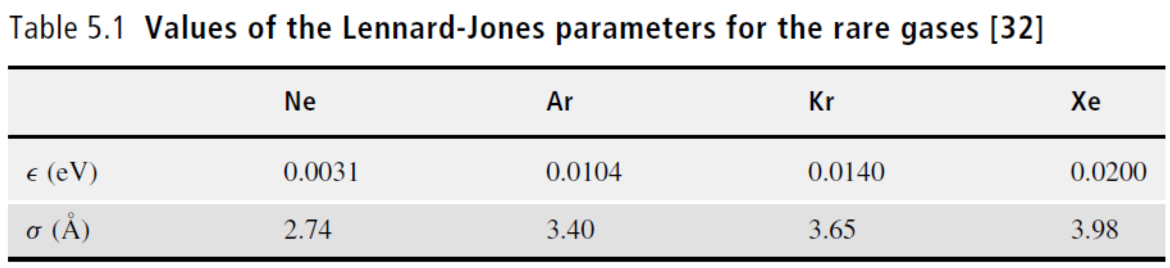
Table 5.1 **Values of the Lennard-Jones parameters for the rare gases [32]**

	Ne	Ar	Kr	Xe
ϵ (eV)	0.0031	0.0104	0.0140	0.0200
σ (Å)	2.74	3.40	3.65	3.98

Question 4) Finding the optimal timestep for a rare gas atom

For this question, Argon was chosen. Values for ε and σ were sourced from Table 5.1 in Lesar and so were taken as 0.0104 eV and 3.40 Å, respectively. These were then converted to SI units using the appropriate conversion factors. The mass of an Argon atom was calculated by dividing its molecular mass (0.039948 kg/mol) by Avagadro's number. The reduced timestep was calculated using the formula $dtr = \frac{dt}{t0}$, which was calculated to be 0.00233.

Next, the optimal timestep was determined by setting up a for loop to use the MDLJ function provided. It was found that there was significant fluctuation in the calculated optimal timestep if fewer than 7 unit cells were used, so 7 was chosen. The same density and input temperature that was used in the MDLJ script was also used here, at 0.9 and 0.2, respectively. 300 timesteps were chosen to get an accurate estimate whilst maintaining a reasonable computation time. The validity of each reduced temperature tested was determined by using equation 12 in the briefing document, which shall be termed the fluctuation, with the objective being to maximise the size of the timestep whilst keeping the fluctuation under 10^{-4}.

After much trial and error, it was found that testing a range of reduced timesteps between the current value of 0.00233 and 0.1 gave the best results. The final reduced timestep was calculated to be 0.01, plus or minus small amounts given that there were always fluctuations. This was used for all further exercises.

Question 5) Modelling a solid system

A density of solid Argon of 1.616 g/cm^3 was sourced from literature[1], and the reduced density was calculated in addition to the other reduced parameters, which was found to be 0.9575. The MDLJ function was used with 3000 timesteps, with an input temperature of -195 °C, as this is below the boiling point of Argon at -185.8 °C and so should ensure the material is a solid. The MDLJ function was modified to also output the number of atoms in the system, with which certain other output parameters were divided for the graphs. The resultant graphs are illustrated in Figure 1.

As can be seen, equilibrium is reached. A cutoff of 250 timesteps was chosen, wherein the first 250 timesteps were removed, to the calculate the averages of the equilibrated system. 250 was chosen as it is clear from visual inspection that all parameters are at equilibrium by the 250th timestep. The graphs with the first 250 timesteps removed are shown in Figure 2. The average values are given below in Table 1.

Table 1. Average reduced parameters for solid Argon.

U*	-7.638
K*	0.505
E*	-7.133
T*	0.336
P*	-2.793

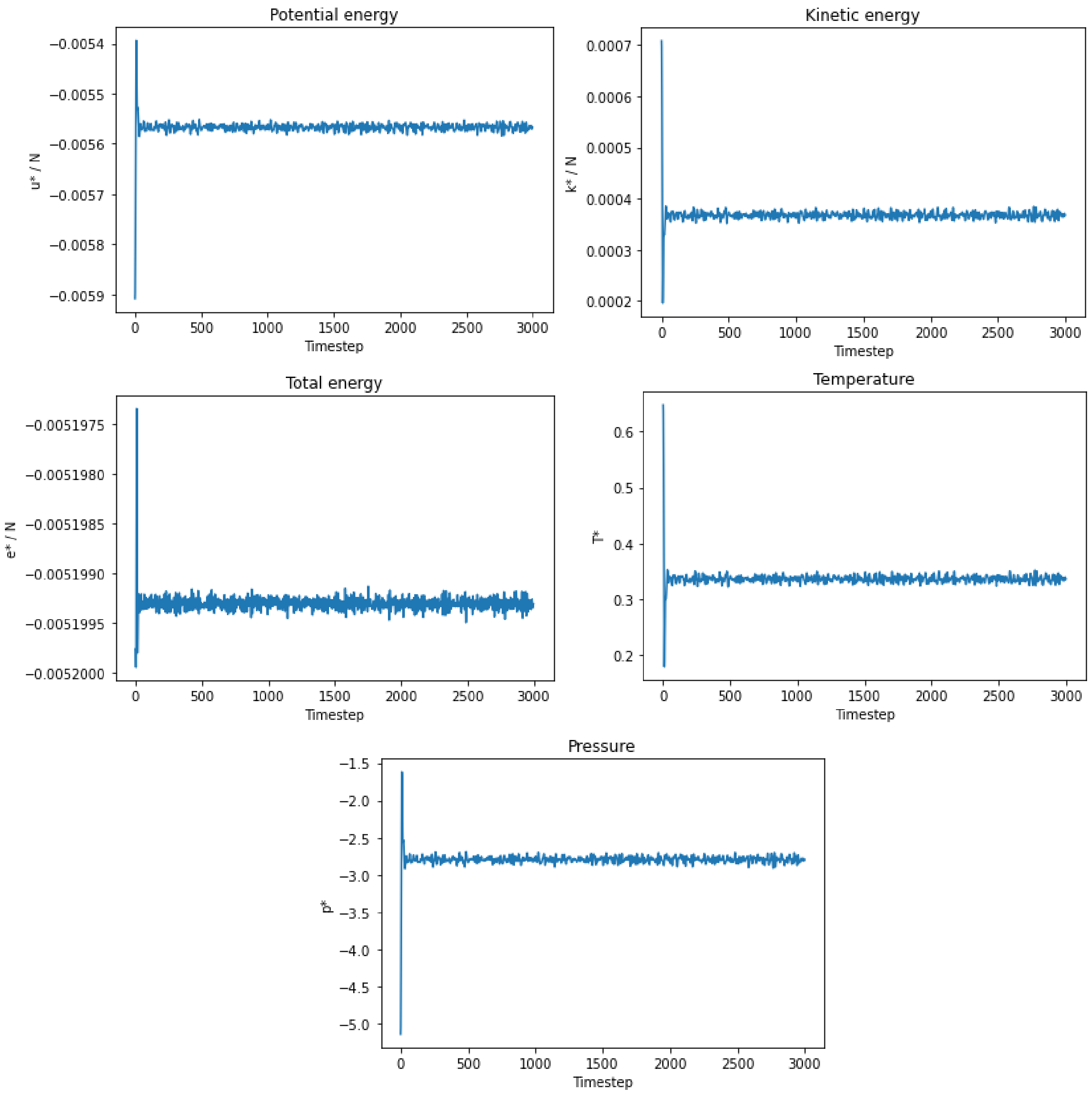

Figure 1. Output parameters for solid Argon.

