# PY 525 Homework 3

Jake Geiser

October 28, 2018

## Problem Results and Discussion

### 1

The model used is a periodic 10x10 box of 16 Lennard-Jones particles. The force acting on the particles is equal to the negative gradient of the L-J potential. Setting up the system, I filled in the columns and rows from the bottom and worked up and applying a starting velocity in the x direction between -3/2 and 3/2.
(a) To calculate the velocities, after each time step, the difference between current and previous positions was found and the total number was normalized in a histogram. The to compare to Maxwell-Boltzmann distribution, I plotted $P(v) = norm * \exp \frac{-v^2}{2k_bT}$ where v is the velocity and the norm values is the highest probability bin in the normalized distribution, which is about 0.5 as seen in Figure 1. As we can see, the calculated velocity distribution and theoretical distribution are in near agreement.
(b) The diffusion the system depends on the initial positions of the particles, for my system, from time $0 \rightarrow 4$ the diffusion constant is around 0.4(Figure 2), but for time $0 \rightarrow 10$ the diffusion constant decreases to 0.2. As time increases and the system reaches 'equilibrium' the diffusion constant goes to zero as there is no meaningful change to the expected $r^2(t)$ value as time progresses as seen in Figure 3.
(c) By limiting the radius that I am checking for the pair correlation function $g(r)$, to avoid the periodic boundary conditions, I get several density samples that suggest a gas due to being constant.

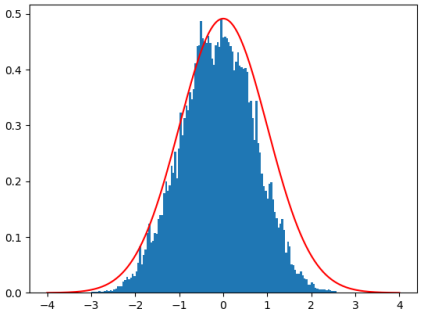


Figure 1: Velocity Distribution

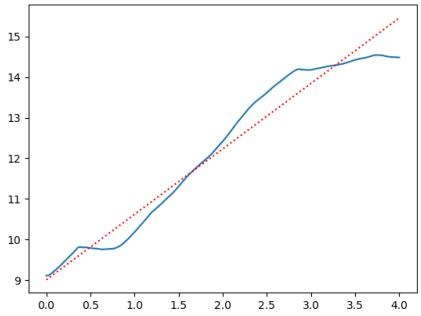


Figure 2: $\langle r^2(t) \rangle$ vs time for 4s

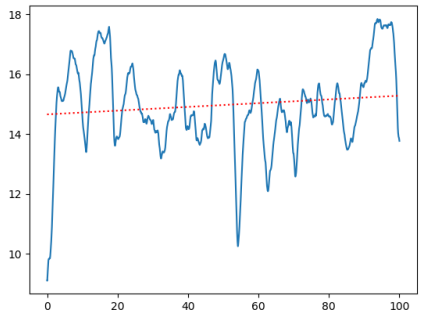


Figure 3: $\langle r^2(t) \rangle$ vs time for 100s

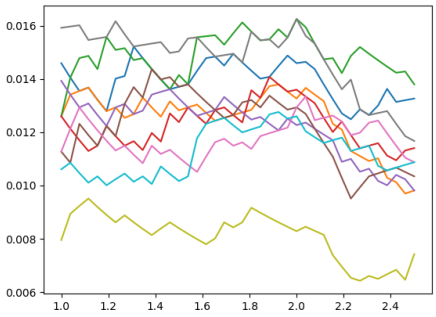


Figure 4: Various g(r) lines vs r)

## 2

Continuing with Lennard-Jones potential, now I started the particles in a lattice shape and very small initial velocities in a new 4x4 periodic box.
(a) In order to find the temperature, I decided to plot the velocity distribution and used the Maxwell-Boltzmann distribution to determine the temperature. In Figure 1 and with a normalization of about 0.5, I get that the temperature is about 0.15 in our units. (b) After equalibrating, the lattice still has structure, but the shapes are now triangles or diamonds instead of squares.
(c) In Figure 7, you can see now that the $\langle r^2(t) \rangle$ changes very little with time, suggesting that the particles form a solid shape.

