PHYS2600: Homework 6*

Due on Tuesday, April 10, 2018

Yangrui Hu

*This Latex template is from http://www.latextemplates.com

Contents

Problem 1

Problem: Calculate speed distributions for particles in a Molecular dynamics simulation considering how, and if, certain peculiar states approach equilibrium. Consider a dilute gas in a 16 $\times$ 16 box containing 25 particles. Give all of the particles an initial v_x which is positive; you might, for example, choose v_x randomly in the range 0 to 1. Show that the probability distribution P(v_x) never approaches the Maxwell-Boltzmann form, as the average value of v_x will always be positive rather than zero. Be sure not use the zeroTotalMomentum() method when initializing velocities and at each time step. Explain why the result you find follows from the conservation of momentum.

Solution: Set the initial velocities in x direction randomly in the range 0 to 1 and plot the histogram of v_x, $|v_x|$, v_y, and v as Figure 3, Figure 2, Figure 4, and Figure 1. The code is shown as Listing 1. The temperature of the system is 100. From these figures we can clearly see that the probability distribution of v_x is not Maxwell-Boltzmann form. And form Figure 3, we can find that the mean of v_x is larger than zero, because the initial v_x are positive and the total momentum in x direction is conserved. The total momentum in x direction is $\sum_i m_i v_{x,i}$ and in this problem we set $m_i = 1$, so $\sum_i v_{x,i}$ is always positive. The reason of the conservation of momentum is that we use periodic boundary condition which means there is no external force acts on this system. According to Newton's second law, the total momentum of the system will be conserved without external force.

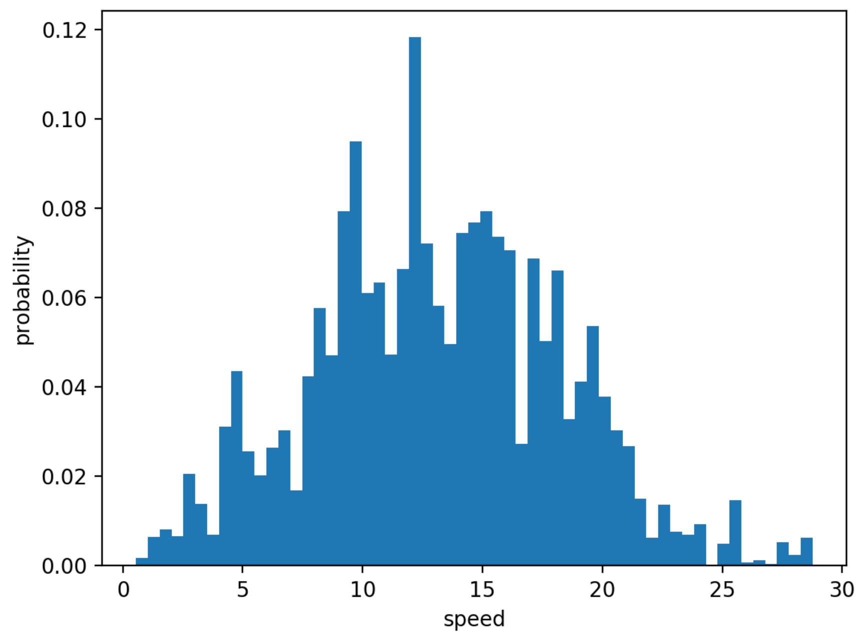

Figure 1: The speed distribution for particles

Problem 1 continued on next page...

