundary rather than directly inside the simulation box. This is illustrated in figure 3.2. In the `accel.jit` function, we see the code that implements the nearest-neighbor interaction calculation:

```
            dx = x[i, :] - x[j, :]  # 1x3 array
            # Apply minimum image convention over periodic boundary
                conditions
            for k in range(3):  # Iterate over X, Y, Z
                if abs(dx[k]) > 0.5 * L:
                    dx[k] -= np.sign(dx[k]) * L
                r2 += dx[k] * dx[k]
```

Periodic boundary conditions have several limitations. We cannot study the spatial correlations or fluctuations that occur on wavelengths greater than the box size L, like density fluctuations that occur near a critical point. Similarly, the interactions between electric charges or dipoles, which may be long-ranged relative to L, are also more difficult to implement. Finally, there may be system size effects in the properties we calculate.

3.5 Calculating system properties

At each time step, our simulation reports the kinetic energy, potential energy, temperature and pressure. The instantaneous kinetic energy, temperature, and pressure are performed at the end of the `move` method in `CHEG231MD`:

```
        # update properties
        self.ke = 0.5*np.sum(self.v*self.v)
        self.T = 2*self.ke/self.N/3
        self.P = self.T*self.rho + self.rho/self.N*self.vir/3
```

Below I describe the calculations that the simulation is performing.

3.5.1 Kinetic energy

We calculate the total (extensive) kinetic energy of the simulation by summing the kinetic energy of each molecule,

$$\langle \text{k.e.} \rangle = \frac{1}{2} \sum_{i=1}^{N} m v_i^2. \tag{3.3}$$

3.5.2 Potential energy

Similarly, to calculate the total potential energy, we sum over all of the pair interactions,

$$\langle \text{p.e.} \rangle = \sum_{i=1}^{N-1} \sum_{j>i}^{N} u_{ij} \tag{3.4}$$

The calculation is performed in the `accel` routine as we calculate the pair interactions.

3.5.3 Total energy

The total energy of the simulation is simply the sum of the kinetic and potential energies,

$$\langle E \rangle = \langle \text{k.e.} \rangle + \langle \text{p.e.} \rangle. \tag{3.5}$$

This is an extensive quantity. The intensive total energy is calculated by dividing by the number of molecules, N. In our simulation, $\langle E \rangle$ should be constant, although some small fluctuations may occur due to the finite numerical precision of the calculation. We can use the energy to monitor the simulation's accuracy. If the energy drifts or fluctuates, it could indicate an error in the simulation.

3.5.4 Temperature

We calculate the instantaneous temperature of the simulation from the average kinetic energy of the molecules,

$$\langle T \rangle = \frac{m}{3Nk} \sum_{i=1}^{N} \langle v_i^2 \rangle. \tag{3.6}$$

3.5.5 Pressure

We will use the following expression to calculate the pressure,

$$P = \rho k T + \frac{1}{3V} \left\langle \sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i \right\rangle \tag{3.7}$$

which is derived from the virial theorem. [6] Here, $\mathbf{r}_i$ is the position vector of molecule i and $\mathbf{F}_i$ is the total force acting on it. The virial (right-most) term in equation 3.7 can be rewritten as

$$\sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i = \sum_{i=1}^{N} \sum_{j>i}^{N} r_{ij} f_{ij}. \tag{3.8}$$

This is the same order of evaluation that we use when we calculate the interaction potential and forces, so we can update the virial factor in the same routine (see section 3.2). The variable `virij` represents the contributions of this calculation.

Now that we understand some of the basics of molecular dynamics simulations, we will examine the Jupyter notebook we ran at the start of this document a little more closely.

Chapter 4

Back to the simulation

In this chapter we will break down our earlier run in the Jupyter notebook and use it to investigate the capabilities of our simulation and to calculate thermodynamic properties of the Lennard-Jones fluid.

4.1 The main loop

After initializing a simulation, we simply created a loop for a number of time steps. At each time step, we calculated the position, velocity, and acceleration of the molecules by executing `sim.move()`. This is the simulation class method that we introduced in section 3.3.

```
# Run time steps of the simulation
for timestep in range(1,1001):

    # advance simulation one timestep
    sim.move()
```

In the code block above, we have a **`for`** loop that runs the simulation for 1000 time steps.