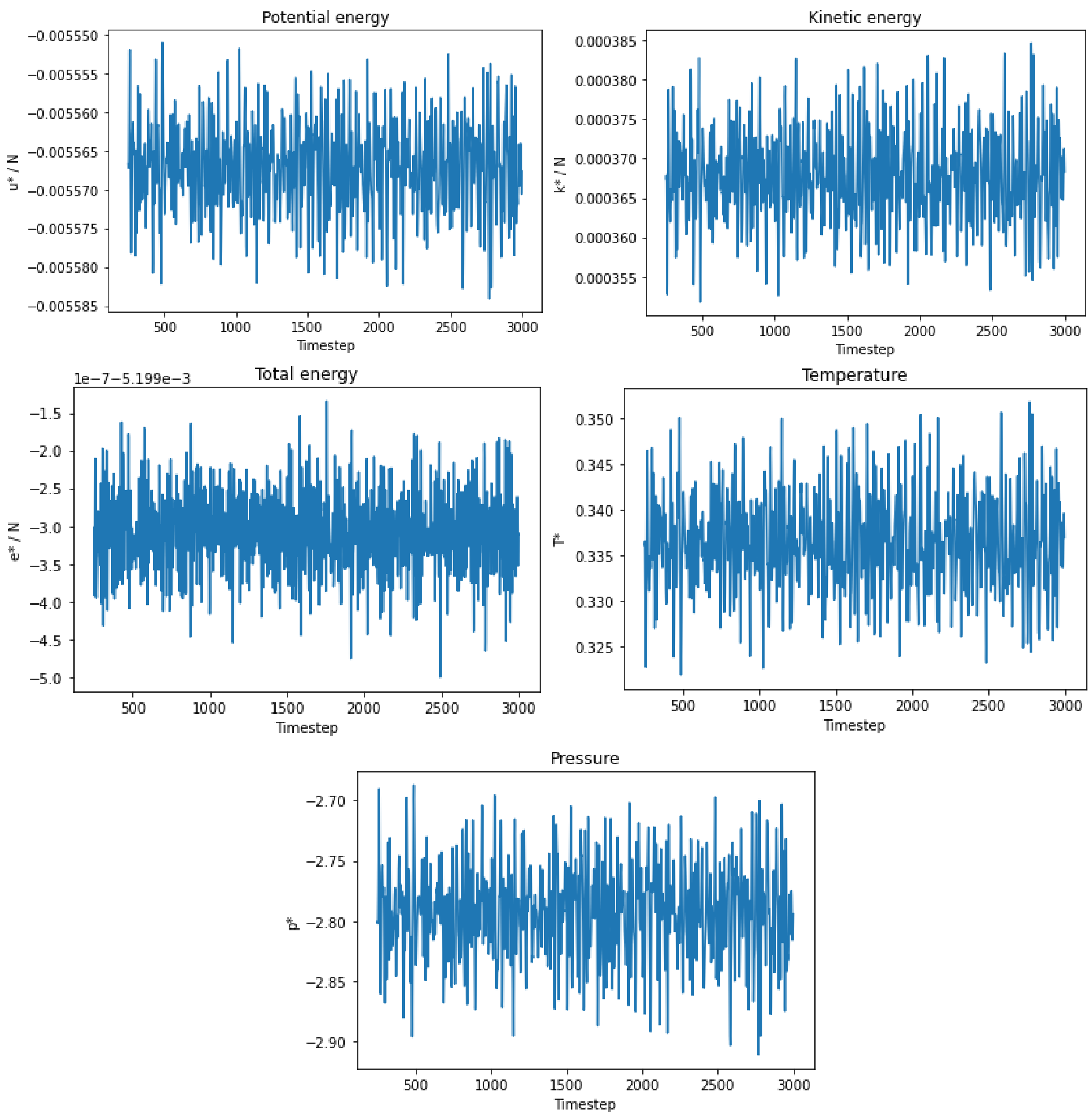

Figure 2. Output parameters at equilibrium for solid Argon.

Question 7) The same for the liquid

From literature, a value for the density of solid Argon of 1.414 g/cm^3 was found[1]. However, in a later question when calculating the pair distribution function, it was found that this gave a solid structure even at a temperature above the melting point of Argon. Hence, a density of 1.3 g/cm^3 was used, as this gave better results for a liquid. The same procedure as in Question 5 was used but with the different density and a temperature of -187 °C. The results are displayed in Figures 3 and 4. As can be seen, the liquid takes longer to reach equilibrium than the solid, as expected. For this reason, the number of timesteps was increased to 5000 and the cutoff to 1500. The liquid parameters also seem to drift away from the average and appear less uniform that the solid. The effect of increasing the number of timesteps was also that the fluctuations of the parameters was reduced and there was a higher pressure (less negative) in the system. The reduced parameters for the liquid are given in Table 2.