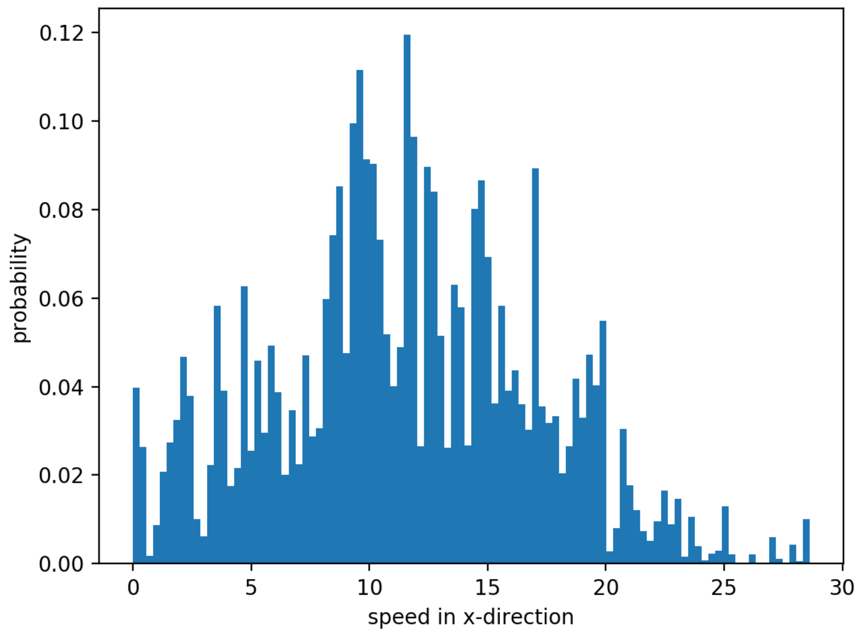

Figure 2: The x-direction speed distribution for particles

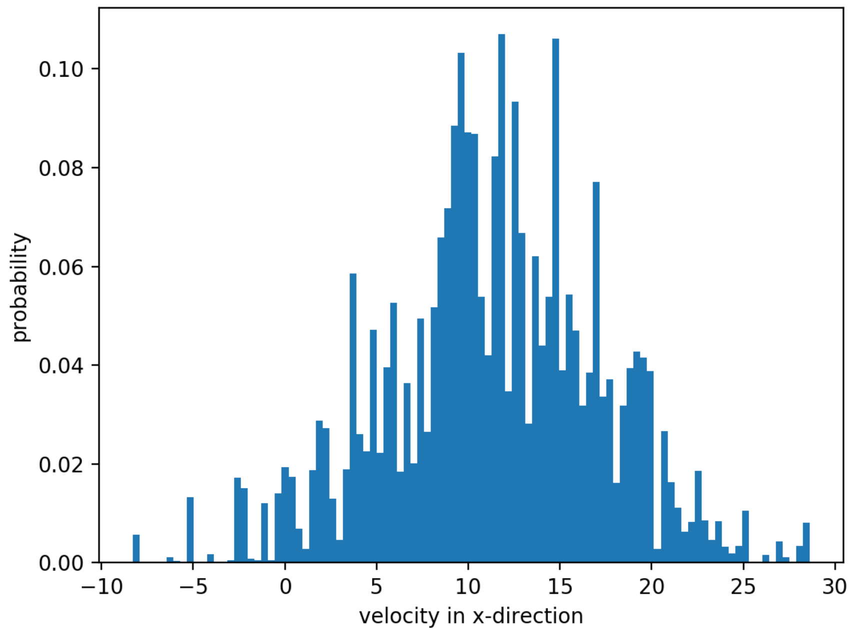

Figure 3: The v_x distribution for particles

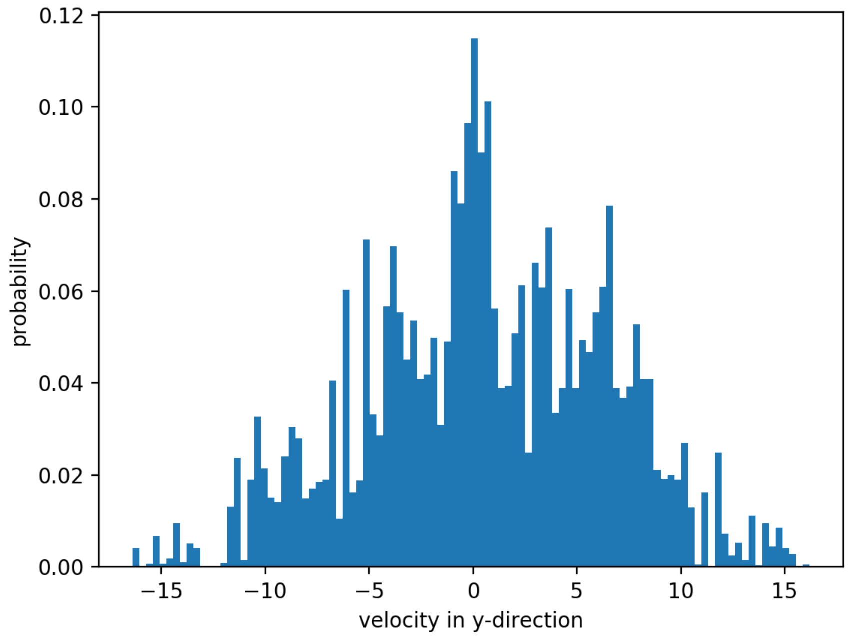

Figure 4: The v_y distribution for particles

Listing 1: Original Script of Problem 1

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as am
from matplotlib import colors, cm
from numba import jit