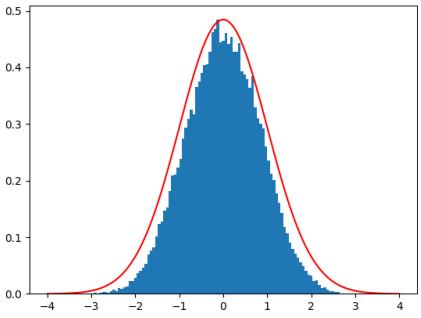


Figure 5: Velocity Distribution

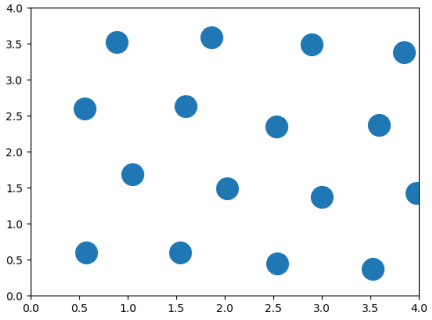


Figure 6: Equilibrium Positions

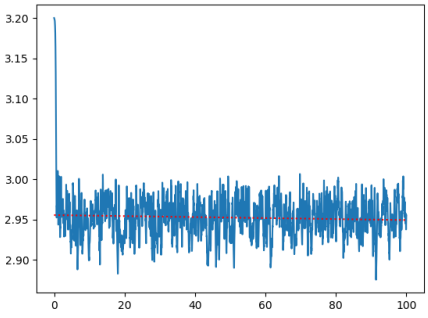


Figure 7: $\langle r^2(t) \rangle$ vs time for 100s

## 3

For this problem, to approximate increasing temperature, I increased the amount the particles would jump over time in each time step until a phase transition was visible.
(a) As I increased the increments as to how much the particles velocity would increase, the system would become unstable at a point, which is the large drop on Figure 8. Before the large drop however, the $\langle r^2(t) \rangle$ values is constant until 0.25. It is around this area I decide to take the velocity distribution as seen in Figure 9. The distribution does not seem to fit very well, but this could also be due to a lack of points. Assuming that the distribution is descriptive of the system, then the Maxwell-Boltzmann distribution tells me that the temperature of this system is 0.5 in our units.