Exercise 4.1: With 1000 time steps, what is the total simulated time in seconds for argon (molecular mass 39.792) and Lennard-Jones parameters $\epsilon/k = 120$ K and $\sigma = 0.34$ nm?

4.1.1 Equilibration and production

Remember, when we start the simulation it is in a highly non-equilibrium (but simple to implement) configuration—a cubic lattice. We use an initial number of time steps to allow it to relax to equilibrium. After this *equilibration* stage, we can begin averaging the instantaneous temperature and pressure during the *production* stage. In the code block below, we use the first 100 out of 1000 time steps to equilibrate the simulation.

```
    # print time step, ke, pe, total e, Tavg, Pavg
    # only average after an equilibration time
    if timestep > 100:
        # production stage
        Pavg += (sim.P-Pavg)/(timestep+1)
        Tavg += (sim.T-Tavg)/(timestep+1)
    else:
        # equilibration stage
        Pavg = sim.P
        Tavg = sim.T
```

4.2 Generating output

As the simulation runs, we monitor it by printing the instantaneous time step, kinetic energy, potential energy, total energy, temperature, and pressure, and the average temperature and pressure:

```
    # print time step, ke, pe, total e, Tavg, Pavg
    if timestep%50==0: # only print every 50 timesteps
        # timestep, <ke>, <pe>, <e>, T, P, Tavg, Pavg
        print("%4d␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f␣␣%6.3f" %
              (timestep,sim.ke/sim.N,sim.pe/sim.N,
                (sim.ke+sim.pe)/sim.N,sim.T,sim.P,Tavg,Pavg))
```

We do this every 50 time steps to avoid slowing the simulation down too much.

The final steps in our Jupyter routine append the instantaneous pressure and temperature to the lists we defined earlier. As we saw in chapter 2, we can use this to plot the evolution of these variables with time after the simulation runs.

```
    pres.append(sim.P)
    temp.append(sim.T)
    time.append(timestep)
```

We used these variables in chapter 2 to plot the instantaneous pressure and temperature in figure 2.1.

Exercise 4.2: Why does temperature fluctuate in the simulation? Plot the standard deviation of the temperature as a function of N.

Exercise 4.3: Try different time steps. What do you observe? Can you break the calculation?

Exercise 4.4: Compare the calculated average pressure to the ideal gas pressure for the same conditions, ρ and T.

4.3 Maxwell-Boltzmann distribution of speed

The Maxwell-Boltzmann distribution of speed is

$$f(v) = 4\pi \left(\frac{m}{2\pi kT}\right)^{3/2} v^2 e^{-mv^2/2kT}. \tag{4.1}$$

In this probability distribution function, v is the molecular speed, m is the molecule mass, and T is the absolute temperature. Our equilibrated simulation should generate speeds with this distribution function!

Exercise 4.5: Modify the simulation to create a histogram of the absolute value of the velocities. Average the histogram over all of the production time steps. Compare your histogram to the Maxwell-Boltzmann distribution for speed, equation 4.1.

4.4 PVT diagrams

4.4.1 Pressure-temperature plot

We should be able to verify Gay-Lussac's "law" with the simulation. Stated succinctly:

> The pressure exerted by a given mass and constant volume of a gas on the sides of its container is directly proportional to its absolute temperature.

Exercise 4.6: Run the simulation at different maximum velocity values while keeping the density constant and plot the average pressure versus average temperature.

4.4.2 Pressure-volume plot

It would be interesting to plot isotherms of the pressure versus density (molar volume) to compare with mechanical equations of state. But it is a little more complicated.

One of the challenges in our simulation is the uncertainty in the initial kinetic energy, since the values are randomly assigned, albeit with a limit and uniform distribution. Nonetheless, the equilibrated temperature can vary quite a bit. There is also a coupling between the density and equilibrated temperature. When we increase or decrease the simulation density, we change its initial potential energy independent of the kinetic energy.

Exercise 4.7: Add code to the equilibration phase that rescales the velocities to match a desired temperature. Use this to create a $P^* - \rho^*$ isotherm for $T^* = 2.0$ for non-dimensional densities between 0.1 and 0.9. Next, compare your isotherm to those calculated using the ideal gas law and the van der Waals equation of state using the parameters for argon. *Hint: Use equations 2.7 and 2.8 to convert between dimensional and non-dimensional variables.*

Chapter 5

Visualization

What is really going on in our simulation? We get numbers out—pressures and temperatures—that make sense from our understanding of thermodynamics. But what does it really look like? How are those molecules zipping around in our cybersystem? Fortunately, there are some good open-source packages available to help us see and interpret our simulations. We will use VMD in the following example.