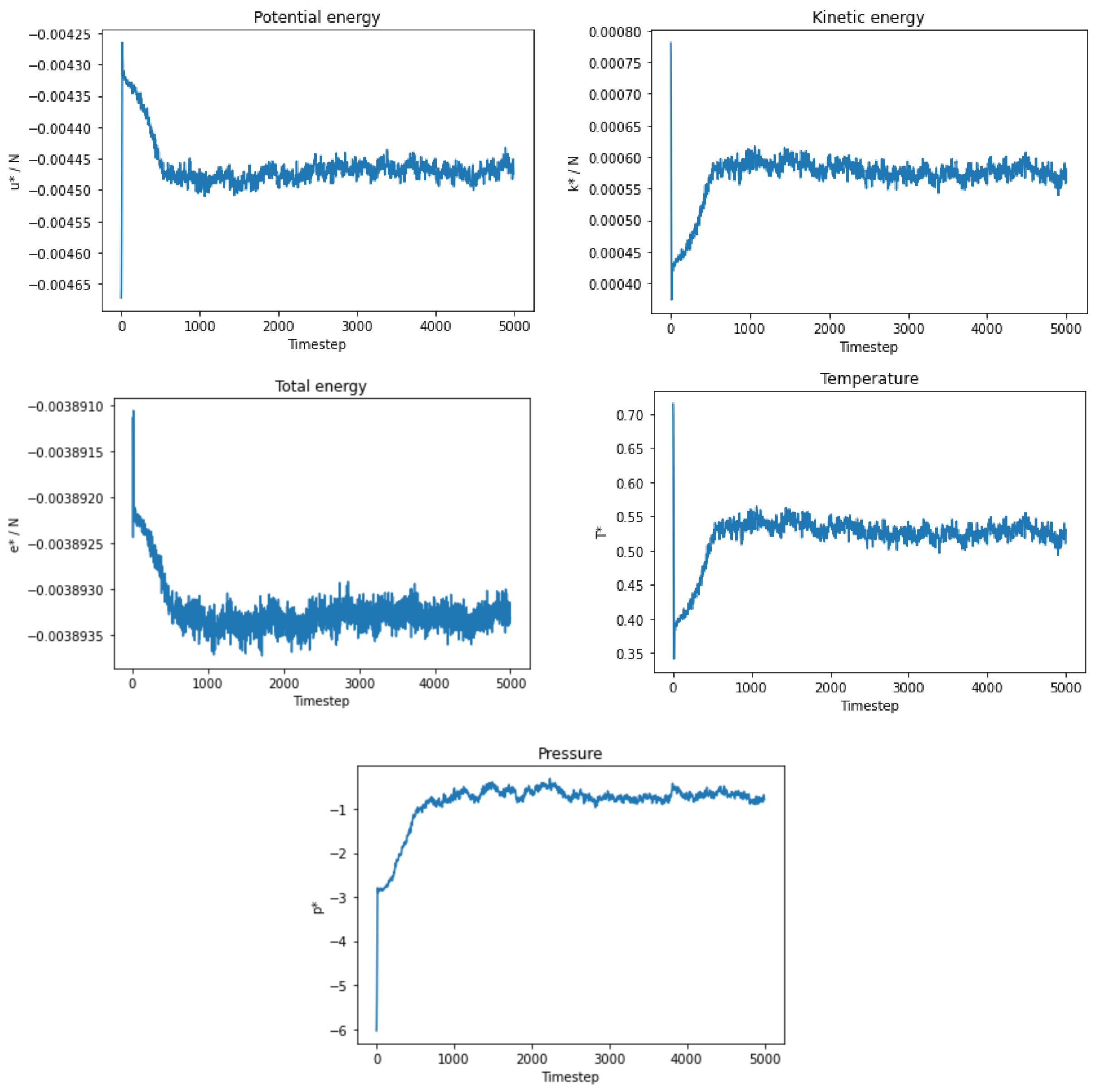

Figure 3. Output parameters for liquid Argon.

Table 2. Average reduced parameters for liquid Argon.

U*	-6.129
K*	0.788
E*	-5.342
T*	0.525
P*	-0.803

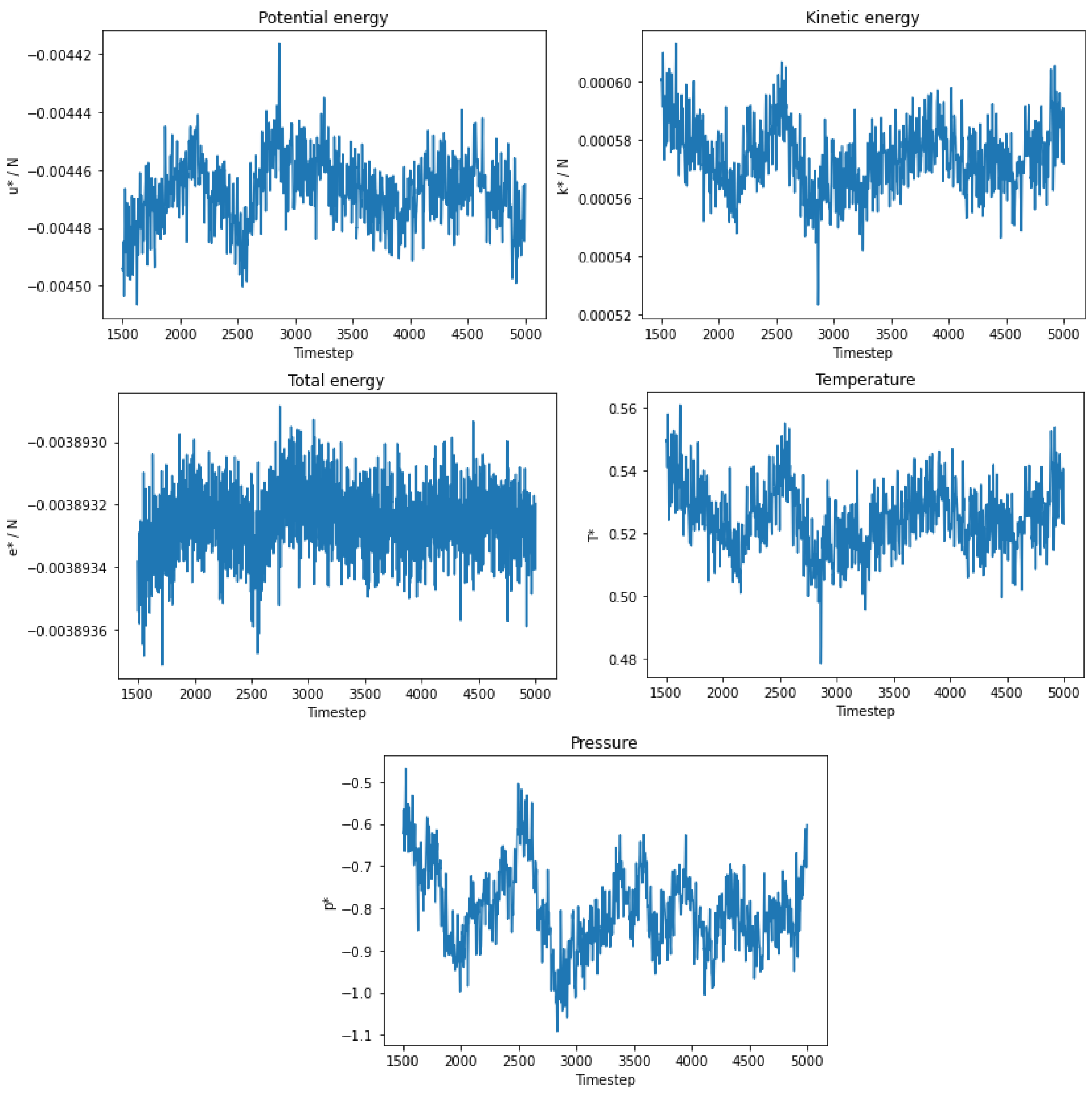

Figure 4. Output parameters at equilibrium for liquid Argon.

Question 8) Calculate fluctuation quantities

To calculate the fluctuation, the variances of the mean of the parameters were calculated. This was done by dividing the results for the solid and liquid into bins, then averaging the data over those bins. This created a list of averages over which the variances could be determined. The variances for the solid and liquid Argon are compared in Table 3. As can be seen, there was considerably more variance with the liquid Argon than with the solid.

