Molecular Dynamics Simulation

Qing Zhou, s2501597

March, 2020

1 Introduction

Molecular dynamics simulation mimics the natural motions of particles in nature. The energy defined by the potential function allows the atoms to move away from and close to neighbouring atoms. And intuitively in different states particles move in a very distinctive fashion, and it is interesting to find out in what ways they differ.

The motivation of this project is to understand what atoms do in real life, assuming a given potential energy function. In order to achieve this goal we run a simulation of Argon gas atoms, and analyse the physical properties of the box in three different states, by varying the temperature and density of the box. Two different initialisation methods were used, where we focus mainly on the default initialisation with lattice configuration, additional investigation on the initialisation was also discussed and demonstrated in the very last section.

This report is laid out as follows: in section 2 we introduce the initialisation techniques and potential function used for the particle-particle interaction; in section 3 we introduce briefly the details of implementing the periodic boundary conditions. In section 4 we introduce the Verlet algorithm which is used particularly in molecular dynamics simulation, and the updating functions. In section 5 we describe the physical properties of our three different simulations, and our expectations. In the last but not least section, we discuss the implications of the results, and present our final results. In the last section, we also presented our extra investigation using a different initialisation method.

2 Initialisation

2.1 Global constants

There are a lot of properties at play during the whole course of the simulation, and we certainly want to keep record of them. A reliable way is to put all the physical constants as globals such that they do not get updated during any step. Below lists all the global constants in this simulation.

Table 1: The physical constants used in this simulation.

Constants	σ[m]	k_B[$\mathrm{m^2\,kg\,s^{-2}\,K^{-1}}$]	T_0[K]	ϵ[$\mathrm{m^2\,kg\,s^{-2}}$]	m[kg]
Values	3.405×10^{-10}	1.38×10^{23}	100	k_BT_0	6.6×10^{-26}

2.2 Potential

We assume a Lennard-Jones potential for the two point interaction, such that when the particles get too close to each other, they feel repulsive force, and when they are too far apart, they attract each other. The Lennard-Jones potential takes the following form

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

Below is an illustration of the Lennard-Jones potential.

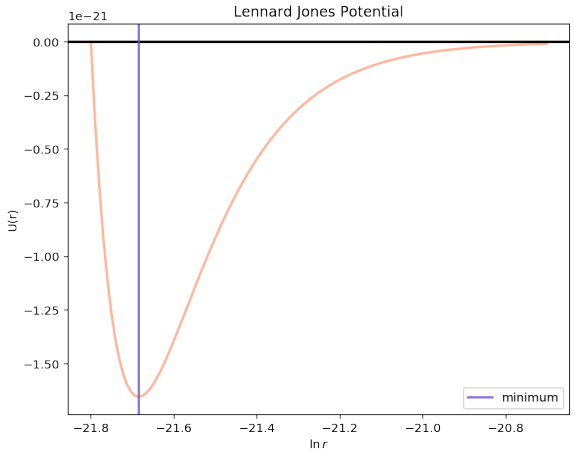


Figure 1: An illustration of the Lennard-Jones potential.

From the above plot it can be seen, the minimum of the potential is at $2^{1/6}\sigma \sim 1.12\sigma$, suggesting that for a pair of particles with such a separation, the force is neither attracting nor repulsing, which is as fixed point. Hence we can learn that to initialise the position of the particles we should avoid such a point.
The L-J potential is a relatively good approximation. Due to its simplicity, it is often used to describe the properties of gases and to model dispersion and overlap interactions in molecular models. It is especially accurate for noble gas atoms and is a good approximation at long and short distances for neutral atoms and molecules.

2.3 Position

Since we have a Lennard-Jones potential for calculating the two-point interactions, the separation between the pairs of particles can not exceed the natural unit σ, below which the potential becomes positive. Hence the positions of the particles are initialised using the lattice cell, for the fact that Argon forms face-centered lattice, where each unit cell consists of eight particles on the vertices and 6 particles on the faces. Below is an illustration of how this unit cell is being copied in space.

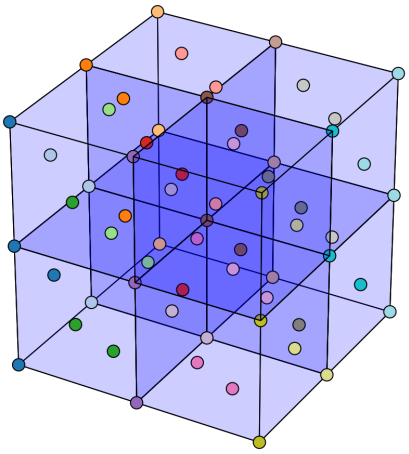


Figure 2: An illustration of a simulation box with a number of cells equals 2^3, so that there are in total 63 particles inside the box colour coded randomly. Details see 'plot_lattice.py'.

In this simulation we initialise the simulation box with a number of cells of 3^3, meaning that there are in total 108 particles inside the box. Other ways of initialising the position of the particles have also been

explored. In addition to the lattice generation technique to avoid the positive potential disaster, we can also use a somewhat more random way of generating the particles, such that we perturb the particles slightly from there lattice configuration, and rescale the positions such that, now the two point distances do not go below 1σ. Details of this generation routine please see the listing of the code in the appendix.

2.4 Velocity

The Maxwell-Boltzmann distribution corresponds to the most likely rate distribution in a collision-based system composed of a large number of non-interacting particles, in which quantum effects can be ignored. Since the molecular interactions in gas are generally quite small, the Maxwell-Boltzmann distribution provides a very good approximation of the state of the gas.
Assuming the system of interest contains a large number of particles, the fraction of the particles within a infinitesimal element of three- dimensional velocity space d^3v, centered on a velocity vector of magnitude v, is $f(v)d^3v$, in which

$$f(v)d^3v = \left(\frac{m}{2\pi k_B T}\right)^{\frac{3}{2}} e^{-\frac{mv^2}{2k_B T}} d^3v \tag{2}$$

where m is the particle mass, k_B is Boltzmann constant and T is thermodynamic temperature. For velocity in a standard Cartesian coordinate system, the element of velocity space can be written as $d^3v = dv_x dv_y dv_z$, here fv is given as a probability distribution function, normalized so that $\int f(v)d^3v$ over all velocities equals one.
The mean speed $< v >$ and root-mean speed $\sqrt{\langle v^2 \rangle}$, which is the expected value of the speed distribution and the second-order raw moment of the speed distribution, can be obtained from properties of the Maxwell distribution. For particles with same mass, the most probable speed v_p is the speed most likely to be possessed by any particles in system and corresponds to the maximum value of $f(v)$

$$\frac{df(v)}{dv} = 0 \quad \Rightarrow \quad v_p = \sqrt{\frac{2k_B T}{m}} \tag{3}$$

and the mean speed reads

$$\langle v \rangle = \int v f v dv = \sqrt{\frac{8k_B T}{m}} = \frac{2}{\sqrt{\pi}} v_p \tag{4}$$

the root mean square speed corresponding to the speed of a particle with median kinetic energy,

$$v_{rms} = \sqrt{\langle v^2 \rangle} = \left(\int v^2 f(v) dv\right)^{\frac{1}{2}} = \sqrt{\frac{3k_B T}{m}} = \sqrt{\frac{3}{2}} v_p \tag{5}$$

Meanwhile, distribution for the speed follows immediately from the distribution of the velocity vector, note that the speed is

$$v = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{6}$$

and the volume element in spherical coordinated

$$dv_x dv_y dv_z = v^2 \sin\theta dv d\theta d\phi = v^2 dv d\Omega \tag{7}$$

where θ and ϕ are the spherical coordinate angles of the velocity vector. Integration of the probability density function of the velocity over the solid angles $d\omega$ yields an additional factor of 4π. The speed distribution with substitution of the speed for the sum of the squares of the vector components:

$$f(v) = \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \left(\frac{m}{k_B T}\right)^{\frac{3}{2}} v^2 e^{-\frac{mv^2}{2k_B T}} \tag{8}$$

By using a 3d multivariate Gaussian located at $(0, 0, 0)$ and with only diagonal elements in the covariance matrix Σ, we can have our velocities distributed according to the rule described above. For details of the velocity generation routine please refer 'Argon_simulation.py'.

