MD

S. Bechan(4146425), K. Elgin(4163389), Z. Ramlakhan(4170229)

March 1, 2015

Abstract

As part of the International Course Computational Physics of the Master Applied Physics program, students are required to simulate the molecular dynamics of Argon molecules. In this report the movement and the interactions of these molecules are simulated in a box. Simulations were based on Verlet's algorithms, which provides iterative methods to determine the positions and velocities of the particles. From these quantities parameters such as the specific heat, pressure and the pair correlation were calculated and plotted.

1 Introduction

Molecular dynamics (MD) is a field of study which uses computer simulatios to depict the movement and interactions of a number of particles confined to a box. The trajectories of the particles are determined by numerically solving Newton's equations of motion. The potential energy $U(r)$ of each particle is given by the *Lennard-Jones potential*, also known as the 6-12 potential,

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

with the ϵ the depth of the potential well and r the distance between two particles. It is evident then that σ represents the distance between two particles such that $U(\sigma) = 0$.
The force $\mathbf{F}$ the particles exert on each other is given by

$$\mathbf{F} = -\nabla U(r) \tag{2}$$

The simulation's algorithm consists of a few steps. First the particles are given an initial position and a time step is defined. The forces the particles exert on each other are then calculated from which the acceleration can also be found. The equations of motion yield the new position as well as velocity of the particles. At this point time is moved forward with the time step. The forces are calculated again and the process repeats itself for a desirable number

of iterations. This method is called the Verlet algorithm.

The following is a report of MD simulation of argon gas in the low temperature limit. The report in particular elaborates on translation of the physical formulas to a code and explaining the steps taken.

2 Simulation

Section 1 described the MD algorithm. This section will explain in more detail how to get from physical formulas to a MD simulation code.

2.1 Initial position

Argon particles are subjected to a face-centered cubic (FCC) lattice which means there are four lattice points per unit cell. The initial positions of n particles can be generated by first defining a unit cell with Cartesian coordinates $(0,0,0),(0.5,0.5,0),(0.5,0,0.5),(0,0.5,0.5)$. Other unit cells are then "stacked" on top of the first cell in the x,y,z-directions such that n is equal to four times the number of stacked cells in a single direction raised to the power three.
The system of particles has a total volume V of

$$V = \frac{nm}{\rho} = \frac{n}{\rho} = L^3 \Leftrightarrow L = \left(\frac{n}{\rho}\right)^{\frac{1}{3}}$$

with the particle mass m and density ρ. m is set to 1 to reduce computational effort. Taking the cubic root of V results in the dimension of the box L which confines the particles. The result is an array describing the initial positions of n particles in a fcc lattice confined to a box of dimension L.

2.2 Initial velocity

The particle speed v is defined by the Maxwell-Boltzmann distribution

$$\begin{aligned} f(v) &= \sqrt{\left(\frac{m}{2\pi k_B T}\right)^3} 4\pi v^2 e^{-\frac{mv^2}{2k_B T}} \\ &= \sqrt{\left(\frac{1}{2\pi T}\right)^3} 4\pi v^2 e^{-\frac{v^2}{2T}} \end{aligned} \tag{3}$$

with the Boltzmann constant k_B and the temperature T. Like m, k_B is also set to 1. To generate the random speeds of the particles the cumulative distribution function CDF is calculated for a range of speeds. CDF is given by the integral over $f(v)$ from $-\infty$ to v, the result of which is

$$CDF = erf\left(\frac{v}{\sqrt{2T}}\right) - \sqrt{\frac{2}{\pi}} v e^{-\frac{v^2}{2T}} \tag{4}$$

CDF is defined over the range $[0,1]$, so by generating a random number from that range and interpolating it against CDF yields a particle random speed.
The speeds are given a direction by multiplying them with random spherical coordinates which translates the velocity vectors to Cartesian coordinates.