Table 3. Variances of parameters for solid and liquid Argon.

Variances	Solid	Liquid
$\sigma^2(U^*)$	7.11×10^{-8}	1.78×10^{-3}
$\sigma^2(K^*)$	7.21×10^{-8}	1.76×10^{-3}
$\sigma^2(E^*)$	2.44×10^{-11}	7.77×10^{-8}
$\sigma^2(T^*)$	3.20×10^{-8}	7.81×10^{-4}
$\sigma^2(P^*)$	4.45×10^{-6}	1.62×10^{-1}

Question 9) Calculate radial distribution functions

As mentioned previously, the liquid density was found to be too high to give a radial distribution function characteristic of a liquid, as was found out when doing this question. The code from the briefing document was modified to work for Python and the MDLJ function was modified to also output the final x, y and z coordinates of every atom. An experimental lattice parameter for Argon of 5.486 Å [2] was chosen to input to the g(r) function and the resultant graphs for the solid and the liquid are illustrated in Figure 5. As can be seen, there are clear peaks in the solid graph, indicating neighbour shells. With the liquid graph, the peaks are less well defined as the is not a rigorous, periodic structure like with the solid.

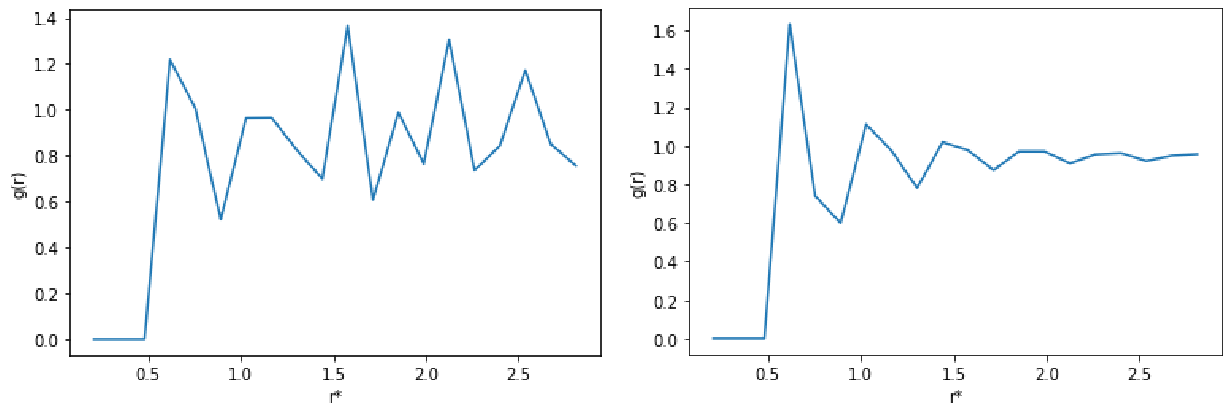

Figure 5. Radial distribution for the solid (left) and liquid (right) Argon.

Question 11) Calculate the velocity autocorrelation function

This was achieved by again modifying the MDLJ function provided. This time, to copy the initial velocities of each atom at each timestep such that the product of the initial and final velocities for all atoms can be stored and output by the function. This output was then averaged over all the atoms at each timestep and multiplied by m/3k_BT to give the c_{vv} for each timestep, which was then plotted. The graphs for the solid and the liquid are compared in Figure 6. The shapes of the graphs are somewhat as expected, as they have the initial high peak followed by the sharp trough, or well, then a plateau. However, the plateau is not around 0, for reasons unknown.

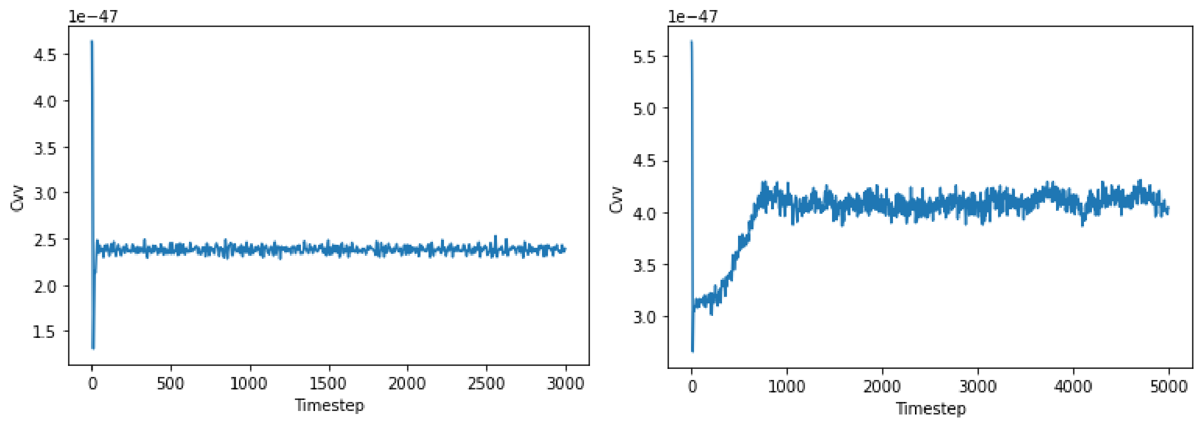

Figure 6. Velocity autocorrelation function for the solid (left) and liquid (right) Argon.

References:

[1] Witzenburg, W.V. 1967. Density of solid argon at the triple point and concentration of vacancies. *Physics Letters A.* **25**(4), pp.293-294.

[2] Barrett, C.S. and Meyer, L. 1965. The Crystal Structures of Argon and Its Alloys. In: Daunt J.G., Edwards D.O., Milford F.J., Yaqub M. (eds) *Low Temperature Physics LT9*. Springer: Boston, pp.1085-1088.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 29 10:55:04 2021

@author: tom_m
"""

from __future__ import division  #make / return a float instead of a
truncated int (for python 2)
#from math import sqrt
#import numpy as np
from numpy import zeros, pi, cos, log, sqrt, round, copy
from random import random #seed
from pylab import figure, plot, show, title
from numba import jit