3 Periodic Boundary Condition

During the whole simulation, periodic boundary condition is applied. This is because we want our box volume to be constant, and the number of particles to be the same throughout this simulation, so that the energy is conserved and no heat nor matter is exchanged with the outside reservoir. In this setup, if we have a high velocity particle flying outside of the box, it should automatically reappear on the other side of the box. This setup applies to forces as well. In order to compute forces under the assumption of a box with periodic boundary condition, we make 27 copies of the original simulation box, and where each contain particles with the same velocities. Using this extended box which is 3^3 the size of the original one, we compute the two point forces and update the positions and velocities of the particles. Below is an illustration of how the original simulation box is copied and extended.

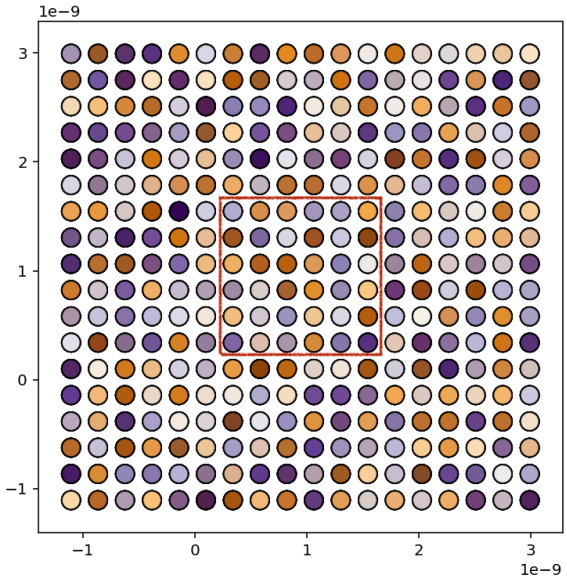


Figure 3: An illustration of the extended box, now colour coded by the velocities of the particles.

The extended box is plotted in the projected $xy-$plane, and the red square outlines the original projected simulation box. In this way, the particles on the vertices of the simulation box can also feel the forces from the other end of the vertices, thus updating the velocities and positions accordingly. Notice that the extension is made in all three dimensions and above is just an illustration how the extension works.

4 Verlet Algorithm

To update the positions and velocities of the particles we implement Verlet algorithm. To find the two-point interaction potential, we first need to find the update rule for the eolvution of the potential, which is given by

$$\boldsymbol{F}(\boldsymbol{x}(t+h)) = -\nabla U(\boldsymbol{x}(t+h)) \tag{9}$$

where U can be found by (1). For each particle, we feed the function $\boldsymbol{F}$ an array of vectors which are the subtraction of the positions of the surrounding particles and the current particle. Then $\boldsymbol{F}$ is effectively the sum of all the divergence of the potential given by all the surrounding particles. The position vector and velocity vector for each particle are updated as

$$\begin{aligned} \boldsymbol{x}(t+h) &= \boldsymbol{x}(t) + h\boldsymbol{v}(t) + \frac{h^2}{2m}\boldsymbol{F}(\boldsymbol{x}(t)) + \mathcal{O}(h^3) \\ \boldsymbol{v}(t+h) &= \boldsymbol{v}(t) + \frac{h}{2m}\boldsymbol{F}(\boldsymbol{x}(t+h)) + \boldsymbol{F}(\boldsymbol{x}(t)) \end{aligned} \tag{10}$$

Hence if initialised with what was described above, we can expect the lattice configuration of the simulation box to be randomised. Below shows some snapshots during one simulation.

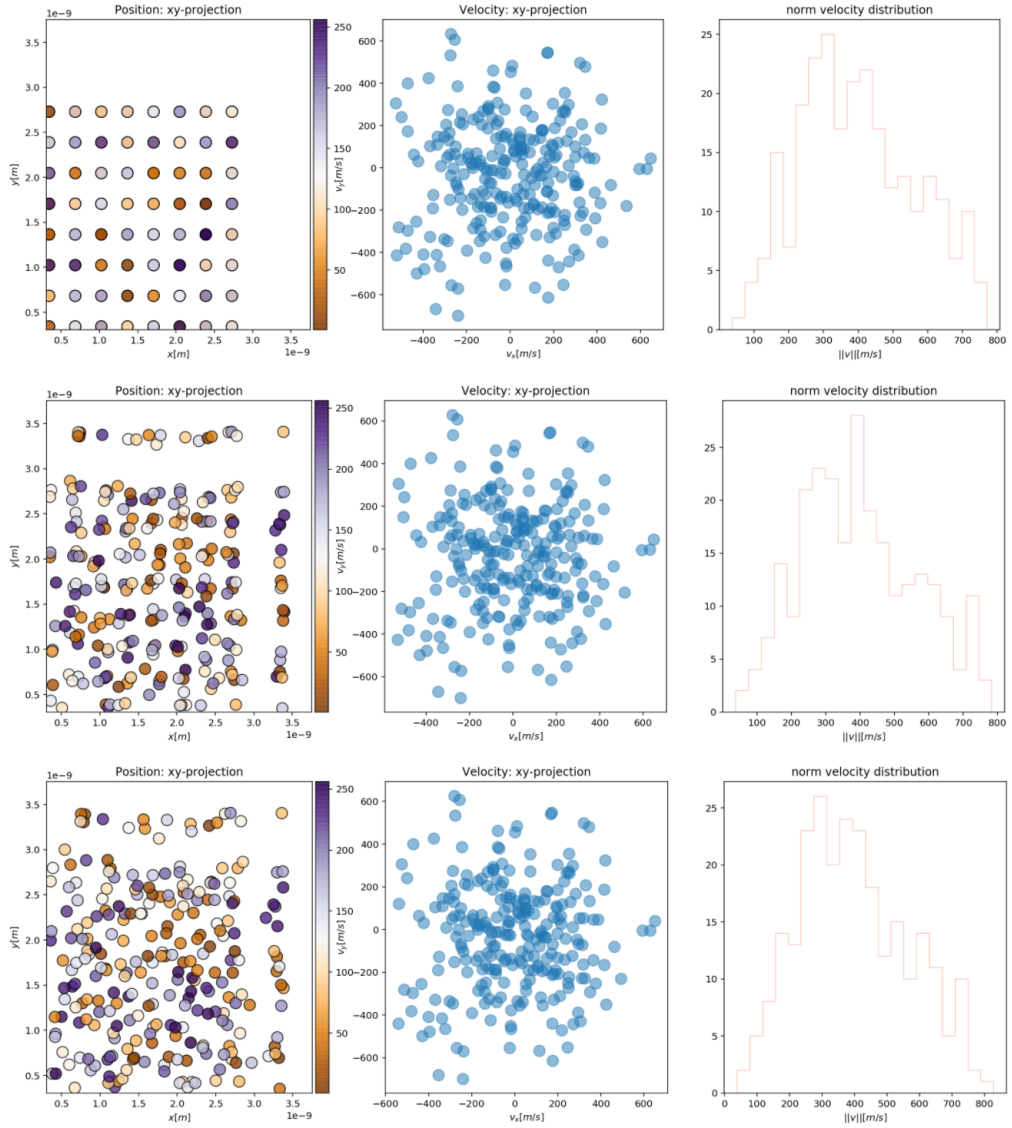


Figure 4: Snapshots for one of the simulation runs(Simulation 2, details see Table2 and 3). The first column shows the particles on the projected xy-plane, where the particles are colour coded by there velocities; the second column shows the distribution of the velocities on the projected $v_x v_y$-plane; and the third column shows the histogram of the norm of the velocity vectors. Notice that the empty spaces are there to ensure when making the extended box, two particles do not overlap.