2.3 Force

The potential $U(r)$ was already given in (1). σ and ϵ are set to 1 as well. Furthermore, $U(r)$ can be represented as the sum of pairwise interactions

$$\begin{aligned} U(r) &= \sum_{i=1}^{n} \sum_{i<j}^{n} u(r_{ij}) \\ &= \sum_{i=1}^{n} \sum_{i<j}^{n} 4 \left(\frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^{6}} \right) \end{aligned} \tag{5}$$

with r_{ij} the magnitude of the vector $\boldsymbol{r_{ij} = r_i - r_j}$.The total force working on particle i is given by taking the negative gradient of (5),

$$\begin{aligned} \mathbf{F}_i &= \sum_{i \neq j}^{n} \mathbf{f}_{ij} = \sum_{i \neq j}^{n} -\frac{du(r_{ij})}{dr_{ij}} \cdot \frac{\mathbf{r}_{ij}}{r_{ij}} \\ &= \sum_{i \neq j}^{n} -24 \left(\frac{2}{r_{ij}^{7}} - \frac{1}{r_{ij}^{4}} \right) \cdot \mathbf{r}_{ij} \end{aligned} \tag{6}$$

2.4 Iteration

The positions and velocities of the particles are updated every iteration and are done so by obeying the equations of motion. Suppose $r(t), \mathbf{v}(t)$ represent the initial position and velocity of a particle, respectively. Then $r(t+\Delta t), \mathbf{v}(t+\Delta t)$ represent that particle's position and velocity after one iteration. The relations between these quantities are given by

$$\begin{aligned} \mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \mathbf{a}(t)\Delta t \\ &= \mathbf{v}(t) + \mathbf{F}(t)\Delta t \end{aligned} \tag{7}$$

$$\begin{aligned} r(t+\Delta t) &= r(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 \\ &= r(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{F}(t)\Delta t^2 \end{aligned} \tag{8}$$

This method is called the Verlet algorithm.

3 Physical parameters

From the simulation a number of physical parameters can be calculated.

- The total energy E_t is given by the sum of the potential, given by (5), and the kinetic energy of all particles and is a conserved parameter. The kinetic energy of the system is given by

$$E_k = \frac{1}{2} \sum_{i=1}^{n} \mathbf{v}_i^2 \tag{9}$$

Via the equipartition theorem, $E_k = \frac{3}{2}k_B NT$, the relation between the velocity of the particles and the temperature of the system becomes clear i.e.

$$T = \frac{2k_B E_k}{3n} \tag{10}$$

- The pair correlation function (or radial distribution function)

$$g(r) = 4\pi r_{ij}^2 \frac{n}{V} dr \tag{11}$$

represents the probability of two particles being separated a distance r. Due to repulsive forces, there is a minimal distance the particles can be separated from each other. On the other hand, attractive forces prevent the particles from getting to far from each other. (11) therefore shows the range of possible separation distances and the probability of those separation distances.

- The pressure can be derived from the virial theorem which, for a pair potential, is given by

$$PV = nk_B T + \frac{1}{3}\sum_{i<j}^{n} \mathbf{r}_{ij} \cdot \mathbf{f}_{ij} \tag{12}$$

In view of (9),(12) is rewritten in a form containing only variables specific to the simulation

$$P = \frac{\rho}{3n}\left(2E_k + \sum_{i<j}^{n} \mathbf{r}_{ij} \cdot \mathbf{f}_{ij}\right) \tag{13}$$

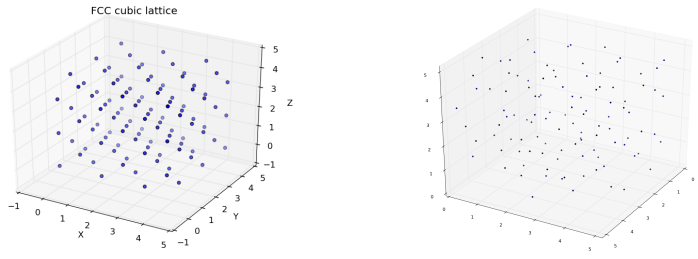
- Specific heat can be calculated with Lebowitz's formula

$$\frac{<\delta E_k^2>}{E_k^2} = \frac{2}{3n}(\left(1 - \frac{3n}{2c_v}\right) \tag{14}$$