@jit
def fLJsum(a, n, rc, x, y, z):
    """
    simple lattice sum for force with cutoffs and
    minimum image convention

     we calculate force (fx,fy,fz), energy (u), and
        part of the pressure (w)
    """
    #  set force components, potential energy, and pressure to 0
    fx = zeros(n, float)
    fy = zeros(n, float)
    fz = zeros(n, float)
    u = 0
    w = 0
    for i in range(n-1):  # note limits
        ftx = 0
        fty = 0
        ftz = 0
        for j in range(i+1,n):  #  note limits
    # mimimum image convention
            dx = x[j] - x[i]
            dy = y[j] - y[i]
            dz = z[j] - z[i]
            dx = dx - round(dx)
            dy = dy - round(dy)
            dz = dz - round(dz)
            dist = a*sqrt(dx**2 + dy**2 + dz**2)
            if(dist < 1e-5):
                print("dist zero:",dist,i,j)
            if dist <= rc:
#                print( dist)
                dphi = (2/dist**(12)-1/dist**6)
                ffx = dphi*a*dx/dist**2
                ffy = dphi*a*dy/dist**2
                ffz = dphi*a*dz/dist**2
                ftx = ftx + ffx
                fty = fty + ffy
                ftz = ftz + ffz
                phi = (1/dist**(12)-1/dist**6)
                u = u + phi
                w = w + dphi
    #  add -f to sum of force on j
                fx[j] = fx[j] - ffx
                fy[j] = fy[j] - ffy
                fz[j] = fz[j] - ffz
      #  sum up force on i (fi)
        fx[i] = fx[i] + ftx
        fy[i] = fy[i] + fty
        fz[i] = fz[i] + ftz
    #  need to multiply LJ by 4 and force and pressure by 24
    #  also need to correct sign in f
    u = 4*u
    w = 24*w
    fx *= -24
    fy *= -24
    fz *= -24
    # ~ for i in range(n):
        # ~ fx[i] = -24*fx[i]
        # ~ fy[i] = -24*fy[i]
        # ~ fz[i] = -24*fz[i]
    return u, w, fx, fy, fz

def initLJMD(nc, tin):
    """
    #create initial positions and velocities
    #for atomic MD calculation
    #  atoms in fcc structure to start
    #  velocities scaled for a given reduced temperature
    #input:
    #  nc = number of fcc cells
    #     n = number of atoms = 4*nc**3
    #     nc=2, n=32
    #     nc=3, n=108
    #  tin = initial temperature
    #
    #output:
    #  n = number of atoms
    #  [sx,sy,sz] = scaled coordinates
    #  [vx,vy,vz] = scaled velocities
    """
    #  created scaled coordinates in an fcc lattice
    ncell = 4
    x = [0, .5, 0, .5]
    y = [0, .5, .5, 0]
    z = [0, 0, .5, .5]
    i1 = 0
    n = ncell*nc**3
    sx = zeros(n, float)
    sy = zeros(n, float)
    sz = zeros(n, float)
    vx = zeros(n, float)
    vy = zeros(n, float)
    vz = zeros(n, float)
    for k in range(nc):
        for l in range(nc):
            for m in range(nc):
                for i in range(ncell):
                    sx[i1] = (x[i] + k-1)/nc
                    sy[i1] = (y[i] + l-1)/nc
                sz[i1] = (z[i] + m-1)/nc
                i1 = i1+1
    #  pick velocities from Maxwell-Boltzmann distribution
    #  for any temperature we want.
    #  Then we will calculate the kinetic energy and thus
    #  the temperature of these atoms and then we will
    #  rescale the velocities to the correct temperature
    k = 0
    px = 0
    py = 0
    pz = 0
    for i in range(n):
        vx[i] = sqrt(-2*log(random()))*cos(2*pi*random())
        vy[i] = sqrt(-2*log(random()))*cos(2*pi*random())
        vz[i] = sqrt(-2*log(random()))*cos(2*pi*random())
        px = px + vx[i]
        py = py + vy[i]
        pz = pz + vz[i]
    # set net momentum to zero and calcuate K
    px = px/n
    py = py/n
    pz = pz/n
    vx -= px
    vy -= py
    vz -= pz
    vx2 = vx**2
    vy2 = vy**2
    vz2 = vz**2
    k = 0.5*(vx2.sum()+vy2.sum()+vz2.sum())
    # ~ for i in range(n):
        # ~ vx[i] = vx[i]-px
        # ~ vy[i] = vy[i]-py
        # ~ vz[i] = vz[i]-pz
        # ~ k = k + vx[i]**2 + vy[i]**2 + vz[i]**2
    # ~ k = .5*k
    # kinetic energy of desired temperature (tin)
    kin = 3*n*tin/2
    # rescale velocities
    sc = sqrt(kin/k)
    vx *= sc
    vy *= sc
    vz *= sc
    # ~ for i in range(n):
        # ~ vx[i] = vx[i]*sc
        # ~ vy[i] = vy[i]*sc
        # ~ vz[i] = vz[i]*sc
    return n, sx, sy, sz, vx, vy, vz


def MDLJ(nc,density,tin,nsteps,dt):
    """
    # MD code for 3D system of LJ atoms
    #  input:  nc = number of cells of fcc unit cell
    #          density = density of LJ system
    #          tin = initial temperature
    #          nsteps = number of time steps
    #          dt = time step
    #  output:  un = potential energy at each time step
    #           kn = kinetic energy at each time step
    #           en = total energy at each time step
    #           tn = temperature at each time step
    #           pn = pressure at each time step
    """
    # initialize positions and velocities
    n, x, y, z, vx, vy, vz = initLJMD(nc, tin)
#    print("n, x, y, z, vx, vy, vz :]n",n, x, y, z, vx, vy, vz )
    # calculate some useful quantities
    vol = n/density
    a = vol**(1/3)
    rc = a/2

    # calculate initial energy and forces
    u, w, fx, fy, fz = fLJsum(a, n, rc, x, y, z)
#    sys.exit(-1)

#    print(u, w, fx, fy, fz )
    # now start the time stepping with the verlet algorithm
    # initialize variables
    xold = zeros(n,float)
    yold = zeros(n,float)
    zold = zeros(n,float)
    xnew = zeros(n,float)
    ynew = zeros(n,float)
    znew = zeros(n,float)

    un = zeros(nsteps,float)
    kn = zeros(nsteps,float)
    en = zeros(nsteps,float)
    tn = zeros(nsteps,float)
    pn = zeros(nsteps,float)

    cvv = zeros((nsteps,n),float)

    # first find the positions at t-dt
    xold = x -vx*dt/a+0.5*fx*dt**2/a
    yold = y -vy*dt/a+0.5*fy*dt**2/a
    zold = z -vz*dt/a+0.5*fz*dt**2/a
#    for i in range(n):
#        xold[i] = x[i] - vx[i]*dt/a + .5*fx[i]*dt**2/a
#        yold[i] = y[i] - vy[i]*dt/a + .5*fy[i]*dt**2/a
#        zold[i] = z[i] - vz[i]*dt/a + .5*fz[i]*dt**2/a