The snapshots were taken every 25 loops, with the first row being the initial state. It can be seen from this simulation, the symmetry given by the lattice configuration is soon broken, and particles more randomly distributed.

5 Pressure and Correlation Function

In a box of particles, pair correlation function $g(r)$ describes how density varies as a function of distance from a reference particle. The pair correlation function is usually determined by calculating the distance between all particle pairs and binning them into a histogram. The histogram is then normalized with respect to an ideal gas, where particle histograms are completely uncorrelated, which means $g(r) = 1$. For three dimensions, this normalization is the number density of the system ρ multiplied by the volume of the spherical shell, which symbolically can be expressed as $\rho 4\pi r^2 dr$.

Table 2: Simulation configurations for three different states of Argon.

simulation configurations			
Simulation	State	ρ	T
Simulation 1	Gas	0.3	3
Simulation 2	Liquid	0.8	1
Simulation 3	Solid	1.2	0.5
Number of particles: 108			

Table 3: Physical properties for three different states of Argon.

Physical properties				
Simulation	State	ρ[kg m^{-3}]	T[K]	V[m^3]
Simulation 1	Gas	5.833×10^2	300	2.896×10^{-26}
Simulation 2	Liquid	1.556×10^3	100	1.086×10^{-26}
Simulation 3	Solid	2.333×10^3	50	7.241×10^{-27}

In this case, the pair correlation function can be computed by

$$g(r) = \frac{2V}{N(N-1)} \frac{\langle n(r) \rangle}{4\pi r^2 \Delta r} \tag{11}$$

where V is the volume of simulation cell, and N is the particle number. $\langle n(r) \rangle$ denotes the average over many configurations.
The radial distribution function is fundamentally important since it can be used to link the microscopic details to macroscopic properties. Hence, we can use the following formula to obtain the pressure of the system:

$$\frac{p}{\rho k_B T} = 1 - \frac{1}{3Nk_BT} \left\langle \sum_i \sum_{j>i} r_{ij} \frac{\partial U(r_{ij})}{r} \right\rangle - \frac{\rho}{6k_BT} \int \mathbf{r} \frac{\partial U}{\partial \mathbf{r}} g(\mathbf{r}) d\mathbf{r} \tag{12}$$

The first term of Eq.12 presents the ideal gas, the second term is the time average of the virial. The last term is a correction term which takes into account the effect on the pressure of the tail of the potential which has been neglected in the dynamics as well as in this simulation.
In order to analyse the properties of the Argon of three different states, we initialise as follows

6 Discussion and Conclusion

In order to set up the box as listed in Table 2, we first calculate the density in the liquid state of Argon, which is $1.3954 \times 10^3\,\mathrm{kg\,m^{-3}}$. Hence with the given ratio we found the density and temperature in the other two states to be The trajectories for the three simulations are shown as follows. Panel figure 3 x 4!

Table 4: Mean pressure with error for three different states of Argon.

Pressure values[Pa]			
Simulation	State	$\bar{p}$	σ_p
Simulation 1	Gas	3.66×10^7	582
Simulation 2	Liquid	3.25×10^7	46
Simulation 3	Solid	2.44×10^7	91

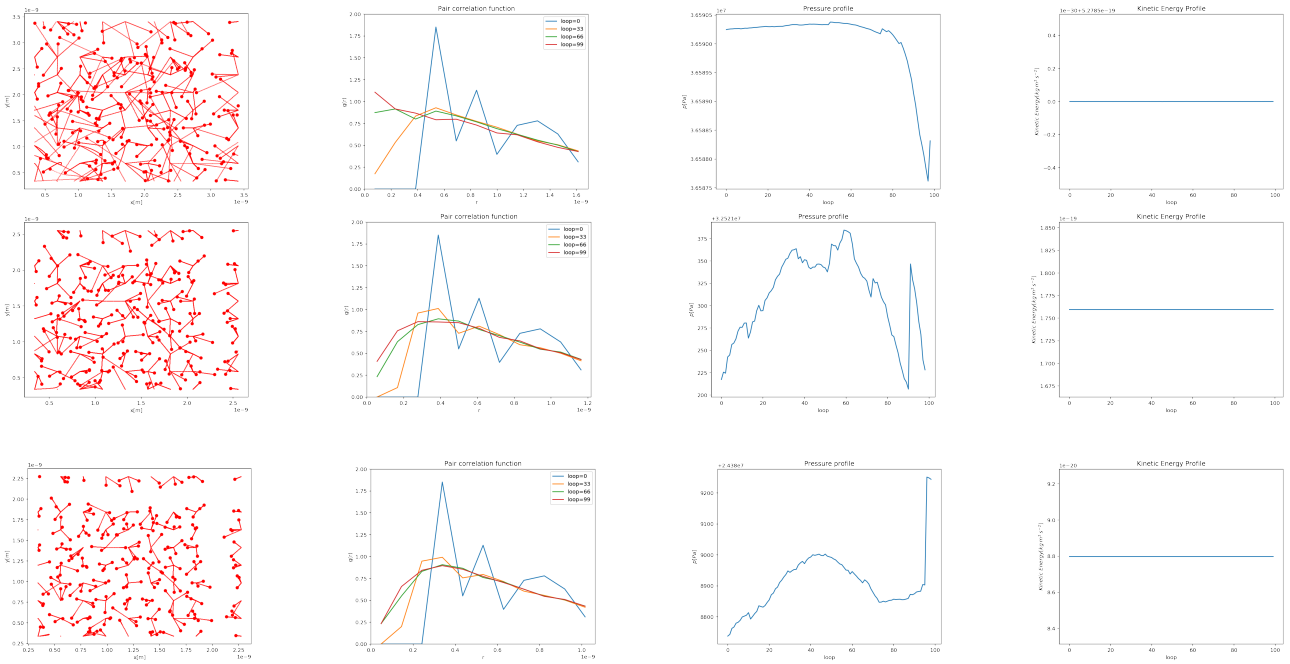


Figure 5: From top to bottom: Simulation 1, Simulation 2, Simulation 3. From left to right: trajectories, correlation function, pressure profile, and conservation of kinetic energy. To get an impression of what this means on the expected values for pressure, we do the dimensional analysis. The SI unit for pressure is Pascal, which is equivalently $\mathrm{kg}\,(\mathrm{m} \cdot \mathrm{s}^2)$, which is equivalent to the unit of $\frac{k_B T}{m}$, such that

$$[p] = \left[\frac{k_B T}{m}\right] = \frac{kg}{m \cdot s^2} \tag{13}$$

The Boltzmann constant is on the order of 10^{-23}, the temperature is on the order of 10^2 and density 10^3 and mass 10^{-25}, making the pressure on the order of 10^7, given that the correction term

$$\frac{1}{3Nk_BT}\left\langle \sum_i \sum_{j>i} r_{ij} \frac{\partial U(r_{ij})}{r} \right\rangle \tag{14}$$

is negligible. Hence we can expect our calculated pressure to be on the order of 10^7 Pa, and the values from three states should be roughly on the same order. It can be seen from the above panel, that the three different states have very distinctive correlation functions. For the gas state, the lattice configuration is quickly broken, and the equilibrated result (as depicted by the red solid line) shows that the correlation function goes from ~ 1 and then asymptotically decays to zero. As for the liquid state, the correlation function starts at ~ 0.5, then peaks at ~ 1 and then decreases. For the solid state, the equilibrated result shows that there is still a strong correlation at distances $r = (\sqrt{2}/2)l, l, 2l$ where l is the side length of the unit lattice cell. These correlations arise because the symmetry of the lattice configuration is not yet fully broken in solid state, yet slightly perturbed. And in this situation particles are oscillating around there initial positions, instead of flying around as seen in the first simulation.