4 Results

4.1 Lattice and movement

Figure 1a displays the initial position of the particles. Here, a lattice can clearly be seen. After a certain number of iterations, the system looks like it does in figure 1b. There is no resemblance to the original lattice at all.



(a) FCC lattice at $t = 0$.

(b) Particle movement after iteration.

Figure 1: The initial system and the system after iteration ($n = 108$).

4.2 Energy conservation

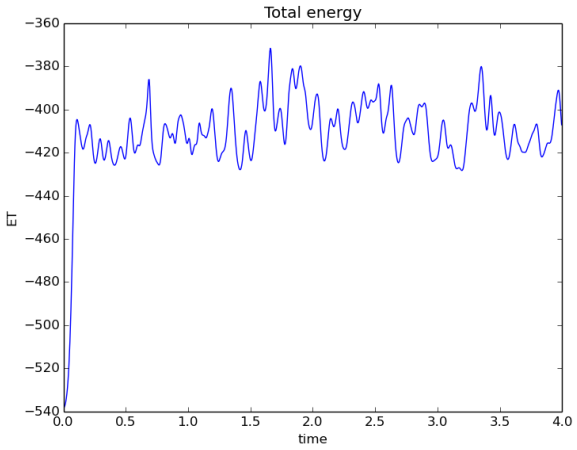


Figure 2: The jump in the graph is due to initializing the temperature.

Figure 2 displays the total energy at 1000 iterations. Remarkable is the initial jump in the energy early in the simulation. The reason for this is the initialization of the temperature. The temperature that is offered as input is not the instantaneous temperature. Rather, it has to be rescaled a number of times with a factor to achieve the desired temperature. This also has an influence on the pressure.

4.3 Pressure

As a result of temperature initialization , the pressure decreases quickly and oscillations remains fairly constant for the rest of the simulation.

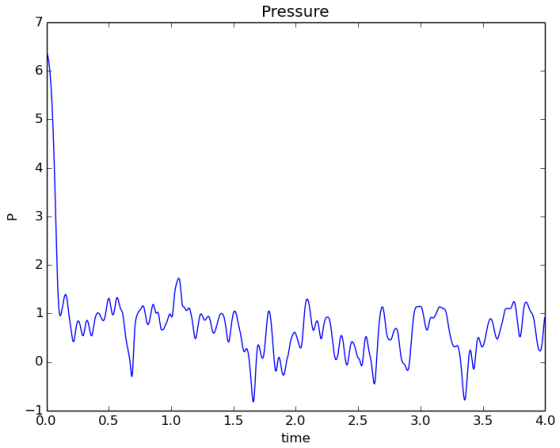
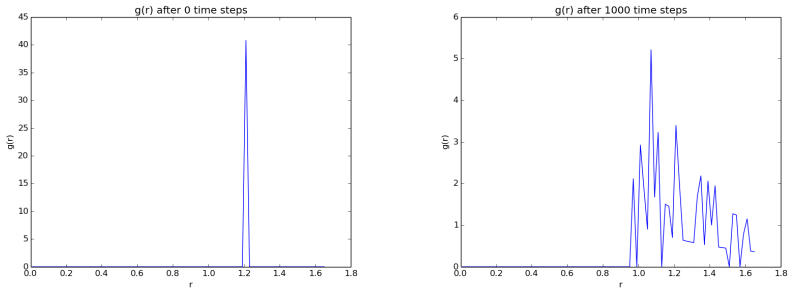


Figure 3: The pressure as function of the time



(a) $t = 0$.