    # start the time steps
    for j in range(nsteps):
        k = 0
     #  find positions for time t + dt
     #  find velocities for time t
     #  find kinetic energy for time t

        vxo = copy(vx)
        vyo = copy(vy)
        vzo = copy(vz)

        xnew = 2*x - xold + fx*dt**2/a
        ynew = 2*y - yold + fy*dt**2/a
        znew = 2*z - zold + fz*dt**2/a
        vx = a*(xnew - xold)/(2*dt)
        vy = a*(ynew - yold)/(2*dt)
        vz = a*(znew - zold)/(2*dt)
        vx2 = vx**2
        vy2 = vy**2
        vz2 = vz**2
        k = 0.5*(vx2.sum()+vy2.sum()+vz2.sum())
#        for i in range(n):
#            xnew[i] = 2*x[i] - xold[i] + fx[i]*dt**2/a
#            ynew[i] = 2*y[i] - yold[i] + fy[i]*dt**2/a
#            znew[i] = 2*z[i] - zold[i] + fz[i]*dt**2/a
#            vx[i] = a*(xnew[i] - xold[i])/(2*dt)
#            vy[i] = a*(ynew[i] - yold[i])/(2*dt)
#            vz[i] = a*(znew[i] - zold[i])/(2*dt)
#            k = k + vx[i]**2 + vy[i]**2 + vz[i]**2
#        k = .5*k
        temp = 2*k/(3*n)
    #  create time series of values
        e = k + u
        un[j] = u/n
        kn[j] = k/n
        en[j] = e/n
        tn[j] = temp
        pn[j] = density*temp + w/(3*vol)

        cvv[j,:] = vxo * vx + vyo * vy + vzo * vz

    # reset positions for next time step
        xold[:] = x
        yold[:] = y
        zold[:] = z
        x[:] = xnew
        y[:] = ynew
        z[:] = znew
#        for i in range(n):
#            xold[i] = x[i]
#            yold[i] = y[i]
#            zold[i] = z[i]
#            x[i] = xnew[i]
#            y[i] = ynew[i]
#            z[i] = znew[i]
    # calculate force and energy at new positions for next cycle
        u, w, fx, fy, fz = fLJsum(a, n, rc, x, y, z)

    return un, kn, en, tn, pn, n, xnew, ynew, znew, cvv

if __name__ == '__main__':
    dt = 1e-5
    nc = 3
    dens = 0.9
    T = 0.2
    nsteps = 100000
    un, kn, en, tn, pn, n, x, y, z, cvv = MDLJ(nc, dens, T,nsteps,dt)
    figure()
    plot(en)
    title("en")
    show()
```