5.1 Export position data

We begin by exporting our data as an Extended XYZ format. This is a text file that records the position of each atom at each time step. Each frame starts with the number of atoms, followed by a comment line (often containing the time step or other metadata), and then the atomic coordinates.

```
5
Timestep: 0
Ar  0.000   0.000   0.000
Ar  0.757   0.586   0.000
Ar -0.757   0.586   0.000
Ar  1.000   1.000   1.000
Ar  1.757   1.586   1.000
5
Timestep: 1
Ar  0.010   0.000   0.000
Ar  0.767   0.586   0.000
Ar -0.747   0.586   0.000
Ar  1.010   1.000   1.000
Ar  1.767   1.586   1.000
...
```

This format is not used for large MD simulations, but it should be fine for the smaller systems we are calculating.

The position of each molecule in our simulation is stored in `sim.x`. This is an $N \times 3$ array. We have to add the molecule name on each line in addition to the number of molecules and frame time step. For example,

```
# Format function to add 'Ar' at the beginning of each line
def custom_formatter(arr):
    return ["Ar␣" + "␣".join(f"{x:.6f}" for x in row) for row in arr]

def write_XYZ():
    with open('MDrun.xyz', 'a') as f:
        f.write(f"{sim.N}␣\n")
        f.write(f"Timestep:␣{timestep}␣\n")
        # Append to the existing text file with custom format
        for line in custom_formatter(sim.x):
            f.write(line + "\n")
```

Now calling `write_XYZ()` at each time step will append the new positions to the file. We just have to add it to the **`for`** loop after calling `sim.move()`.

```
# Run time steps of the simulation
for timestep in range(1,1001):

    # advance simulation one timestep
    sim.move()
    write_XYZ()
```

If we do not want to image the equilibration period, we can add it to the **`if-else`** statement when we average over the production steps.

Now we should have an `MDrun.xyz` file that records the position of every molecule for each time step in the directory.

5.2 Importing into VMD

Start VMD. (See section A.3 in the Appendix for information on how to download it.) Then follow these steps:

1. Load the XYZ file. In the VMD **Main** window, select **File** and **New molecule**.
2. Browse or type the path for your filename and select **Load**.
3. The Display window will begin to fill with many lines.
4. After all the frames are loaded, select **Graphics** and **Rendering**.
5. In the **Graphical Representations** window that appears, Select **VDW** under **Drawing method**. Adjust the **Sphere scale** to 0.1. Change the **Resolution** if you like.
6. In the **Main** window under **Display** toggle between the default **Perspective** and **Orthographic**.

7. Use the controls in the lower part of the **Main** window to play the animation and control its speed.

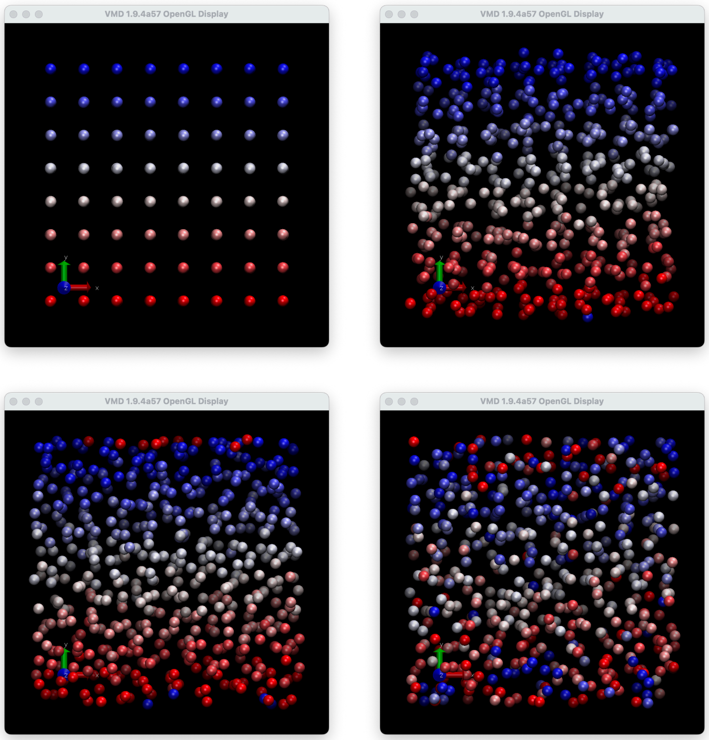
VMD 1.9.4a57 OpenGL Display
VMD 1.9.4a57 OpenGL Display
VMD 1.9.4a57 OpenGL Display
VMD 1.9.4a57 OpenGL Display