(b) After 1000 iterations.

Figure 4: The initial pair correlation distribution and after 1000 iterations.

4.4 Pair correlation function

Figure 4 shows the results of the pair correlation distribution. Figure 4a shows g at $t = 0$. At this point the particles are confined to the FCC lattice, so the probability to find a particle at the lattice constant is maximal. After 1000 iteration g evolves. The probability to find a particle close to another, i.e. $r < 1.2$ is zero. This is due to the repulsive forces. On the other hand, the probability to find particle far away from another one is also small if not zero. This is caused by the attractive forces.

4.5 Specific heat

The specific heat also seems to oscillate around a constant value as can be seen in figure 5

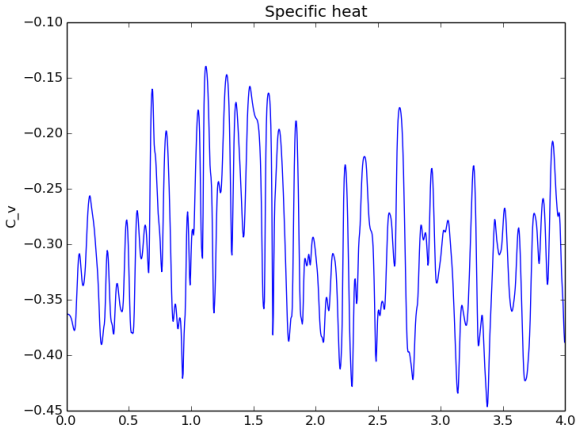


Figure 5: Specific heat with 1000 iterations.

5 Conclusion

In this report the dynamics of argon molecules confined to a finite box was simulated by employing Verlet's algorithm. It was found that the particles still moved quite locally due to interactions, but they weren't restricted to their lattice positions anymore. Quantities such as pressure, correlation function and specific all fluctuate with temperature. If the number of iterations is large enough the oscillations have constant mean. Quantities which can be further investigated are the diffusion constant and statistical errors.

6 Appendix

```
# Code courtesy of github user cfinch, code location on
    github: https://github.com/cfinch/Shocksolution_Examples/
    blob/master/PairCorrelation/paircorrelation.py
def PairCorrelationFunction_3D(x,y,z,S,rMax,dr):
    """Compute the three-dimensional pair correlation
        function for a set of
    spherical particles contained in a cube with side length
         S.  This simple
    function finds reference particles such that a sphere of
         radius rMax drawn
    around the particle will fit entirely within the cube,
        eliminating the need
    to compensate for edge effects.  If no such particles
        exist, an error is
    returned.  Try a smaller rMax...or write some code to
        handle edge effects! ;)

    Arguments:
        x               an array of x positions of centers
            of particles
```

```
        y               an array of y positions of centers
            of particles
        z               an array of z positions of centers
            of particles
        S               length of each side of the cube in
            space
        rMax            outer diameter of largest spherical
            shell
        dr              increment for increasing radius of
            spherical shell

    Returns a tuple: (g, radii, interior_x, interior_y,
        interior_z)
        g(r)            a numpy array containing the
            correlation function g(r)
        radii           a numpy array containing the radii
            of the
                        spherical shells used to compute g(r
                            )
        interior_x      x coordinates of reference particles
        interior_y      y coordinates of reference particles
        interior_z      z coordinates of reference particles
    """
    from numpy import zeros, sqrt, where, pi, average,
        arange, histogram

    # Find particles which are close enough to the cube
        center that a sphere of radius
    # rMax will not cross any face of the cube
    bools1 = x>rMax
    bools2 = x<(S-rMax)
    bools3 = y>rMax
    bools4 = y<(S-rMax)
    bools5 = z>rMax
    bools6 = z<(S-rMax)

    interior_indices, = where(bools1*bools2*bools3*bools4*
        bools5*bools6)
    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 S.  Decrease rMax\
                or increase the size of the cube.")

    edges = arange(0., rMax+1.1*dr, dr)
    num_increments = len(edges)-1
    g = zeros([num_interior_particles, num_increments])
    radii = zeros(num_increments)
    numberDensity = len(x)/S**3
```