```
figure()
plot(un)
title("un")
show()
figure()
plot(kn)
title("kn")
show()
```

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 29 23:27:36 2021

@author: tom_m
"""

import numpy as np
from MDLJ import MDLJ

eps = 0.0104 # epsilon of Ar in eV
sig = 3.40 # sigma of Ar in Angstroms
Mr = 39.948 # molecular mass of Ar in g/mol

dt = 5 * 10**-15 # time step in seconds

eps *= 1.60218 * 10**-19 # conversion to J
sig *= 10**-10 # conversion to m
Mr /= 1000 # conversion to kg/mol

m = Mr / (6.022 * 10**23) # mass of a single atom in kg

t0 = np.sqrt((m * sig**2) / eps)

dtr = dt / t0 # reduced time step

print('Time step in seconds =', dt)
print('Reduced time step =', dtr)

nc = 7 # number of repeat fcc unit cells
dens = 0.9 # density
Tin = 0.2 # initial temperature
steps = 300 # number of time steps

tests = 100 # number of time steps to be tried to find optimal one
e = np.zeros((steps, tests)) # array to store the system energies for
every
# time step at each iteration
merit = np.zeros(tests) # array to store the figure of merit
time = np.linspace(dtr, 0.1, tests) # timesteps to be tested

# returns the potential energy, kinetic energy, total energy, temperature
# pressure, number of atoms, x y and z coordinates and velocity
# autocorrelation function at each time step
u, k, e[:,0], T, p, n, x, y, z, cvv = MDLJ(nc, dens, Tin, steps, dtr)

e_avg = np.mean(e[:,0])
e_min = np.min(e[:,0])
e_max = np.max(e[:,0])

merit[0] = (e_max - e_min) / e_avg # fluctuation from average e

print('Average e for initial timestep =', e_avg)
print('Fluctuation from average e for initial timestep =', merit[0])

merit_f = 0 # value of final accepted fluctuation
dt_f = 0 # value of final accepted timestep
```

```
# will run through each timestep and calculate the fluctuation from the
average
for i in range(1, tests):

    u, k, e[:,i], T, p, n, x, y, z, cvv = MDLJ(nc, dens, Tin, steps,
time[i])

    e_avg = np.mean(e[:,i])
    e_min = np.min(e[:,i])
    e_max = np.max(e[:,i])

    merit[i] = (e_max - e_min) / e_avg

    # sets the final fluctuation and timestep as the ones that are still
less
    # than 10**-4, with the condition that if the fluctuation goes above
    # 10**-4 then the iteration stops

    if np.absolute(merit[i]) < 10**-4:

        merit_f = merit[i]
        dt_f = time[i]

    else:
        break

print('Optimised reduced timestep (s) =', dt_f)
print('Fluctuation from average e for optimised timestep =', merit_f)

dto = dt_f * t0

print('Optimised timestep (s) =', dto)
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 30 17:40:22 2021

@author: tom_m
"""

import numpy as np
from MDLJ import MDLJ
import matplotlib.pyplot as plt


eps = 0.0104 # epsilon of Ar in eV
sig = 3.40 # sigma of Ar in Angstroms
Mr = 39.948 # molecular mass of Ar in g/mol
rho = 1.616 # mass density of solid Ar in g/cm^3
kb = 1.381 * 10**-23

dtr = 0.01 # optimised reduced time step in seconds, approximated due to
# fluctuations

eps *= 1.60218 * 10**-19 # conversion to J
sig *= 10**-10 # conversion to m
Mr /= 1000 # conversion to kg/mol
rho *= 10**3 # conversion to kg/m^3

m = Mr / (6.022 * 10**23) # mass of a single atom in kg

dens = (rho / Mr) * 6.022 * 10**23 # atomic density in atoms/m^3
rdens = dens * sig**3 # reduced density
print('Reduced density of the solid =', rdens)

nc = 7 # number of repeat fcc unit cells
Tin = 273.15 - 195 # initial temperature in K
tin = (kb * Tin) / eps # reduced temperature
steps = 3000 # number of time steps
cut = 250 # first number of timesteps to be cutoff


# returns the potential energy, kinetic energy, total energy, temperature
# pressure, number of atoms, x y and z coordinates and velocity
# autocorrelation function at each time step
u, k, e, T, p, n, X, Y, Z, cvv = MDLJ(nc, rdens, tin, steps, dtr)

# x values to plot functions with cutoff
x = np.linspace(cut, steps, (steps - cut)+1)

plt.figure()
plt.plot(u/n)
plt.title('Potential energy')
plt.ylabel('u* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(k/n)
plt.title('Kinetic energy')
plt.ylabel('k* / N')
plt.xlabel('Timestep')
plt.figure()
plt.plot(e/n)
plt.title('Total energy')
plt.ylabel('e* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(T)
plt.title('Temperature')
plt.ylabel('T*')
plt.xlabel('Timestep')

plt.figure()
plt.plot(p)
plt.title('Pressure')
plt.ylabel('p*')
plt.xlabel('Timestep')


plt.figure()
plt.plot(x, u[(cut-2):-1]/n)
plt.title('Potential energy')
plt.ylabel('u* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, k[(cut-2):-1]/n)
plt.title('Kinetic energy')
plt.ylabel('k* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, e[(cut-2):-1]/n)
plt.title('Total energy')
plt.ylabel('e* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, T[(cut-2):-1])
plt.title('Temperature')
plt.ylabel('T*')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, p[(cut-2):-1])
plt.title('Pressure')
plt.ylabel('p*')
plt.xlabel('Timestep')

u_avg = np.mean(u[(cut-2):-1])
u_min = np.min(u[(cut-2):-1])
u_max = np.max(u[(cut-2):-1])

u_fluc = (u_max - u_min) / u_avg # fluctuation from average u

print('Average potential energy =', u_avg)
print('Fluctuation of potential energy from average =', u_fluc)
```

```
k_avg = np.mean(k[(cut-2):-1])
k_min = np.min(k[(cut-2):-1])
k_max = np.max(k[(cut-2):-1])

k_fluc = (k_max - k_min) / k_avg # fluctuation from average k

print('Average kinetic energy =', k_avg)
print('Fluctuation of kinetic energy from average =', k_fluc)

e_avg = np.mean(e[(cut-2):-1])
e_min = np.min(e[(cut-2):-1])
e_max = np.max(e[(cut-2):-1])

e_fluc = (e_max - e_min) / e_avg # fluctuation from average e

print('Average total energy =', e_avg)
print('Fluctuation of total energy from average =', e_fluc)

T_avg = np.mean(T[(cut-2):-1])
T_min = np.min(T[(cut-2):-1])
T_max = np.max(T[(cut-2):-1])

T_fluc = (T_max - T_min) / T_avg # fluctuation from average T

print('Average temperature =', T_avg)
print('Fluctuation of temperature from average =', T_fluc)

p_avg = np.mean(p[(cut-2):-1])
p_min = np.min(p[(cut-2):-1])
p_max = np.max(p[(cut-2):-1])

p_fluc = (p_max - p_min) / p_avg # fluctuation from average p

print('Average pressure =', p_avg)
print('Fluctuation of pressure from average =', p_fluc)


"""
Below is for question 8

"""

# number of bins for averaging
nbins = (steps - cut) // cut


binsu = np.zeros((nbins, cut))
binsk = np.zeros((nbins, cut))
binse = np.zeros((nbins, cut))
binsT = np.zeros((nbins, cut))
binsp = np.zeros((nbins, cut))

# divides the outputs into bins
for i in range(nbins):

    binsu[i,:] = u[cut*(i+1):cut*(i+2)]
    binsk[i,:] = k[cut*(i+1):cut*(i+2)]
    binse[i,:] = e[cut*(i+1):cut*(i+2)]
    binsT[i,:] = T[cut*(i+1):cut*(i+2)]
    binsp[i,:] = p[cut*(i+1):cut*(i+2)]

# averages the outputs in each bin
avgbu = np.mean(binsu, axis=1)
avgbk = np.mean(binsk, axis=1)
avgbe = np.mean(binse, axis=1)
avgbT = np.mean(binsT, axis=1)
avgbp = np.mean(binsp, axis=1)

# calculates the variance of the outputs
varu = np.mean(np.square(avgbu)) - np.square(np.mean(avgbu))
vark = np.mean(np.square(avgbk)) - np.square(np.mean(avgbk))
vare = np.mean(np.square(avgbe)) - np.square(np.mean(avgbe))
varT = np.mean(np.square(avgbT)) - np.square(np.mean(avgbT))
varp = np.mean(np.square(avgbp)) - np.square(np.mean(avgbp))

# calculates the average of the outputs
avgu = np.mean(avgbu)
avgk = np.mean(avgbk)
avge = np.mean(avgbe)
avgT = np.mean(avgbT)
avgp = np.mean(avgbp)

print('Variance of potential energy =', varu)
print('Variance of kinetic energy =', vark)
print('Variance of total energy =', vare)
print('Variance of temperature =', varT)
print('Variance of pressure =', varp)


"""
Below is for question 9

"""

def gr(nbin, a, n, x, y, z):
    """
    calculates the radial distribution function

    Parameters
    ----------
    nbin : number of bins
    a : length of unit cell (lattice parameter)