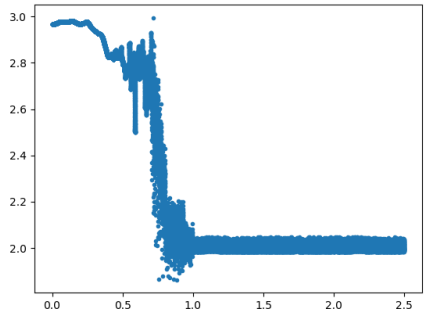


Figure 8: $\langle r^2(t) \rangle$ vs time as T increases

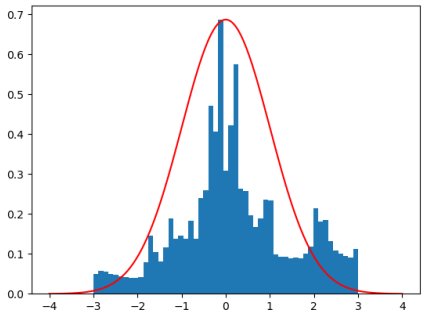


Figure 9: Velocity Distribution

# Appendix

## 1

```
### This is my Python 3.6 code ###
import numpy as np
import matplotlib.pylab as py
import copy as cp
from scipy.optimize import curve_fit

### Useful constatns
N = 16 #16 # number of particles
L = 10. # length of the box
m = 1. # mass of the particles
dt = 0.01 # time step or interval between steps
time = 100 # desired run time in seconds
window = 5 # how many time steps befor plot is updated
nts = int(time/dt) # how many discrete time steps will occur
dr = 2. # our dr for calculation density of particles
```

```
### Storing arrays
## Position arrays
x1 = np.zeros(N)
y1 = np.zeros(N)
x2 = np.zeros(N)
y2 = np.zeros(N)
x3 = np.zeros(N)
y3 = np.zeros(N)
## calculation arrays
radii = np.linspace(dr/2.,2.5,40)
g_r = np.zeros([10,len(radii)])
r_t = np.zeros(int(time/dt))
### Useful functions
def Force(dx,dy): # dx and dy distance from paricle i - particle i+1
    x = 12/((dx**2+dy**2)**7)*dx - 6/((dx**2+dy**2)**4)*dx
    y = 12/((dx**2+dy**2)**7)*dy - 6/((dx**2+dy**2)**4)*dy
    return [4.*x,4.*y]
def R(xt,yt,xold,yold,Fx,Fy): # Calculate the new positions
    xnew = 2*xt - xold + Fx/m*dt**2.
    ynew = 2*yt - yold + Fy/m*dt**2.
    return [xnew,ynew]
def PBC(px1,py1,px2,py2,length): # Calculate the nearest 'version' of other particles
    rtemp = 2.*length
    rmin = [1,1] # place holder
    for k in [-1.,0.,1.]:
        for l in [-1.,0.,1.]:
            xb = px2 + length * k
            yb = py2 + length * l
            x = px1 - xb
            y = py1 - yb
            if np.sqrt(x**2. + y**2.) < rtemp:
                rmin = [x,y] # error is taking place here and at return?
                rtemp = np.sqrt(x**2. + y**2.)
    return rmin
def chk(px3,py3):
    if px3 < 0: px3 = px3%L + L
    if px3 > L: px3 = px3%L
    if py3 < 0: py3 = py3%L + L
    if py3 > L: py3 = py3%L
    return px3,py3
def GR(xarr,yarr,r,dr,N): # for calculating density function after equilibrating
    g_r = []
    for j in range(N):
        count = 0
        for i in range(N):
```

```
            if i != j:
                temp = PBC(xarr[j],yarr[j],xarr[i],yarr[i],L)
                dist = np.sqrt(temp[0]**2. + temp[1]**2.)
                if (dist-dr/2. < r) and (dist+dr/2. >= r):
                    count += 1
        g_r += [1./(N-1.)*count/(2*np.pi*r*dr)]