The pressure for the three different simulations are listed as follows From W. Wagner et al. we know for Argon at $\sim 87\,\mathrm{K}$, the pressure is $\sim 100\,\mathrm{kPa}$, and with each increment of $\sim 10\,\mathrm{K}$, we expect the pressure to increase by one order. The details are presented as follows. This is very different from what we expected as

Table 5: Vapour pressure for Argon.

P[Pa]	1	10	100	1k	10k	100k
T[k]	-	47	53	61	71	87

Table 6: Additional simulation.

Simulation	State	ρ	$V[\mathrm{m}^3]$
Simulation x	Gas	0.176	4.687×10^{-23}

computed from (12), whose expected values are somewhat on the same order. This implies that, it is insufficient to infer an educated guess for pressure, in a simulation concerning only Lennard-Jones potential as two-point interaction potential. To further investigate pressure we have to take other facts into consideration.

In a nutshell, we have simulated a box of Argon particles, initialised with the lattice configuration. We have allowed the box to evolve for sufficiently long enough epochs such that the box is well equilibrated. For all three simulations we found that the symmetry given by the lattice configuration is to some extend broken. We have found that the three states are indeed distinctively correlated, with the solid state appearing a strong localised correlation, and the gas appearing a smoother behaviour in correlation, and liquid lies somewhere in between.

7 Further Investigation

Additionally, we have evolved the box using random initialisation. In this case, in order to circumvent positive potential, we compute the pair-wise distances and rescale the position, such that the smallest separation is now greater than 1σ. Below we show the result of this additional experiment. The Argon particles are initialised in gas state, with the following properties The snapshots of Simulation x are shown as follows.

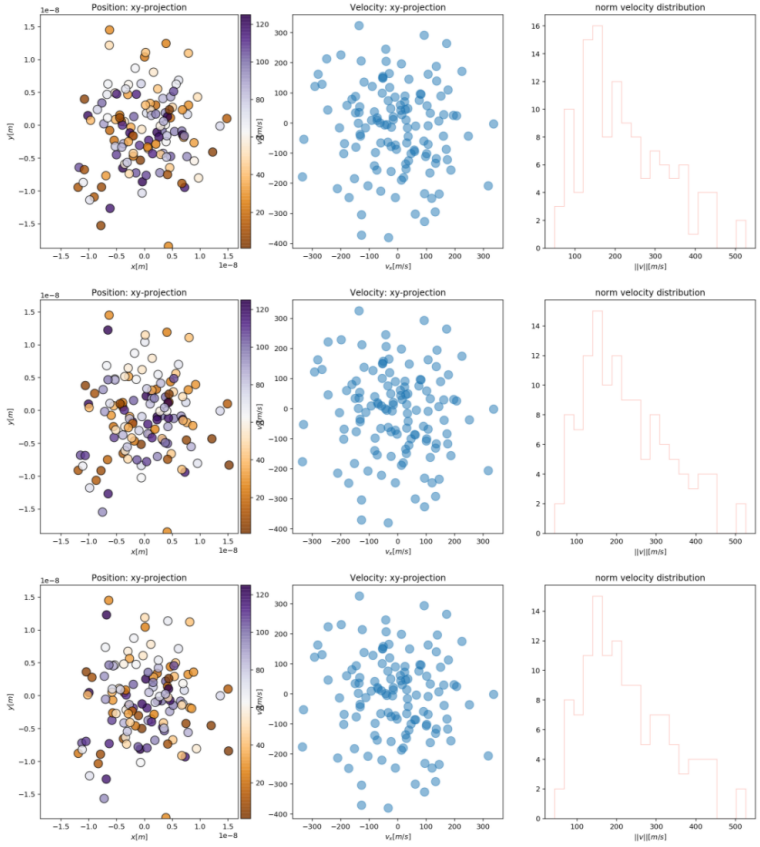
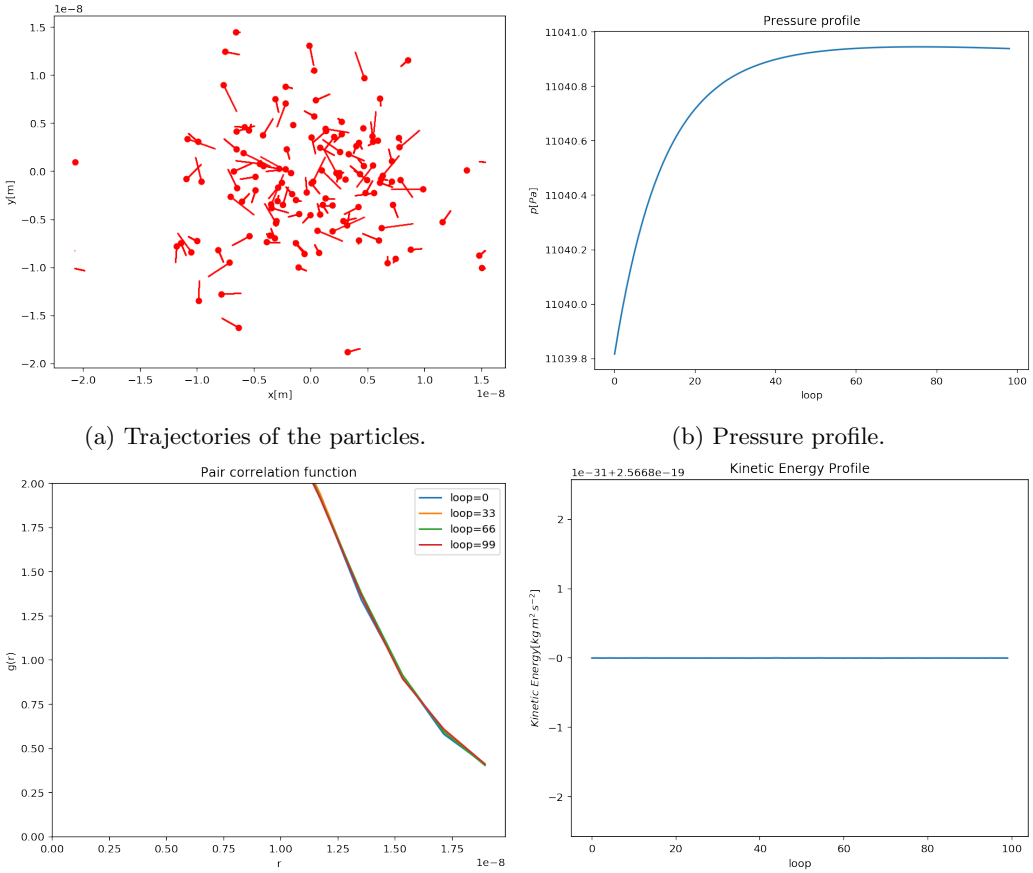


Figure 5: Snapshots for Simulation x, the positions of the particles are now initialised randomly. Details see 'random_generator' in 'Argon_simulation.py'.

The corresponding trajectories, correlation function, pressure profile, and kinetic energy are shown as follows.



(a) Trajectories of the particles.

(b) Pressure profile.

(c) Correlation function.

(d) Kinetic energy.

Figure 6: Outputs for Simulation x.

We have found the pressure to be 11040 Pa, with an error of 0.25 Pa. We have also found a smoother behaviour in the pressure profile, with even lower error. In addition to what as found above, this correlation function serves as a complement to what was shown and discussed above.