Figure 5.1: Representations of the MD simulation at time steps 0, 50, 100, and 999 for $N = 512$, $\rho^* = 0.7$, and $v^*_{\text{max}} = 5.8$. The average pressure and temperature for this run is $P^* = 3.35 \pm 0.22$ and $T^* = 2.09 \pm 0.04$.

In figure 5.1 I show four representations of the MD simulation with `VMD` spanning time steps from the start through to the end of the run. The simulation begins in the cubic lattice and quickly becomes more disordered by `timestep = 50`. By the 100th time step, we begin averaging properties. Notice several red and blue molecules on either side of the simulation box. These have passed through to the other side because of the periodic boundary conditions. By the last time step, the artificial color gradient we started with is diffuse, but remains relatively visible.

Chapter 6

Going further

This chapter has some advanced exercises and concepts.

6.1 Advanced excercises

Try these! Or at least think about how you might approach them.

Exercise 6.1: Convert the simulation to two-dimensions. Create a movie to visualize the molecular motion.

Exercise 6.2: The simulation represents the N, V, E ensemble, also known as the microcanonical ensemble. The number of molecules, volume, and energy are constant. As we saw, this is different than the N, V, T, or canonical, ensemble. Temperature fluctuates in our simulation! Implement a credible thermostat to run the simulation at constant temperature.

Exercise 6.3: There are a lot of inefficiencies in our code. Can you speed it up? How fast can you get it to go? Try simulating larger systems for longer times. But really, we should probably just run LAMMPS or GROMACS!

Exercise 6.4: Calculate the mean-squared displacement and self- and collective diffusivities.

6.2 Advanced concepts

Here we will present a few advanced concepts. The first is a matter of practical importance relating to error in the energy calculation. After this, we discuss the simulation and its relationship to entropy.

6.2.1 Energy correction

With a relatively small simulation size, it is possible that the interaction potential does not fully decay at the length $L/2$. Moreover, even in a large simulation, it would be computationally expensive to evaluate the interactions at long ranges, which would make only small contributions to the total energy. In either case, the truncation of the interaction calculation creates an error in the energy; however, we can easily correct it.

6.2.2 Entropy

Our simulation mimics reality in another key way: entropy is not directly calculable. Like the real world, we can measure (calculate) variables such as the pressure, temperature, and volume, and additionally, we have direct access to the kinetic and internal energy.

Appendix A

More simulation resources

Writing a simple MD simulation, especially in Python, is principally a pedagogical exercise. Far more powerful codes exist and can be used with a little effort. Here are a few.

A.1 LAMMPS

`https://www.lammps.org`

LAMMPS is a high-performance simulation package developed over the past three decades. It can be installed in a Conda environment as pre-built packages on Linux and macOS machines. Pre-built Windows packages also exist. See the LAMMPS documentation.

A.2 GROMACS

`https://www.gromacs.org`

GROMACS "is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles." GROMACS is primarily designed for biochemical molecules like proteins, lipids and nucleic acids. These molecules have many complicated bonded interactions. GROMACS is also used for research on non-biological systems.

A.3 VMD

`https://www.ks.uiuc.edu/Research/vmd/`

Visual Molecular Dynamics (VMD) "is a molecular visualization program for displaying, animating, and analyzing large biomolecular systems using 3-D graphics and built-in scripting."

VMD also handles data like the `xyz` file format that we write in section 5.1. We can use the program to visualize our simulations.

Relatively up-to-date downloads are available for many computing platforms, including Windows, macOS, and Linux. If you have a newer Apple computer that uses an ARM processor ("Apple Silicon" like the M1, M2, or M3 processor) you should download version 1.9.4.