@jit(nopython=True)
def jit_lennardJonesForce(N, L, x, y):

    f = np.zeros((2*N), np.double)
    virial = 0
    tiny = 1.0e-40
    halfL = L/2

    cutoff = L/3

    for ii  in range(N):
        for jj in range(ii+1, N):
            dx = x[ii] - x[jj]
            if( dx > halfL): dx = dx - L
            if( dx < -halfL): dx = dx + L

            dy = y[ii] - y[jj]
            if( dy > halfL): dy = dy - L
            if( dy < -halfL): dy = dy + L
```

Problem 1 continued on next page...

```
                if dx**2+dy**2 > cutoff:
                    continue

                r2inv = 1.0/(dx*dx + dy*dy + tiny)
                c = 48.0 * r2inv**7 - 24.0 * r2inv**4
                fx = dx * c
                fy = dy * c

                f[2*ii] += fx;
                f[2*ii+1] += fy;
                f[2*jj] -= fx; # Newton's 3rd law
                f[2*jj+1] -= fy;

                virial += fx*dx + fy*dy;

    return f, virial

class MolecularDynamics:

    sampleInterval = 100

    def __init__(self, N=4, L=10.0, initialTemperature=0.0, dt = 0.001):

        self.N = N  # number of particles
        self.L = L      # length of square side
        self.initialTemperature = initialTemperature

        self.t = 0.0 # initial time
        self.dt = dt

        self.x = np.zeros(2*N) # NumPy array of N (x, y) positions
        self.v = np.zeros(2*N) # array of N (vx, vy) velocities
        self.vArray = np.array([])


    def force(self):

        f, virial = self.lennardJonesForce_jit()
        return f

    def lennardJonesForce_jit(self):
        N = self.N
        L = self.L

        x = self.x[::2]
        y = self.x[1::2]
        f, virial = jit_lennardJonesForce(N, L, x, y)

        return f, 0.5*virial

    def verletStep(self):

        a = self.force()
        self.x += self.v * self.dt + 0.5 * self.dt**2 * a
        self.x = self.x % self.L
        self.v += 0.5 * self.dt * (a + self.force())

    def evolve(self, time=10.0):

        steps = int(abs(time/self.dt))

        for i in range(steps):
            self.verletStep()
            self.t += self.dt
            if (i % self.sampleInterval == 0):
                self.vArray =  np.append(self.vArray, self.v)

    def rectangularLatticePositions(self):

        if np.abs(np.sqrt(self.N) - np.floor(np.sqrt(self.N))) > 1e-10:
            N_new = np.rint(np.sqrt(self.N))**2
            self.N = int(N_new)
            self.x = np.zeros(2*self.N) # NumPy array of N (x, y) positions
            self.v = np.zeros(2*self.N) # array of N (vx, vy) velocities

            print("N must be a square integer to use this method\n"+
                  "Setting N to {} and reseting x and v".format(N_new))

        nx = int(np.sqrt(self.N))
        ny = nx
        dx = self.L / nx
        dy = self.L / ny

        for i in range(nx):
            x = (i + 0.5) * dx
            for j in range(ny):
                y = (j + 0.5) * dy
                self.x[2*(i*ny+j)] = x
                self.x[2*(i*ny+j)+1] = y


    def randomVelocities(self):
        self.v = np.random.random(2*self.N) - 0.5
        self.v[::2] = self.v[::2] + 0.5
        T = self.temperature()
        self.v *= np.sqrt(self.initialTemperature/T)

    def kineticEnergy(self):

        return 0.5 * (self.v * self.v).sum()


    def temperature(self):

        return self.kineticEnergy() / self.N
```

Problem 1 continued on next page. . .