    return sum(g_r)/N

#for r in radii:
### Set initial conditions
j = 1.
k = 0.
for i in range(N):
    k += 1
    if k == L:
        k -= L-1
        j += 1
    x1[i] = k ; y1[i] = j
for it in range(N): # generate random velocities in x direction
    v0 = np.random.uniform(low=-1.5,high=1.5)
    x2[it] = x1[it] + v0*dt
y2 = cp.deepcopy(y1)

v_t = []

py.figure()
counter = 0
for it in range(nts):
    if it%window == 0:
        py.clf()
        py.plot(x2,y2,'o',markersize=20) # Plot in loop to serve as animation
        py.xlim(0,L)
        py.ylim(0,L)
        py.pause(0.0001)
    rexp = []
    for t in range(N):
        Fxs = []
        Fys = []
        for be in range(0,t):
            temp = PBC(x2[t],y2[t],x2[be],y2[be],L) # calculate ranges for particle t
            dist = np.sqrt(temp[0]**2.+temp[1]**2.)
            if dist < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
```

```
        for af in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[af],y2[af],L) # calculate ranges for particle t
            dist = np.sqrt(temp[0]**2.+temp[1]**2.)
            if dist < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
        for RR in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[RR],y2[RR],L)
            rexp += [abs(temp[0]**2.+temp[1]**2.)]
        rexp += []
        Rtemp = R(x2[t],y2[t],x1[t],y1[t],sum(Fxs),sum(Fys)) # new positions stored
        # Check to see if positions need to be fixed
        x3[t] = Rtemp[0]
        y3[t] = Rtemp[1]
        x3[t], y3[t] = chk(x3[t],y3[t])
    r_t[it] = sum(rexp)/(len(rexp))

    # Shift information
    x1 = cp.deepcopy(x2)
    y1 = cp.deepcopy(y2)
    x2 = cp.deepcopy(x3)
    y2 = cp.deepcopy(y3)
    for t in range(N):
        dist1 = np.sqrt(x1[t]**2. + y1[t]**2.)
        dist2 = np.sqrt(x2[t]**2. + y2[t]**2.)
        v_t += [(dist2-dist1)/dt]
    if ((it+1)*dt)%10 < 0.00001:
        counter2 = 0
        for r in radii:
            g_r[counter,counter2] = GR(x3,y3,r,dr,N)
            counter2 += 1
        counter += 1



py.figure()
rang = np.linspace(0,time,nts)
py.plot(rang,r_t)
def func(t,D,c):
    return 4*D*t+c
par, con = curve_fit(func,rang,r_t)
py.plot(rang,func(rang,par[0],par[1]),'r:')
print('The diffusion constant for time 0<t<4 is', par[0])
```

```
X = np.linspace(-4,4,250)
py.figure()
y,x, _ = py.hist(v_t,bins='auto',range=(-3,3),normed=True)
py.plot(X,np.exp(-X**2./2.)*y.max(),'r')

py.figure()
for it in range(10):
    py.plot(radii,g_r[it,:])
py.show()
```