Appendix B

Code

Python and its libraries are constantly evolving. Usually older code will run on later versions of Python,[1] but codes that use newer features will often cause errors with older versions.

You can check the version of Python and several libraries using the following:

```
import sys
print(sys.version)

import numpy as np
print(np.__version__)

import numba
print(numba.__version__)
```

I ran the following code with:

- python version 3.12.4 build h99e199e_1, channel defaults
- numba version 0.60.0 build py312hd77ebd4_0, channel defaults
- numpy version 1.26.4 build py312h7f4fdc5_0, channel defaults

B.1 CHEG231MD.py listing

```
"""
CHEG231MD.py
Simple MD code in Python (it's slow!) for a Lenard-Jones fluid
based on QuickBASIC code by Richard L. Rowley [1]
and FORTRAN code by Smit and Frenkel [2]

Eric M. Furst
November 2023
```

[1]Some major version changes broke a lot of older code.

```
[1] Richard L. Rowley (1994). Statistical Mechanics for Thermophysical
    Property Calculations. Prentice Hall, New York.
[2] Daan Frenkel and Berend Smit (2002). Understanding Molecular
    Simulation, 2nd ed. Academic Press, New York.

Revisions
October 2024 - sped up with numba jit of accel routine and LJ calc.

Requires:
    python (tested with ver. 3.12.4 build h99e199e_1 channel defaults)
    numba  (tested with ver. 0.60.0 build py312hd77ebd4_0 channel
        defaults)
    numpy  (tested with ver. 1.26.4 build py312h7f4fdc5_0 channel
        defaults)

Needs:
    nn - total number of particles per dimension (thus, N = nn^3)
    rho - dimensionless number density of particles (units of rho/sigma
        ^3)
    vmax - maximum velocity (units of sqrt[epsilon/m])

Next to do:
    write a 2D animated version
    a challenge for students: dimensionless variables
    can a fast r_cutoff be implemented based on
    from scipy.spatial.distance import pdist, squareform?
"""

import numpy as np
from numba import jit

X, Y, Z = 0, 1, 2 # helpful for indexing

class MDSimulation:
    """
    x - new position coordinate array (N x 3)
    v - velocity array (N x 3)
    """

    def __init__(self, nn, rho, vmax):
        """
        Initialize the simulation
        - particles start on a cubic lattice
        - velocities are assigned a random fraction of the maximum
        - velocities are corrected to eliminate net flow
        """

        # initial variables
        self.N = nn**3  # total number of particles
        self.rho = rho
        self.vol = self.N/rho # volume of the simulation
        self.L = self.vol**(1/3) # size of the simulation
        self.dL = self.L/nn # initial cubic array lattice size

        self.dt = 0.005 # dimensionless time step (0.005 is common)
        self.dt2 = self.dt*self.dt
        # initialize position and acceleration arrays
        self.x = np.zeros((self.N,3)) # init N by 3 array of positions
        self.xnew = np.zeros((self.N,3)) # init N by 3 array of
            positions
        self.a = np.zeros((self.N,3)) # init N by 3 array of
            accelerations

        # state of the simulation
        self.pe = 0 # total potential energy
        self.ke = 0 # total kinetic energy
        self.vir = 0 # virial coef
        self.T = 1 # temperature
        self.P = 0 # pressure

        # create a cubic array of particles
        particle = 0
        for i in range(0, nn):
            for j in range(0,nn):
                for k in range(0,nn):
                    self.x[particle,X]=(i+0.5)*self.dL
                    self.x[particle,Y]=(j+0.5)*self.dL
                    self.x[particle,Z]=(k+0.5)*self.dL
                    particle += 1

        # assign random velocities from uniform distribution (units?)
        self.v = vmax*np.random.rand(self.N,3) # init N x 3 array of
            velocities

        # normlize the velocities s/t net velocity is zero
        vcm = np.sum(self.v,axis=0)/self.N # 1x3 array of <vx>, <vy>, <
            vz>
        for particle in range(0,self.N):
            self.v[particle,:] -= vcm

        # set the initial accelerations
        self.accel()

    def move(self):
        """
        Verlet velocity algorithm
        """
        # find new positions
        self.xnew = self.x + self.v*self.dt + self.a*self.dt2/2.

        # apply periodic boundary conditions
        beyond_L = self.xnew[:,:] > self.L # N x 3 Boolean array
        self.xnew[beyond_L] -= self.L
        beyond_zero = self.xnew[:,:] < 0
        self.xnew[beyond_zero] += self.L

        self.x = self.xnew   # update positions
        self.v = self.v + self.a*self.dt/2 #half update velocity

        # call accelerate
        self.accel()
        # finish velocity update
        self.v = self.v + self.a*self.dt/2

        # update properties
        self.ke = 0.5*np.sum(self.v*self.v)
        self.T = 2*self.ke/self.N/3
        self.P = self.T*self.rho + self.rho/self.N*self.vir/3


    # Non-JIT-compiled class method
    def accel(self):
        """
        Acceleration calculated using F = ma
        """
        self.a = np.zeros((self.N, 3))  # Zero out all accelerations
        self.pe = 0
        self.vir = 0

        # Call the JIT-compiled function to calculate acceleration
        self.pe, self.vir = accel_jit(self.N, self.x, self.a, self.L)

# Functions are below
# We're using numba jit to make the loop of the calculation faster

# JIT-compiled function to calculate acceleration
# Note: numba works by "pass in reference" with numpy arrays,
# so self.a values are modified and used in self.move()
# in addition to returning the values of potential energy and
# the virial factor
@jit(nopython=True, parallel=False)
def accel_jit(N, x, a, L):
    pe = 0
    vir = 0

    for i in range(0, N - 1):
        for j in range(i + 1, N):
            r2 = 0
            dx = x[i, :] - x[j, :]  # 1x3 array
            # Apply minimum image convention over periodic boundary
                conditions
            for k in range(3):  # Iterate over X, Y, Z
                if abs(dx[k]) > 0.5 * L:
                    dx[k] -= np.sign(dx[k]) * L
                r2 += dx[k] * dx[k]

            # Calculate the force, potential, and virial contribution
            forceri, potential, virij = LJ(r2)
            a[i, :] += forceri * dx
            a[j, :] -= forceri * dx
            pe += potential
            vir += virij

    return pe, vir

# JIT-compiled function to calculate interaction
@jit(nopython=True)
def LJ(r2):
    """
    Compute force and potential from LJ interaction
    r2 is r^2, forceri returns force/r
    """
    r2i = 1/r2
    r6i = r2i**3
    potential = 4*(r6i*r6i-r6i)
    virij = 48*(r6i*r6i-0.5*r6i)
    forceri = virij*r2i
    return forceri, potential, virij
```