References

[1] Wagner, W. (1973), "New vapour pressure measurements for argon and nitrogen and a new method for establishing rational vapour pressure equations", Cryogenics, 13 (8): 470–482

8 Appendix

co

```
#!/usr/bin/env python
# coding: utf-8

from math import *
import numpy as np
import tqdm
import seaborn as sns
from scipy.optimize import newton, bisect
import matplotlib.animation as animation
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure, show
```

```
from mpl_toolkits.axes_grid1 import make_axes_locatable


class particle_in_box:
    global sigma
    global kB
    global T0
    global epsilon
    global m

    sigma = 3.405e-10  # Angstrom
    kB = 1.38e-23  # Boltzmann constant
    T0 = 100  # starting temperature of the box
    epsilon = T0*kB  # initialising the parameter epsilon, it changes w.r.t. the mean temperature of the box
    m = 6.6e-26  # mass of an Argon artom

    def __init__(self, Num=8, gentype=1, sep=1.6, T=1, loop=300, tol=.2, hlevel=10, *args, **kwargs):

        self.loop = 0
        self.sep = sep
        self.T = T0*T  # change temperature
        self.epsilon = self.T*kB
        self.loop_end = loop  # end loop
        self.tol = tol  # define an effective radius to calculate potential
        self.N = Num  # number of particles
        self.hlevel = hlevel  # parameter that defines the stepsize
        self.reff = self.r_eff()  # updating effective radius
        self.gentype = gentype

        if gentype == 1:
            try:
                self.Pos_stack = np.zeros((self.loop_end+1, self.N, 3))
                self.Vel_stack = np.zeros((self.loop_end+1, self.N, 3))
                self.Etarget = (self.N - 1)*1 * kB * self.T
                self.Pos, self.Vel, self.Lmin, self.Lmax = self.random_generator()
                self.V = (self.Lmax-self.Lmin)**3
                self.density = m*self.N/self.V
            except ValueError:
                raise ValueError("input Num should be an int^3!")

        elif gentype == 2:
            self.Pos_stack = np.zeros((self.loop_end+1, self.N, 3))
            self.Vel_stack = np.zeros((self.loop_end+1, self.N, 3))
            self.Etarget = (self.N - 1)*1 * kB * self.T
            self.Pos, self.Vel, self.Lmin, self.Lmax = self.cor_random_generator()
            self.V = (self.Lmax-self.Lmin)**3
            self.density = m*self.N/self.V

        elif gentype == 3:
            # the initialisation of Pos_stack, Vel_stack, E_target are done inside the function
            self.Pos, self.Vel, self.Lmin, self.Lmax = self.lattice_generator()
            self.N = len(self.Pos)
            self.V = (self.Lmax-self.Lmin)**3
            self.density = m*self.N/self.V

        else:
            raise NameError("gentype should be 1,2 or 3!")
        print('density is: ', '%.3e' % self.density, "kg m^-3.")
        print("temeperature is: ",np.round(self.T,2),"K.")
        print("simulation box size: ",'%.3e' % self.V,"m^3")

        self.arr = self.cube_extension()

    # default initialisation, this generator gives initial position on a meshgrid, then give it a small perturbation
    def random_generator(self):

        N = int(np.round(self.N**(1/3)))

        # place the particles on randomly
```

```
81          Pos = np.random.randn(N**3, 3)
82          # the velocities are generated from a 3D normal distribution
83          v0 = sqrt(epsilon/m)
84          mean = (0, 0, 0)
85          cov = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
86          Vel = np.random.multivariate_normal(mean, cov, N**3)
87          l = np.sqrt(self.Etarget/(m*np.sum(Vel**2)))
88          Vel = l*Vel
89          deltad = np.array([0])
90          xmatrix = Pos
91          # permute xmatrix
92          for i in range(1, N):
93              newx = np.concatenate((xmatrix[i:], xmatrix[0:i]), axis=0)
94              sig = xmatrix-newx
95              D = np.sqrt(np.sum(sig**2, axis=1))
96              deltad = np.concatenate((deltad, D), axis=None)
97          deltad = np.unique(deltad)
98          deltad = np.abs(deltad[deltad > 0])
99          self.dmin = np.min(deltad)
100
101         # rescale the position so that now the smallest separation is < 1.1 sigma
102         Pos *= self.sep/(deltad.min()/sigma)
103
104         # define the edge of the box
105         minarr = np.array([Pos[:, 0].min(), Pos[:, 1].min(), Pos[:, 2].min()])
106         maxarr = np.array([Pos[:, 0].max(), Pos[:, 1].max(), Pos[:, 2].max()])
107         Lmin = np.min(minarr)
108         Lmax = np.max(maxarr)
109
110         self.Pos_stack[0] = Pos
111         self.Vel_stack[0] = Vel
112
113         return Pos, Vel, Lmin, Lmax
114
115     # given N, generates N**3 cubes of lattice partices
116     def lattice_generator(self):
117
118         def lattice_generator(N):
119             n = 2
120             Pos = np.zeros((n**3, 3))
121             for i in range(len(Pos)):
122                 Pos[i][0] = i//n**2
123                 Pos[i][1] = (i-Pos[i][0]*n**2)//n
124                 Pos[i][2] = i-Pos[i][0]*n**2-Pos[i][1]*n
125
126             p0 = Pos[-1]/2
127             R = np.sqrt(np.sum(p0**2))
128             for item in Pos:
129
130                 p1 = np.array([item[0]+.5, item[1]+.5, item[2]])
131                 p2 = np.array([item[0]+.5, item[1], item[2]+.5])
132                 p3 = np.array([item[0], item[1]+.5, item[2]+.5])
133                 Pos = np.vstack((Pos, p1, p2, p3))
134
135             Pos = np.unique(Pos, axis=0)
136
137             Dist = np.sqrt((Pos[:, 0]-p0[0])**2 +
138                            (Pos[:, 1]-p0[0])**2+(Pos[:, 2]-p0[0])**2)
139             ind = Dist < 1.1*R
140             Pos = Pos[ind]
141
142             ext = np.zeros((N**3, 3))
143             Post = Pos
144             for i in range(len(ext)):
145                 ext[i][0] = i//N**2
146                 ext[i][1] = (i-ext[i][0]*N**2)//N
147                 ext[i][2] = i-ext[i][0]*N**2-ext[i][1]*N
148
149                 Post = np.vstack((Post, Pos+ext[i]))
150
151             Post = np.unique(Post, axis=0)
152             mask = (Post[:, 0] == Post.max()) + (Post[:, 1] ==
```

```
                                                    Post.max()) + (Post[:, 2] == Post.max())

            Post = Post[~mask]

            return Post

        Pos = lattice_generator(self.N)
        # rescale position so that the smallest separation is now user-defined sigma, and in the attractive range
        Pos *= self.sep*sigma/(sqrt(2)/2)
        # add a small translation
        Pos += 1*sigma
        N = len(Pos)
        self.Pos_stack = np.zeros((self.loop_end+1, N, 3))
        self.Vel_stack = np.zeros((self.loop_end+1, N, 3))
        self.Etarget = (N - 1)*1 * kB * self.T

        # generate velocities
        v0 = sqrt(epsilon/m)
        mean = (0, 0, 0)
        cov = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
        Vel = np.random.multivariate_normal(mean, cov, N)
        for i, a in enumerate(Vel):
            while sqrt(np.sum(a*a)) > 3:
                Vel[i] = np.random.multivariate_normal(mean, cov, 1)
        l = np.sqrt(self.Etarget/(m*np.sum(Vel**2)))
        Vel = l*Vel

        # define the edge of the box
        minarr = np.array([Pos[:, 0].min(), Pos[:, 1].min(), Pos[:, 2].min()])
        maxarr = np.array([Pos[:, 0].max(), Pos[:, 1].max(), Pos[:, 2].max()])
        sep = sqrt(2) * self.sep * sigma
        Lmin = np.min(minarr)
        Lmax = np.max(maxarr)+1*sep

        self.Pos_stack[0] = Pos
        self.Vel_stack[0] = Vel

        return Pos, Vel, Lmin, Lmax

    # correlated position and velocity, with higher velocities in the box centre

    def cor_random_generator(self):
        N = self.N
        mean = (0, 0, 0)
        cov = [[.1, 0, 0], [0, .1, 0], [0, 0, .1]]
        Pos = np.random.multivariate_normal(mean, cov, N)

        deltad = np.array([0])
        xmatrix = Pos
        # permute xmatrix
        for i in range(1, N):
            newx = np.concatenate((xmatrix[i:], xmatrix[0:i]), axis=0)