## 2

```
### This is my python 3.6 code ###
import numpy as np
import matplotlib.pylab as py
import copy as cp
from scipy.optimize import curve_fit
### Solving for hw3_2 lattice structure ###
### Useful constatns
N = 16 #16 # number of particles
L = 4. # length of the box
m = 1. # mass of the particles
dt = 0.005 # time step or interval between steps
time = 10. # desired run time in seconds
window = 15
nts = int(time/dt) # how many discrete time steps will occur
### Storing arrays
x1 = np.zeros(N)
y1 = np.zeros(N)
x2 = np.zeros(N)
y2 = np.zeros(N)
x3 = np.zeros(N)
y3 = np.zeros(N)
v_t = []
r_t = np.zeros(int(time/dt))
### Set initial conditions
j = 0.
k = 0.
for i in range(N):
    if k == L:
        k -= L
        j += 1
    x1[i] = k+0.5 ; y1[i] = j+0.5
    k += 1
for it in range(N):
```

```
    rand = np.random.randint(0,2)
    rand2 = np.random.choice([-1,1])
    if rand == 0:
        x2[it] = x1[it] + 0.0001* dt * rand2
        y2[it] = y1[it]
    if rand == 1:
        x2[it] = x1[it]
        y2[it] = y1[it] + 0.0001* dt * rand2

### Useful functions
def Force(dx,dy): # dx and dy distance from paricle i - particle i+1
    x = 12/((dx**2+dy**2)**7)*dx - 6/((dx**2+dy**2)**4)*dx
    y = 12/((dx**2+dy**2)**7)*dy - 6/((dx**2+dy**2)**4)*dy
    return [4.*x,4.*y]
def R(xt,yt,xold,yold,Fx,Fy): # Calculate the new positions
    xnew = 2*xt - xold + Fx/m*dt**2.
    ynew = 2*yt - yold + Fy/m*dt**2.
    return [xnew,ynew]
def PBC(px1,py1,px2,py2,length): # Calculate the nearest 'version' of other particles
    rtemp = 2.*length
    rmin = [1,1] # place holder
    for k in [-1.,0.,1.]:
        for l in [-1.,0.,1.]:
            xb = px2 + length * k
            yb = py2 + length * l
            x = px1 - xb
            y = py1 - yb
            if np.sqrt(x**2. + y**2.) < rtemp:
                rmin = [x,y] # error is taking place here and at return?
                rtemp = np.sqrt(x**2. + y**2.)
    return rmin
def chk(px3,py3):
    if px3 < 0: px3 = px3%L + L
    if px3 > L: px3 = px3%L
    if py3 < 0: py3 = py3%L + L
    if py3 > L: py3 = py3%L
    return px3,py3
### Carry out particle motion
py.figure()
for it in range(nts):
    if it%window == 0:
        py.clf()
        py.plot(x2,y2,'o',markersize=20) # Plot in loop to serve as animation
        py.xlim(0,L)
        py.ylim(0,L)
```

```
        py.pause(0.001)
    rexp = []
    for t in range(N):
        Fxs = []
        Fys = []
        for be in range(0,t):
            temp = PBC(x2[t],y2[t],x2[be],y2[be],L) # calculate ranges for particle t
            if np.sqrt(temp[0]**2.+temp[1]**2.) < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
        for af in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[af],y2[af],L) # calculate ranges for particle t
            if np.sqrt(temp[0]**2.+temp[1]**2.) < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
        for RR in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[RR],y2[RR],L)
            rexp += [abs(temp[0]**2.+temp[1]**2.)]
        rexp += []
        Rtemp = R(x2[t],y2[t],x1[t],y1[t],sum(Fxs),sum(Fys)) # new positions stored
        # Check to see if positions need to be fixed
        x3[t] = Rtemp[0]
        y3[t] = Rtemp[1]
        x3[t], y3[t] = chk(x3[t],y3[t])
    r_t[it] = sum(rexp)/(len(rexp))
    # Shift information
    x1 = cp.deepcopy(x2)
    y1 = cp.deepcopy(y2)
    x2 = cp.deepcopy(x3)
    y2 = cp.deepcopy(y3)
    if it*dt > 2:
        for t in range(N):
            dist1 = np.sqrt(x1[t]**2. + y1[t]**2.)
            dist2 = np.sqrt(x2[t]**2. + y2[t]**2.)
            v_t += [(dist2-dist1)/dt]

py.figure()
rang = np.linspace(0,time,nts)
py.plot(rang,r_t)
def func(t,D,c):
    return 4*D*t+c
par, con = curve_fit(func,rang,r_t)
py.plot(rang,func(rang,par[0],par[1]),'r:')
```

```
#print('The diffusion constant for time 0<t<4 is', par[0])

X = np.linspace(-4,4,250)
py.figure()
y,x, _ = py.hist(v_t,bins='auto',range=(-3,3),normed=True)
py.plot(X,np.exp(-X**2./2.)*y.max(),'r')
#py.show()

print(x1,'\n',y1,'\n',x2,'\n',y2)
```