    n : number of atoms
    x : x coordinates of atoms
    y : y coordinates of atoms
    z : z coordinates of atoms

    """

    rc = a/2
    xb = rc/nbin
    g = np.zeros((nbin, 1))
    bp = np.zeros((nbin, 1))
    ng = np.zeros((nbin, 1))
    # bin for all the distances
    for i in range(n-1):
        for j in range(i+1, n):

            dx = x[j] - x[i]
            dy = y[j] - y[i]
            dz = z[j] - z[i]
            dx = dx - np.round(dx)
            dy = dy - np.round(dy)
            dz = dz - np.round(dz)
            dist = a * np.sqrt(dx**2 + dy**2 + dz**2)

            if dist <= rc:
                ib = np.floor(dist//xb)
                ib = int(ib)
                g[ib] = g[ib] + 1

    factor = 2 * a**3 / (4 * np.pi * n**2 * xb)

    for k in range(nbin):
        bp[k] = (k + 3/2) * xb
        ng[k] = factor * g[k] / ((k+1) * xb)**2

    return bp, ng

a = 5.486

b, g = gr(20, a, n, X, Y, Z)

plt.figure()
plt.plot(b, g)
plt.ylabel('g(r)')
plt.xlabel('r*')


"""
Below is for question 11

"""

# averages the velocity products for all atoms at each timestep
Cvv = np.mean(cvv, axis=1)

Cvv *= (m / 3 * kb * Tin)

plt.figure()
plt.plot(Cvv)
plt.ylabel('Cvv')
plt.xlabel('Timestep')
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 30 20:23:52 2021

@author: tom_m
"""

import numpy as np
from MDLJ import MDLJ
import matplotlib.pyplot as plt


eps = 0.0104 # epsilon of Ar in eV
sig = 3.40 # sigma of Ar in Angstroms
Mr = 39.948 # molecular mass of Ar in g/mol
rho = 1.3 # mass density of liquid Ar in g/cm^3
kb = 1.381 * 10**-23

dtr = 0.01 # optimised reduced time step in seconds, approximated due to
# fluctuations

eps *= 1.60218 * 10**-19 # conversion to J
sig *= 10**-10 # conversion to m
Mr /= 1000 # conversion to kg/mol
rho *= 10**3 # conversion to kg/m^3

m = Mr / (6.022 * 10**23) # mass of a single atom in kg

dens = (rho / Mr) * 6.022 * 10**23 # atomic density in atoms/m^3
rdens = dens * sig**3
print('Reduced density of the liquid =', rdens)

t0 = np.sqrt((m * sig**2) / eps)

#dtr = dt / t0 # reduced time step

nc = 7 # number of repeat fcc unit cells
Tin = 273.15 - 187 # initial temperature in K
tin = (kb * Tin) / eps # reduced temperature
steps = 5000 # number of time steps
cut = 1500


# returns the potential energy, kinetic energy, total energy, temperature
# pressure, number of atoms, x y and z coordinates and velocity
# autocorrelation function at each time step
u, k, e, T, p, n, X, Y, Z, cvv = MDLJ(nc, rdens, tin, steps, dtr)

# x values to plot functions with cutoff
x = np.linspace(cut, steps, (steps - cut) + 1)

plt.figure()
plt.plot(u/n)
plt.title('Potential energy')
plt.ylabel('u* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(k/n)
plt.title('Kinetic energy')
plt.ylabel('k* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(e/n)
plt.title('Total energy')
plt.ylabel('e* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(T)
plt.title('Temperature')
plt.ylabel('T*')
plt.xlabel('Timestep')

plt.figure()
plt.plot(p)
plt.title('Pressure')
plt.ylabel('p*')
plt.xlabel('Timestep')


plt.figure()
plt.plot(x, u[(cut-2):-1]/n)
plt.title('Potential energy')
plt.ylabel('u* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, k[(cut-2):-1]/n)
plt.title('Kinetic energy')
plt.ylabel('k* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, e[(cut-2):-1]/n)
plt.title('Total energy')
plt.ylabel('e* / N')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, T[(cut-2):-1])
plt.title('Temperature')
plt.ylabel('T*')
plt.xlabel('Timestep')

plt.figure()
plt.plot(x, p[(cut-2):-1])
plt.title('Pressure')
plt.ylabel('p*')
plt.xlabel('Timestep')

u_avg = np.mean(u[(cut-2):-1])
u_min = np.min(u[(cut-2):-1])
u_max = np.max(u[(cut-2):-1])
```

```python
u_fluc = (u_max - u_min) / u_avg # fluctuation from average u

print('Average potential energy =', u_avg)
print('Fluctuation of potential energy from average =', u_fluc)

k_avg = np.mean(k[(cut-2):-1])
k_min = np.min(k[(cut-2):-1])
k_max = np.max(k[(cut-2):-1])

k_fluc = (k_max - k_min) / k_avg # fluctuation from average k

print('Average kinetic energy =', k_avg)
print('Fluctuation of kinetic energy from average =', k_fluc)

e_avg = np.mean(e[(cut-2):-1])
e_min = np.min(e[(cut-2):-1])
e_max = np.max(e[(cut-2):-1])

e_fluc = (e_max - e_min) / e_avg # fluctuation from average e

print('Average total energy =', e_avg)
print('Fluctuation of total energy from average =', e_fluc)

T_avg = np.mean(T[(cut-2):-1])
T_min = np.min(T[(cut-2):-1])
T_max = np.max(T[(cut-2):-1])

T_fluc = (T_max - T_min) / T_avg # fluctuation from average T

print('Average temperature =', T_avg)
print('Fluctuation of temperature from average =', T_fluc)

p_avg = np.mean(p[(cut-2):-1])
p_min = np.min(p[(cut-2):-1])
p_max = np.max(p[(cut-2):-1])

p_fluc = (p_max - p_min) / p_avg # fluctuation from average p

print('Average pressure =', p_avg)
print('Fluctuation of pressure from average =', p_fluc)


"""
Below is for question 8

"""

# number of bins for averaging
width = cut // 6
nbins = (steps - cut) // width


binsu = np.zeros((nbins, width))
binsk = np.zeros((nbins, width))
binse = np.zeros((nbins, width))
binsT = np.zeros((nbins, width))
binsp = np.zeros((nbins, width))
```

```
# divides the outputs into bins
for i in range(nbins):

    binsu[i,:] = u[width*(i+1):width*(i+2)]
    binsk[i,:] = k[width*(i+1):width*(i+2)]
    binse[i,:] = e[width*(i+1):width*(i+2)]
    binsT[i,:] = T[width*(i+1):width*(i+2)]
    binsp[i,:] = p[width*(i+1):width*(i+2)]

# averages the outputs in each bin
avgbu = np.mean(binsu, axis=1)
avgbk = np.mean(binsk, axis=1)
avgbe = np.mean(binse, axis=1)
avgbT = np.mean(binsT, axis=1)
avgbp = np.mean(binsp, axis=1)

# calculates the variance of the outputs
varu = np.mean(np.square(avgbu)) - np.square(np.mean(avgbu))
vark = np.mean(np.square(avgbk)) - np.square(np.mean(avgbk))
vare = np.mean(np.square(avgbe)) - np.square(np.mean(avgbe))
varT = np.mean(np.square(avgbT)) - np.square(np.mean(avgbT))
varp = np.mean(np.square(avgbp)) - np.square(np.mean(avgbp))

# calculates the average of the outputs
avgu = np.mean(avgbu)
avgk = np.mean(avgbk)
avge = np.mean(avgbe)
avgT = np.mean(avgbT)
avgp = np.mean(avgbp)

print('Variance of potential energy =', varu)
print('Variance of kinetic energy =', vark)
print('Variance of total energy =', vare)
print('Variance of temperature =', varT)
print('Variance of pressure =', varp)



"""
Below is for question 9

"""

def gr(nbin, a, n, x, y, z):
    """
    calculates the radial distribution function

    Parameters
    ----------
    nbin : number of bins
    a : length of unit cell (lattice parameter)
    n : number of atoms
    x : x coordinates of atoms
    y : y coordinates of atoms
    z : z coordinates of atoms

    """
    rc = a/2
    xb = rc/nbin
    g = np.zeros((nbin, 1))
    bp = np.zeros((nbin, 1))
    ng = np.zeros((nbin, 1))

    # bin for all the distances
    for i in range(n-1):
        for j in range(i+1, n):

            dx = x[j] - x[i]
            dy = y[j] - y[i]
            dz = z[j] - z[i]
            dx = dx - np.round(dx)
            dy = dy - np.round(dy)
            dz = dz - np.round(dz)
            dist = a * np.sqrt(dx**2 + dy**2 + dz**2)

            if dist <= rc:
                ib = np.floor(dist//xb)
                ib = int(ib)
                g[ib] = g[ib] + 1

    factor = 2 * a**3 / (4 * np.pi * n**2 * xb)

    for k in range(nbin):
        bp[k] = (k + 3/2) * xb
        ng[k] = factor * g[k] / ((k+1) * xb)**2

    return bp, ng

a = 5.486

b, g = gr(20, a, n, X, Y, Z)

plt.figure()
plt.plot(b, g)
plt.ylabel('g(r)')
plt.xlabel('r*')


"""
Below is for question 11

"""

# averages the velocity products for all atoms at each timestep
Cvv = np.mean(cvv, axis=1)

Cvv *= (m / 3 * kb * Tin)

plt.figure()
plt.plot(Cvv)
plt.ylabel('Cvv')
plt.xlabel('Timestep')
```