            sig = xmatrix-newx
            D = np.sqrt(np.sum(sig**2, axis=1))
            deltad = np.concatenate((deltad, D), axis=None)
        deltad = np.unique(deltad)
        deltad = np.abs(deltad[deltad > 0])
        self.dmin = np.min(deltad)
        # rescale the position so that now the smallest separation is < 1.1 sigma
        Pos *= self.sep/(deltad.min()/sigma)

        # the velocities are generated from a 3D normal distribution
        v0 = sqrt(epsilon/m)
        mean = (0, 0, 0)
        cov = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
        Vel = np.random.multivariate_normal(mean, cov, N)
        l = np.sqrt(self.Etarget/(m*np.sum(Vel**2)))
        Vel = l*Vel

        def Cartesian_to_Spherical(x, y, z):
            r = np.sqrt(x*x+y*y+z*z)
```

```
            theta = np.arctan2(y, x)
            phi = np.arccos(z/r)
            return r, theta, phi

        R, Theta, Phi = Cartesian_to_Spherical(
            Pos[:, 0]-.5*D, Pos[:, 1]-.5*D, Pos[:, 2]-.5*D)
        indsort = np.argsort(np.abs(R))  # sort R in ascending order
        Vel[:, 0][indsort] = np.sort(Vel[:, 0][::-1])
        Vel[:, 1][indsort] = np.sort(Vel[:, 1][::-1])
        Vel[:, 2][indsort] = np.sort(Vel[:, 2][::-1])

        # define the edge of the box
        minarr = np.array([Pos[:, 0].min(), Pos[:, 1].min(), Pos[:, 2].min()])
        maxarr = np.array([Pos[:, 0].max(), Pos[:, 1].max(), Pos[:, 2].max()])
        Lmin = np.min(minarr)  # *.999
        Lmax = np.max(maxarr)  # *1.001
        self.Pos_stack[0] = Pos
        self.Vel_stack[0] = Vel

        return Pos, Vel, Lmin, Lmax

    def r_eff(self):
        tol_percentage = self.tol
        def U(r): return 4*self.epsilon*((sigma/r)**12-(sigma/r)**6)
        Umin = U(2**(1/6)*sigma)
        def Usol(r): return U(r)-(1-tol_percentage)*Umin
        reff = bisect(Usol, (2**(1/6))*sigma, (2**(1/6)+3)*sigma)
        return reff

    def F(self, x, v):
        # if there is more than one particle in the force computing range
        if len(x.flatten())/3 > 1:
            r = np.sqrt(np.sum(x*x, axis=1))
            vtot = np.sum(v**2)
            eps = m*vtot/(len(x))
            U = 4*self.epsilon*((sigma/r)**12-(sigma/r)**6)
            dUdr = 24*self.epsilon*(-sigma/r**2) * \
                (sigma/r)**5*(2*(sigma/r)**6-1)
            div = -dUdr/r
            rdU = np.sum(dUdr*r)
            delvec = (np.zeros(x.shape)+div[:, np.newaxis])*x
            return np.sum(delvec, axis=0), eps, np.sum(U), rdU
        # if there is only one particle in the force computing range
        else:
            r = np.sqrt(np.sum(x*x))
            vtot = np.sum(v**2)
            eps = m*vtot
            U = 4*self.epsilon*((sigma/r)**12-(sigma/r)**6)
            dUdr = 24*self.epsilon*(-sigma/r**2) * \
                (sigma/r)**5*(2*(sigma/r)**6-1)
            div = -dUdr/r
            rdU = dUdr*r
            delvec = div*x
            return delvec, eps, U, rdU

    def cube_extension(self):
        arr = np.zeros((3**3, 3))
        D = self.Lmax - self.Lmin
        for i in range(27):
            arr[i][0] = i//3**2
            arr[i][1] = (i-arr[i][0]*3**2)//3
            arr[i][2] = i-arr[i][0]*3**2-arr[i][1]*3
        arr = D*(arr-np.ones(arr.shape))
        return arr

    def Verlet(self):
        self.pressure = np.zeros(self.loop_end)
        PE = np.zeros((self.loop_end, self.N))
        for loop_this in tqdm.tqdm(range(self.loop_end)):
            if loop_this >= self.loop_end:
                break
            h = sigma/(np.max(self.Vel))/self.hlevel
```

```
            Pos_ext = np.array([[0, 0, 0]])
            Vel_ext = np.array([[0, 0, 0]])
            D = self.Lmax - self.Lmin
            for i in range(len(self.arr)):
                newbox = self.Pos + self.arr[i][np.newaxis, :]
                Pos_ext = np.concatenate((Pos_ext, newbox), axis=0)
                Vel_ext = np.concatenate((Vel_ext, self.Vel), axis=0)

            Pos_ext = Pos_ext[1:]
            Vel_ext = Vel_ext[1:]
            '''
            fig = figure(figsize=(7.5,6))
            ax = fig.add_subplot(111)
            ax.plot(Pos_ext[:,0],Pos_ext[:,1],'o')
            show()
            '''
            correction = np.zeros(self.N)
            # for each particle, calcuate the neighbouring particles inside reff
            for i in range(self.N):

                P0 = self.Pos[i]
                V0 = self.Vel[i]
                Dist = np.sqrt(
                    np.sum((Pos_ext-P0*np.ones(Pos_ext.shape))**2, axis=1))
                mask = (Dist > 0) & (Dist < D/2)

                Peff = Pos_ext[mask]
                Veff = Vel_ext[mask]

                delva, self.epsilon, U0, sum1 = self.F(Peff, Veff)
                xth = P0+h*V0+.5*h**2/m*delva
                delvah, self.epsilon, Ui, sum2 = self.F(xth, V0)
                vth = V0+.5*h/m*(delvah+delva)

                if np.any(xth < self.Lmin):
                    ind = np.where(xth < self.Lmin)[0]
                    n = (xth[ind]-self.Lmin)//D
                    xth[ind] -= n*D

                if np.any(xth >= self.Lmax):
                    ind = np.where(xth > self.Lmax)
                    n = (xth[ind]-self.Lmin)//D
                    xth[ind] -= n*D

                correction[i] = sum2
                self.Pos[i] = xth
                self.Vel[i] = vth
                # rescale velocities
                l = np.sqrt(self.Etarget/(m*np.sum(self.Vel**2)))
                self.Vel = l*self.Vel
                self.Pos_stack[self.loop+1] = self.Pos
                self.Vel_stack[self.loop+1] = self.Vel
                PE[self.loop][i] = Ui
                # print("correction",correction)
            #print("pressure term",(1-.5*np.sum(correction)/(3*self.N*kB*T0)))
            self.pressure[self.loop] = (
                1-.5*np.sum(correction)/(3*self.N*kB*self.T))*kB*self.T*self.density/m
            self.epsilon = m*np.sum(self.Vel**2)/self.N
            Vel_r = np.sqrt(np.sum(self.Vel**2, axis=1))

            self.loop += 1

        return self.Pos_stack, self.Vel_stack, np.sum(PE, axis=1), self.pressure

    # get the kinetic energy of the particles per snapshot

    def get_KE(self):
        sum1 = np.sum(.5*m*self.Vel_stack[1:]**2, axis=2)
        # array of length loop_end+1 which gives the KE per snapshot
        sum2 = np.sum(sum1, axis=1)
        KE = sum2
        fig = figure(figsize=(7.5, 6))
```

```
368         ax = fig.add_subplot(111)
369         ax.plot(np.arange(len(KE)), KE)
370         ax.set_xlabel('loop')
371         ax.set_ylabel(r'$Kinetic \ Energy[kg\,m^2\,s^{-2}]$')
372         ax.set_title('Kinetic Energy Profile')
373         show()
374 
375     def get_plots(self):
376         for i in range(self.loop_end+1):
377             vx = self.Vel_stack[i][:, 0]
378             vy = self.Vel_stack[i][:, 1]
379             vz = self.Vel_stack[i][:, 2]
380             vr = np.sqrt(vx**2+vy**2+vz**2)
381             ind = np.argsort(vr)
382             t = (np.arange(len(vr))+np.ones(len(vr)))[ind]
383             if i % 20 == 0:
384                 fig = figure(figsize=(18, 6))
385                 ax1 = fig.add_subplot(131)
386                 ax2 = fig.add_subplot(132)
387                 ax3 = fig.add_subplot(133)
388                 ax1.plot(self.Pos_stack[i][:, 0], self.Pos_stack[i][:, 1], marker='o',
389                          markersize=3, ls=' ', color='white', alpha=.1, zorder=1)
390                 im1 = ax1.scatter(self.Pos_stack[i][:, 0], self.Pos_stack[i][:, 1],
391                                   marker='o', s=140, c=t, cmap='PuOr', edgecolor='k',
    alpha=.8, zorder=2)
392                 ax2.scatter(
393                     self.Vel_stack[i][:, 0], self.Vel_stack[i][:, 1], marker='o', s=140,
    alpha=.5)
394                 ax3.hist(np.sqrt(vx**2+vy**2+vz**2), bins=20,
395                          histtype='step', color='salmon', alpha=.4)
396                 divider = make_axes_locatable(ax1)