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

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

    # Average g(r) for all interior particles and compute
        radii
    g_average = 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] = average(g[:,i])/(4./3.*pi*(rOuter**3
            - rInner**3))

    return (g_average, radii, x[interior_indices], y[
        interior_indices], z[interior_indices])
    # Number of particles in shell/total number of particles
        /volume of shell/number density
    # shell volume = 4/3*pi(r_outer**3-r_inner**3)
####
```

```
# Rmatrixfile.py --- Here we have a function that creates
    our initial R and V matrices for dens and T
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.special import erf
from scipy.interpolate import interp1d as interp
from cdf import MB_ICDF
from velocitygen import velgen
def init_matrix(dens,T):
    rows=3
    i=0
    n=4*rows**3
    vol=n/dens
    L=vol**(1.0/3)
    R=np.zeros((n,3),dtype=float)
    for x in range(0,rows):
        for y in range(0,rows):
            for z in range (0,rows):
                R[i:i+4,:]=np.array([[0+x,0+y,0+z],[0.5+x
                    ,0.5+y,0+z],[0.5+x,0+y,0.5+z],[0+x,0.5+y
                    ,0.5+z]])
                i=i+4
    R=R*L/rows
```

```
    V = velgen(n,T,L)
    return R,V,L,n
```

```
# cdf.py  --- Cumulative Maxwell-Boltzmann
#distribution function and its inverse
import numpy as np
from scipy.special import erf
from scipy.interpolate import interp1d as interp
def MB_ICDF(T,L):
#    Cumulative Distribution function of the
#    Maxwell-Boltzmann speed distribution
    a = np.sqrt(T)
    v = np.arange(0,L,0.1)
    cdf = erf(v/(np.sqrt(2)*a)) - np.sqrt(2/np.pi)* v* np.
        exp(-v**2/(2*a**2))/a
    icdf = interp(cdf,v)
    return icdf
```

```
# velocitygen.py --- Generates particle velocities V
import numpy as np
from cdf import MB_ICDF
from scipy.interpolate import interp1d as interp
def velgen(n,T,L):
    rand_nums = np.random.random(n)
    icdf = MB_ICDF(T,L)
    Vmag = icdf(rand_nums)
# Spherical polar coords - generate random angle for
    velocity vector, uniformly distributed over the surface
    of a sphere
    theta = np.arccos(np.random.uniform(-1,1,n))
    phi = np.random.uniform(0,2*np.pi,n)
    v = np.array([np.sin(theta)*np.cos(phi), np.sin(theta) *
         np.sin(phi), np.cos(theta)])
    # convert to cartesian units
    V = np.transpose(Vmag*v)
    return V
```

```
! forcetestken.f95 --- Calculates forces
subroutine Force(R,tf,n,L)
implicit none
real(8), intent(in) :: L
integer, intent(in) :: n
real(8), intent(in) :: R(n, 3)
real(8), intent(inout) :: tf(n, 3)
!f2py intent(in, out) tf
real(8) :: dr(3), dr2, F
real(8), parameter :: rmax = 3.2_8
integer :: i, j
tf = 0._8
do i = 0, n
    do j = 1, i-1
        dr = R(i,:) - R(j,:)
        dr = dr - nint(dr/L)*L
```

```
        dr2 = sum(dr**2)
        if (dr2<rmax**2) then
            F=24*(2/dr2**7 - 1/dr2**4)
            if (F>1000._8) print *, i, j, dr2
            tf(i,:) = tf(i,:) + F*dr
            tf(j,:) = tf(j,:) - F*dr
        end if
    end do
end do
end subroutine
```