Appendix C

Math

C.1 Potential and force

If the non-dimensional pair interaction potential is

$$u^*(r) = 4\left(\frac{1}{{r^*}^{12}} - \frac{1}{{r^*}^6}\right) \tag{C.1}$$

then the force given by $\mathbf{f} = -\boldsymbol{\nabla} u$ is

$$\mathbf{f}^*(\mathbf{r}^*) = 4\left(12\frac{\mathbf{r}^*}{{r^*}^{14}} - 6\frac{\mathbf{r}^*}{{r^*}^8}\right) \tag{C.2}$$

where $\mathbf{r}^*$ is the vector between particle centers. (Remember, the asterisk indicates that it is dimensionless—scaled by σ in the case of the separation.)

You may be wondering about the factor of four in the interaction potential. This scaling makes the potential at its minimum equal to $-\epsilon$.

Bibliography

[1] B. J. Alder and T. E. Wainwright. Phase Transition for a Hard Sphere System. *The Journal of Chemical Physics*, 27(5):1208–1209, November 1957.

[2] B. J. Alder and T. E. Wainwright. Studies in Molecular Dynamics. I. General Method. *The Journal of Chemical Physics*, 31(2):459–466, August 1959.

[3] E. Fermi, P. Pasta, S. Ulam, and M. Tsingou. Studies of nonlinear problems. Technical Report LA-1940, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), May 1955.

[4] Robert Garner. Early Popular Computers, 1950 - 1970. https://ethw.org/Early_Popular_Computers,_1950_-_1970, January 2015.

[5] George Dyson. *Turing's Cathedral.* Vintage, New York, 2012.

[6] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation.* Academic Press, New York, 2nd edition, 2002.

[7] Jesse M. Kinder and Philip Nelson. *A Student's Guide to Python for Physical Modeling.* Princeton University Press, Princeton, NJ, 2nd edition, 2021.