397                 cax1 = divider.append_axes('right', size='5%', pad=0.05)
398                 fig.colorbar(im1, cax=cax1, orientation='vertical')
399                 ax1.ticklabel_format(style='sci', axis='both')
400                 ax1.set_xlim([self.Lmin*.9, self.Lmax*1.1])
401                 ax1.set_ylim([self.Lmin*.9, self.Lmax*1.1])
402                 ax1.set_xlabel(r'$x[m]$')
403                 ax1.set_ylabel(r'$y[m]$')
404                 ax2.set_xlabel(r'$v_x[m/s]$')
405                 ax2.set_ylabel(r'$v_y[m/s]$')
406                 ax3.set_xlabel(r'$||v|||[m/s]$')
407                 ax1.set_title('Position: xy-projection')
408                 ax2.set_title('Velocity: xy-projection')
409                 ax3.set_title('norm velocity distribution')
410                 show()
411                 if i == 60:
412                     break
413 
414     def get_correlation(self):
415 
416         def pairCorrelation_3D(p, D, rMax, dr):
417             """Compute the three-dimensional pair correlation function for a set of
418             spherical particles contained in a cube with side length D.
419             Arguments:
420                 p               3d positions of the particles
421                 D               length of each side of the cube in space
422                 rMax            outer diameter of largest spherical shell
423                 dr              increment for increasing radius of spherical shell
424             Returns a tuple: (g, radii, interior_indices)
425                 g(r)            a numpy array containing the correlation function g(r)
426                 radii           a numpy array containing the radii of the
427                                 spherical shells used to compute g(r)
428             """
429 
430             # Find particles which are close enough to the cube center that a sphere of radius
431             # rMax will not cross any face of the cube
432 
433             x = p[:, 0]
434             y = p[:, 1]
435             z = p[:, 2]
436             mask1 = (x > rMax) + (x < (D - rMax))
437             mask2 = (y > rMax) + (y < (D - rMax))
```

```
            mask3 = (z > rMax) + (z < (D - rMax))
            interior_indices, = np.where(mask1 + mask2 + mask3)
            num_interior_particles = len(interior_indices)

            if num_interior_particles < 1:
                raise RuntimeError("No particles found for which a sphere of radius rMax\
                        will lie entirely within a cube of side length D.  Decrease rMax\
                        or increase the size of the cube.")

            edges = np.arange(0., rMax + 1.1 * dr, dr)

            num_increments = len(edges) - 1
            g = np.zeros([num_interior_particles, num_increments])
            radii = np.zeros(num_increments)
            numberDensity = len(x) / D**3

            # Compute pairwise correlation for each interior particle
            for p in range(num_interior_particles):
                index = interior_indices[p]
                d = np.sqrt((x[index] - x)**2 + (y[index] - y)
                            ** 2 + (z[index] - z)**2)
                d[index] = 2 * rMax

                (result, bins) = np.histogram(d, bins=edges, normed=False)
                g[p, :] = result / numberDensity

            # Average g(r) for all interior particles and compute radii
            g_average = np.zeros(num_increments)
            for i in range(num_increments):
                radii[i] = (edges[i] + edges[i+1]) / 2.
                rOuter = edges[i+1]
                rInner = edges[i]
                g_average[i] = np.mean(
                    g[:, i]) / (4.0 / 3.0 * pi * (rOuter**3 - rInner**3))

            return (g_average, radii)
        D = self.Lmax-self.Lmin
        Pos_stack = self.Pos_stack
        fig = figure(figsize=(7.5, 6))
        ax = fig.add_subplot(111)
        for i in range(len(Pos_stack)):
            #if (i > 0) & (i % (self.loop_end//3) == 0):
            if i % (self.loop_end//3) == 0:
                Pos = Pos_stack[i]
                g, r = pairCorrelation_3D(Pos, D, D/2., D/20.)
                ax.plot(r, g, label='loop='+str(i))
                ax.legend()
        ax.set_ylim([0, 2])
        ax.set_xlabel('r')
        ax.set_ylabel('g(r)')
        ax.set_title('Pair correlation function')
        show()
        return fig, ax

    def get_pressure(self):
        pressure = self.pressure[1:]
        N = len(pressure)
        fig = figure(figsize=(7.5, 6))
        ax = fig.add_subplot(111)
        ax.plot(np.arange(N), pressure)
        ax.set_xlabel('loop')
        ax.set_ylabel(r'$p[Pa]$')
        ax.set_title('Pressure profile')
        show()
        print("Mean: ",np.mean(pressure),"std: ",np.std(pressure))

    def get_trajectories(self):
        Pos_stack = self.Pos_stack
        fig = figure(figsize=(7.5, 6))
        ax = fig.add_subplot(111)
        for i in range(len(Pos_stack)):
            if i == len(Pos_stack)-1:
```

```
                ax.plot(Pos_stack[i][:, 0], Pos_stack[i]
                        [:, 1], '.', markersize=10, color='r')
            ax.plot(Pos_stack[i][:, 0], Pos_stack[i]
                    [:, 1], '.', markersize=.5, color='r')
            ax.set_xlabel('x[m]')
            ax.set_ylabel('y[m]')
        show()


if __name__ == "__main__":

    # initialise with lattice generator
    """
    below gives the configuration for the liquid state, with rho = 1.39e3 kg /^3, T = 100 K,
    using the default generation model
    """

    loop = 100
    hlevel = 50
    tol = 0.1

    # gas
    print("Gas state: ")
    Argon_particle = particle_in_box(
        Num=3, sep=1.022/(3/8.)**(1/3.), T=3, loop=loop, tol=tol, gentype=3, hlevel=hlevel)
    Argon_particle.Verlet()
    # Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
    ",round(np.std(self.pressure),2))
    # Argon_particle.get_correlation()
    # Argon_particle.get_pressure()
    # Argon_particle.get_KE()


    print("Liquid state: ")
    Argon_particle = particle_in_box(
        Num=3, sep=1.022, T=1, loop=loop, tol=tol, gentype=3, hlevel=hlevel)

    Argon_particle.Verlet()
    # Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
    ",round(np.std(self.pressure),2))
    # Argon_particle.get_correlation()
    # Argon_particle.get_pressure()
    # Argon_particle.get_KE()

    # solid
    print("Solid state: ")
    Argon_particle = particle_in_box(
        Num=3, sep=1.022/1.5**(1/3.), T=.5, loop=loop, tol=tol, gentype=3, hlevel=hlevel)
    Argon_particle.Verlet()
    # Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
    ",round(np.std(self.pressure),2))
    # Argon_particle.get_correlation()
    # Argon_particle.get_pressure()
    # Argon_particle.get_KE()


    """
    below gives the configuration for the liquid state, with rho = 1.39e3 kg /^3, T = 100 K,
    using the default generation model
    """
    '''
    #initialise with random generator
    print("Liquid state: ")
    Argon_particle = particle_in_box(Num=216,sep = 1.0001,T =
    1,loop=100,tol=.1,gentype=1,hlevel=100)
    Argon_particle.Verlet()
```

```
    Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    Argon_particle.get_correlation()
    Argon_particle.get_pressure()
    Argon_particle.get_KE()


    print("Gas state: ")
    Argon_particle = particle_in_box(Num=216,sep = 1.05/(3/8.)**(1/3.),T =
    3,loop=100,tol=.1,gentype=1,hlevel=100)
    Argon_particle.Verlet()
    Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    Argon_particle.get_correlation()
    Argon_particle.get_pressure()
    Argon_particle.get_KE()

    print("Solid state: ")
    Argon_particle = particle_in_box(Num=216,sep = 1.05/1.5**(1/3.),T =
    .5,loop=100,tol=.1,gentype=1,hlevel=100)
    Argon_particle.Verlet()
    Argon_particle.get_plots()
    Argon_particle.get_trajectories()
    Argon_particle.get_correlation()
    Argon_particle.get_pressure()
    Argon_particle.get_KE()
    '''
```