```
# update.py --- updates R,V after every iter
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from forcetestken import force

def update_func(n,L,R,V,dt,tf,T):
    tf=force(R,tf,L,[n])
    target=T
    R=(R+V*dt)%L
    V+=tf*dt
    avV=np.sum(V**2)/n
    temp=1/3.0*avV
    lada=math.sqrt(target/temp)
    V=lada*V
    return R,V
```

```
# josplot.py --- Animation
import numpy as np
#plotting
import matplotlib.pyplot as plt
import matplotlib.animation as animation
# 3D plotting
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d.art3d import juggle_axes
class AnimatedScatter(object):
    """
    Class for particle animation
    constructor takes as arguments
    -- numpoints: the number of points
    -- box_len: the side of the (cubic) box
    -- pos: numpy array of shape [numpoints, 3] containing
        the position coordinates
    -- mom: numpy array of shape [numpoints, 3] containing
        the momentum coordinates
    -- part_update: function which calculates the new
        particle positions
    This function must output the updated arrays pos and mom
    -- args: the function part_update takes as arguments
        numpoints, box_len, pos, mom, *args
    Example: anim_md.AnimatedScatter(n, box_len, pos,
        simulate, mom, n_t, dt)
```

```
    where 'simulate' is defined as
    def simulate(n, box_len, pos, mom, arglist):
    ...
    ...
    return pos, mom
    """
    def __init__(self, n, L, R, V, update_func, dt, tf,T):
        self.numpoints = n
        self.pos = R
        self.mom = V
        self.box_len = L
        self.arglist = dt, tf, T
        self.stream = self.data_stream()
        self.angle = 30
        self.part_update = update_func
        self.fig = plt.figure(figsize=(16,12))
        self.FLOOR = 0.0
        self.CEILING = self.box_len
        self.ax = self.fig.add_subplot(111,projection = '3d'
            )
        self.ani = animation.FuncAnimation(self.fig, self.
            update, interval=1,init_func=self.setup_plot,
            blit=True)

    def setup_plot(self):
        """ Set world coordinates, colors, symbol siez ('s')
             """
        x, y, z = next(self.stream)
        c = ['b']
        self.scat = self.ax.scatter(x, y, z,c=c, s=10,
            animated=True)
        self.ax.set_xlim3d(self.FLOOR, self.CEILING)
        self.ax.set_ylim3d(self.FLOOR, self.CEILING)
        self.ax.set_zlim3d(self.FLOOR, self.CEILING)
        return self.scat,
    def data_stream(self):
        """
        Calls particle update routine, copies it to the
            relevant section of the 'data' array which
        is then yielded
        """
        self.pos, self.outlist = self.part_update(self.
            numpoints, self.box_len, self.pos, self.mom, *
            self.arglist)
        data = np.transpose(self.pos)
        while True:
            self.pos, self.mom = self.part_update(self.
                numpoints, self.box_len, self.pos, self.mom,
                *self.arglist)
            data[:3, :] = np.transpose(self.pos)
            yield data
    def update(self, i):
        """ Use new particle position for drawing next frame
             """
```

```
        data = next(self.stream)
        data = np.transpose(data)
        self.scat._offsets3d = juggle_axes(data[:,0],data
            [:,1],data[:,2], 'z')
        self.ax.view_init(30,self.angle)
        plt.draw()
        return self.scat,
    def show(self):
        plt.show()
```

```
# MD.py --- Runs simulation
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from forcetestken import force
from Rmatrixfile import init_matrix
from update import update_func
from josplot import AnimatedScatter
from pot import pot
rho_c = 3.2
dens = 0.8
T = 1
R,V,L,n = init_matrix(dens,T)
dt = 0.004
tf = np.zeros((n,3))
a = AnimatedScatter(n, L, R, V, update_func, dt, tf, T)
a.show()
```