## 3

```
### This is my Python 3.6 code ###
import numpy as np
import matplotlib.pylab as py
import copy as cp

### Useful constatns
N = 16 #16 # number of particles
L = 4. # length of the box
m = 1. # mass of the particles
dt = 0.0001 # time step or interval between steps
time = 2.5 # desired run time in seconds
window = 25
nts = int(time/dt) # how many discrete time steps will occur
### Storing arrays
x3 = np.zeros(N)
y3 = np.zeros(N)
r_t = np.zeros(int(time/dt))
v_t1 = []
### Set initial conditions from previous problem
x1 = np.array([0.12692186,1.14655152,2.12135533,3.13075539,0.71435339,1.7078084,
               2.76639917,3.73238345,0.24519842,1.22906841,2.21510929,3.25110702,
               0.92330001,1.90003208,2.83814643,3.81833175])
y1 = np.array([0.43337238,0.66226779,0.48126928,0.44449269,1.58228456,1.49729541,
               1.48091992,1.45707639,2.52132384,2.47500839,2.40250353,2.44809026,
               3.56358399,3.44055439,3.39514417,3.44174923])
"""x2 = np.array([0.12813631,1.14474598,2.12813627,3.13452539,0.7121143,1.70428552,
               2.77176138,3.72894095,0.24511683,1.22839879,2.21769505,3.24973686,
               0.91858518,1.89690116,2.83374492,3.82396372])
y2 = np.array([0.43685195,0.65948376,0.48362289,0.44397257,1.5840192,1.49571746,
               1.48045777,1.44377887,2.51956455,2.46973809,2.4122407,2.44968079,
               3.57012440,3.43870932,3.39977178,3.43913385])"""
x2 = cp.deepcopy(x1)
y2 = cp.deepcopy(y1)
```

```
### Useful functions
def Force(dx,dy): # dx and dy distance from paricle i - particle i+1
    x = 12/((dx**2+dy**2)**7)*dx - 6/((dx**2+dy**2)**4)*dx
    y = 12/((dx**2+dy**2)**7)*dy - 6/((dx**2+dy**2)**4)*dy
    return [4.*x,4.*y]
def R(xt,yt,xold,yold,Fx,Fy,T_inc): # Calculate the new positions
    x = 2*xt - xold + Fx/m*dt**2.
    y = 2*yt - yold + Fy/m*dt**2.
    xnew = (x-xt)*(T_inc/100 + 1) + xt
    ynew = (y-yt)*(T_inc/100 + 1) + yt
    return [xnew,ynew]
def PBC(px1,py1,px2,py2,length): # Calculate the nearest 'version' of other particles
    rtemp = 2.*length
    rmin = [1,1] # place holder
    for k in [-1.,0.,1.]:
        for l in [-1.,0.,1.]:
            xb = px2 + length * k
            yb = py2 + length * l
            x = px1 - xb
            y = py1 - yb
            if np.sqrt(x**2. + y**2.) < rtemp:
                rmin = [x,y] # error is taking place here and at return?
                rtemp = np.sqrt(x**2. + y**2.)
    return rmin
def chk(px3,py3):
    if px3 < 0: px3 = px3%L + L
    if px3 > L: px3 = px3%L
    if py3 < 0: py3 = py3%L + L
    if py3 > L: py3 = py3%L
    return px3,py3
### Carry out particle motion
py.figure()
T = 0.
for it in range(nts):
    if it%window == 0:
        py.clf()
        py.plot(x2,y2,'o',markersize=20) # Plot in loop to serve as animation
        py.xlim(0,L)
        py.ylim(0,L)
        py.pause(0.001)
    if (it+1)%1000 == 0:
        T += 0.05
        print(T, it*dt)
    rexp = []
    for t in range(N):
```

```
        Fxs = []
        Fys = []
        for be in range(0,t):
            temp = PBC(x2[t],y2[t],x2[be],y2[be],L) # calculate ranges for particle t
            if np.sqrt(temp[0]**2.+temp[1]**2.) < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
        for af in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[af],y2[af],L) # calculate ranges for particle t
            if np.sqrt(temp[0]**2.+temp[1]**2.) < 3.1:
                temp2 = Force(temp[0],temp[1]) # calculate forces for these ranges
                Fxs += [temp2[0]]
                Fys += [temp2[1]]
        for RR in range(t+1,N):
            temp = PBC(x2[t],y2[t],x2[RR],y2[RR],L)
            rexp += [abs(temp[0]**2.+temp[1]**2.)]
        rexp += []
        Rtemp = R(x2[t],y2[t],x1[t],y1[t],sum(Fxs),sum(Fys),T) # new positions
        # Check to see if positions need to be fixed
        x3[t] = Rtemp[0]
        y3[t] = Rtemp[1]
        x3[t], y3[t] = chk(x3[t],y3[t])
    r_t[it] = sum(rexp)/(len(rexp))
    # Shift information
    x1 = cp.deepcopy(x2)
    y1 = cp.deepcopy(y2)
    x2 = cp.deepcopy(x3)
    y2 = cp.deepcopy(y3)
    if it*dt > 0.25 and it*dt < 0.4:
        for t in range(N):
            dist1 = np.sqrt(x1[t]**2. + y1[t]**2.)
            dist2 = np.sqrt(x2[t]**2. + y2[t]**2.)
            v_t1 += [(dist2-dist1)/dt]

py.figure()
rang = np.linspace(0,time,nts)
py.plot(rang,r_t,'.')

X = np.linspace(-3,3,250)
py.figure()
y,x, _ = py.hist(v_t1,bins='auto',range=(-3,3),normed=True)
py.plot(X,np.exp(-X**2./2.)*y.max(),'r')
py.figure()
y,x, _ = py.hist(v_t2,bins='auto',range=(-3,3),normed=True)
```

```
py.plot(X,np.exp(-X**2./2.)*y.max(),'r')
py.show()
```