Argon_Simulation.py

```
#!/usr/bin/env python
# coding: utf-8


from Argon_simulation import particle_in_box
import numpy as np


fac = 1

np.random.seed(8888)
# gas
print("Gas state: ")
Argon_particle = particle_in_box(
    Num=4, sep=1.022/(3/8.)**(1/3.), T=fac*3, loop=100, tol=0.1, gentype=3, hlevel=40)
Argon_particle.Verlet()
print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
    ",round(np.std(self.pressure),2))
Argon_particle.get_plots()
Argon_particle.get_trajectories()
Argon_particle.get_correlation()
Argon_particle.get_pressure()
Argon_particle.get_KE()



np.random.seed(8888)
print("Liquid state: ")
Argon_particle = particle_in_box(
    Num=4, sep=1.022, T=fac*1, loop=100, tol=0.1, gentype=3, hlevel=100)

Argon_particle.Verlet()
print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
    ",round(np.std(self.pressure),2))
# Argon_particle.get_plots()
Argon_particle.get_trajectories()
Argon_particle.get_correlation()
Argon_particle.get_pressure()
Argon_particle.get_KE()


```

```
41
42 np.random.seed(8888)
43 # solid
44 print("Solid state: ")
45 Argon_particle = particle_in_box(
46     Num=4, sep=1.022/1.5**(1/3.), T=fac*.5, loop=100, tol=0.1, gentype=3, hlevel=300)
47 Argon_particle.Verlet()
48 print("pressure is: ",round(np.mean(self.pressure),2),"Pa, with error:
     ",round(np.std(self.pressure),2))
49 # Argon_particle.get_plots()
50 Argon_particle.get_trajectories()
51 Argon_particle.get_correlation()
52 Argon_particle.get_pressure()
53 Argon_particle.get_KE()
54
55
56
57
58 # Now with random initialsation
59 print("random, Gas state: ")
60 Argon_particle = particle_in_box(
61     Num=125, sep=1.022/(3/8.)**(1/3.), T=fac*3, loop=100, tol=0.1, gentype=1, hlevel=10)
62 Argon_particle.Verlet()
63 Argon_particle.get_plots()
64 Argon_particle.get_trajectories()
65 Argon_particle.get_correlation()
66 Argon_particle.get_pressure()
67 Argon_particle.get_KE()
```

sim_run.py

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  from math import *
5  import numpy as np
6  import matplotlib.pyplot as plt
7  from mpl_toolkits.mplot3d import Axes3D
8  from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection
9
10 def lattice_generator(N):
11     n = 2
12     Pos = np.zeros((n**3,3))
13     for i in range(len(Pos)):
14         Pos[i][0] = i//n**2; Pos[i][1] = (i-Pos[i][0]*n**2)//n;Pos[i][2] =
       i-Pos[i][0]*n**2-Pos[i][1]*n
15
16
17     p0 = Pos[-1]/2
18     R = np.sqrt(np.sum(p0**2))
19     for item in Pos:
20
21         p1 = np.array([item[0]+.5,item[1]+.5,item[2]])
22         p2 = np.array([item[0]+.5,item[1],item[2]+.5])
23         p3 = np.array([item[0],item[1]+.5,item[2]+.5])
24         Pos = np.vstack((Pos,p1,p2,p3))
25
26     Pos = np.unique(Pos,axis=0)
27
28     Dist = np.sqrt((Pos[:,0]-p0[0])**2+(Pos[:,1]-p0[0])**2+(Pos[:,2]-p0[0])**2)
29     ind = Dist<1.1*R
30     Pos = Pos[ind]
31
32
33     ext = np.zeros((N**3,3))
34     Post = Pos
35     for i in range(len(ext)):
36         ext[i][0] = i//N**2; ext[i][1] = (i-ext[i][0]*N**2)//N;ext[i][2] =
       i-ext[i][0]*N**2-ext[i][1]*N
37
38         Post = np.vstack((Post,Pos+ext[i]))
39
```

```
    Post = np.unique(Post, axis=0)

    return Post




def plot_cube(cube_definition):
    cube_definition_array = [
        np.array(list(item))
        for item in cube_definition
    ]

    points = []
    points += cube_definition_array
    vectors = [
        cube_definition_array[1] - cube_definition_array[0],
        cube_definition_array[2] - cube_definition_array[0],
        cube_definition_array[3] - cube_definition_array[0]
    ]

    points += [cube_definition_array[0] + vectors[0] + vectors[1]]
    points += [cube_definition_array[0] + vectors[0] + vectors[2]]
    points += [cube_definition_array[0] + vectors[1] + vectors[2]]
    points += [cube_definition_array[0] + vectors[0] + vectors[1] + vectors[2]]

    points = np.array(points)

    edges = [
        [points[0], points[3], points[5], points[1]],
        [points[1], points[5], points[7], points[4]],
        [points[4], points[2], points[6], points[7]],
        [points[2], points[6], points[3], points[0]],
        [points[0], points[2], points[4], points[1]],
        [points[3], points[6], points[7], points[5]]
    ]



    faces = Poly3DCollection(edges, linewidths=1, edgecolors='k')
    faces.set_facecolor((0,0,1,0.1))

    ax.add_collection3d(faces)

    # Plot the points themselves to force the scaling of the axes
    ax.scatter(points[:,0], points[:,1], points[:,2], s=0)
    ax.axis('off')




fig = plt.figure(figsize=(10,10))
n=2
Post = lattice_generator(n)
m = np.arange(len(Post))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Post[:,0],Post[:,1],Post[:,2],c=m,s=140,alpha=1,cmap='tab20',edgecolor='k')


for i in range(n):
    for j in range(n):
        for k in range(n):
            cube_definition = [(i,j,k),(i,j+1,k), (i+1,j,k), (i,j,k+1)]
            plot_cube(cube_definition)

plt.show()
```

plot_lattice.py