```
# Made by Craig Finch van shocksolution.com
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from forcetestken import force
from Rmatrixfile import init_matrix
from update import update_func
from josplot import AnimatedScatter
from pot import pot
from paircorr3D import PairCorrelationFunction_3D
rho_c = 3.2
dens = 0.8
T = 1
R,V,L,n = init_matrix(dens,T)
dt = 0.004
tf = np.zeros((n,3))
nsteps=0
for i in range(0,nsteps):
    R,V=update_func(n,L,R,V,dt,tf,T)
x=R[:,0]
y=R[:,1]
z=R[:,2]
rMax=0.32*L
```

```
dr=0.02
g, radii, interior_x, interior_y, interior_z=
    PairCorrelationFunction_3D(x,y,z,L,rMax,dr)
fig = plt.figure()
plt.plot(radii,g)
plt.title('g(r) after %d time steps' % nsteps)
plt.xlabel('r')
plt.ylabel('g(r)')
plt.show()
#fig.figsave('g_1000.png')
```

```
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from forcetestken import force
from Rmatrixfile import init_matrix
from update import update_func
from josplot import AnimatedScatter
from pot import pot
from paircorr3D import PairCorrelationFunction_3D # Written
    by Craig Finch of shocksolution.com
from pressure import pressure
rho_c = 3.2
dens = 0.8
T = 1
R,V,L,n = init_matrix(dens,T)
dt = 0.004
tf = np.zeros((n,3))
nsteps=1000
cv=np.zeros((1,nsteps))
cv_error=np.zeros((1,nsteps))
P = np.zeros((1,nsteps))
U=np.zeros((1,nsteps))
v = np.zeros(nsteps)
ET = np.zeros(nsteps)
t=np.zeros((1,nsteps))
t[0,:]=np.arange(0,nsteps*dt,dt)
f =0
p=np.zeros((nsteps))
def calc_Cv(n,L,R,V,dt,tf,T,cv,error):
    Ekin=0.5*V**2
    delta=np.std(Ekin**2)
    Exp=np.mean(Ekin**2)
    X=delta/Exp
    cv_error[0,i]=np.var(Ekin)/math.sqrt(n)
    cv[0,i]=(3.0/2)*n*(1/(1-(3.0*n/2)*X))
    return cv,cv_error

def press_calc(R,V,n,rho_c):
    S =  0
    dr = np.zeros((1,3))
    for j in range(0,n):
        for x in range(1,j-1):
```

```
            dr[:] = np.subtract(R[x,:], R[j,:])
            dr = dr - L * np.rint(dr/L)
            rho = np.sum(dr**2)

            if (rho < rho_c**2):
                dU = (48*rho**(-7) - 24*rho**(-4))
                f = -dU * dr
                S = S + np.sum(dr*f)

    P[0,i] = dens/(3*n)  *(np.sum(V**2) + S)
    return P



for i in range(0,nsteps):
    R,V=update_func(n,L,R,V,dt,tf,T)
    cv,cv_error=calc_Cv(n,L,R,V,dt,tf,T,cv,cv_error)
    P = press_calc(R,V,n,rho_c)
    U = pot(R,L,U,n)
    v = 0.5 * np.sum(V**2)
    ET[i]= v + U
    #p[i] = pressure(R,V,L,dens,p,n)
#print  p
cv_error = np.mean(cv_error)
print cv_error
figcv = plt.figure()
plt.plot(np.transpose(t),np.transpose(cv))
plt.ylabel('C_v')
plt.title('Specific heat')
plt.show()

figet = plt.figure()
plt.title('Total energy')
plt.plot(np.transpose(t),ET)
plt.xlabel('time')
plt.ylabel('ET')
plt.show()

figcv.savefig('cv_1000.png')
figet.savefig('et_1000.png')
plt.plot(np.transpose(t),np.transpose(P))
plt.xlabel('time')
plt.ylabel('P')
plt.title('Pressure')
plt.show()
```