```

    def velocityHistogram(self):
        vx = self.vArray[::2]
        vy = self.vArray[1::2]

        plt.figure()
        plt.hist(vx, bins= 100, normed=1)
        plt.xlabel("velocity in x-direction")
        plt.ylabel("probability")

        plt.figure()
        plt.hist(np.abs(vx), bins= 100, normed=1)
        plt.xlabel("speed in x-direction")
        plt.ylabel("probability")

        plt.figure()
        plt.hist(vy, bins= 100, normed=1)
        plt.xlabel("velocity in y-direction")
        plt.ylabel("probability")

        v = np.sqrt(vx**2+vy**2)
        plt.figure()
        plt.hist(v, bins= 'auto', normed=1)
        plt.xlabel("speed")
        plt.ylabel("probability")

    def showPlots(self):
        plt.show()


def ideal_gas(Temp = 100):
    gas = MolecularDynamics(N=25, L=16, initialTemperature = Temp, dt = 0.0001)
    gas.rectangularLatticePositions()
    gas.randomVelocities()
    gas.evolve(time=10.0)
    gas.velocityHistogram()
    gas.showPlots()
    return gas

ideal_gas(Temp = 100)
```

Problem 2

Problem: The diffusion of an atom in a system containing a large number of atoms is known as self-diffusion. Study the diffusion of particles in a dilute system. For example, take 16 particles in a 16 $\times$ 16 box and calculate the mean-square displacement of a single particle as a function of time. Show that the motion is indeed diffusive, i.e. $(\Delta r)^2 \approx Dt$, and find the value of the diffusion constant. You should also study how D varies with density. Be sure to properly allow for the teleportation that occurs because of periodic boundary conditions when you calculate Δr.

Solution: Set the number of particles $N = 1$, the length of the box $L = 16$. We focus on the trajectory of one particle and calculate the mean-square displacement $(\Delta r)^2$. The left figure in Figure 5 shows the relationship between $(\Delta r)^2$ and time. This relation is almost linear and the slope is $D = 338.18$. So the motion is diffusive as $(\Delta r)^2 \approx Dt$. The initial condition of this simulation is random, so just one simulation cannot get the good result. Like doing an experiment, we should do MD simulation many times and calculate the average value of D. The right figure of Figure 5 shows the distribution of D when doing MD simulation 100 times. This distribution should be Maxwell-Boltzmann distribution for the random initial conditions which is consistent with my simulation results.
When I try to study the relationship between diffusion constant and density, I keep the number of particles as $N = 16$ and change the length of the box. The density of particle is $\rho = N/L^2$. Figure 6 shows how D varies with density. From Figure 6, we can state that the diffusion constant will decrease as density increasing. It is not difficult to understand: imagine you are walking on the street, if there are too many people, you can not walk fast.

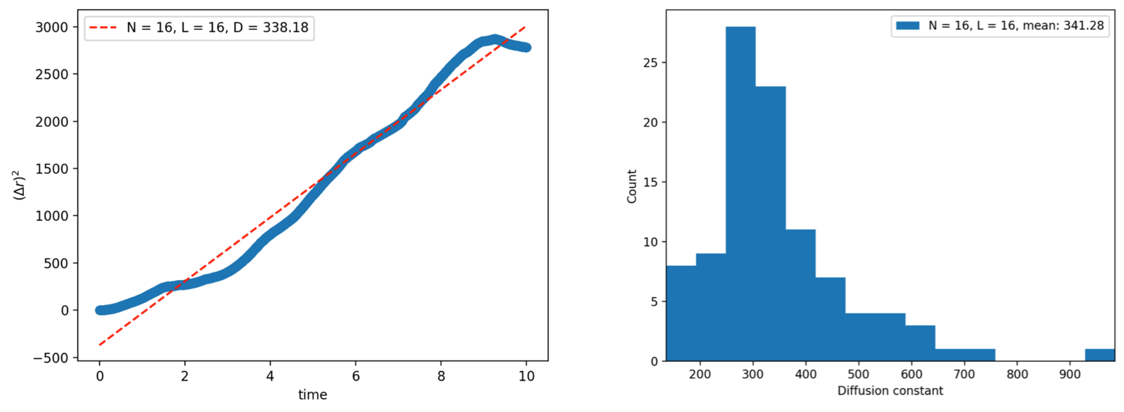

Figure 5: Left: the evolution of the mean-square displacement $(\Delta r)^2$ for one particle; Right: the distribution of the diffusion constant under 100 simulations

Problem 2 continued on next page. . .

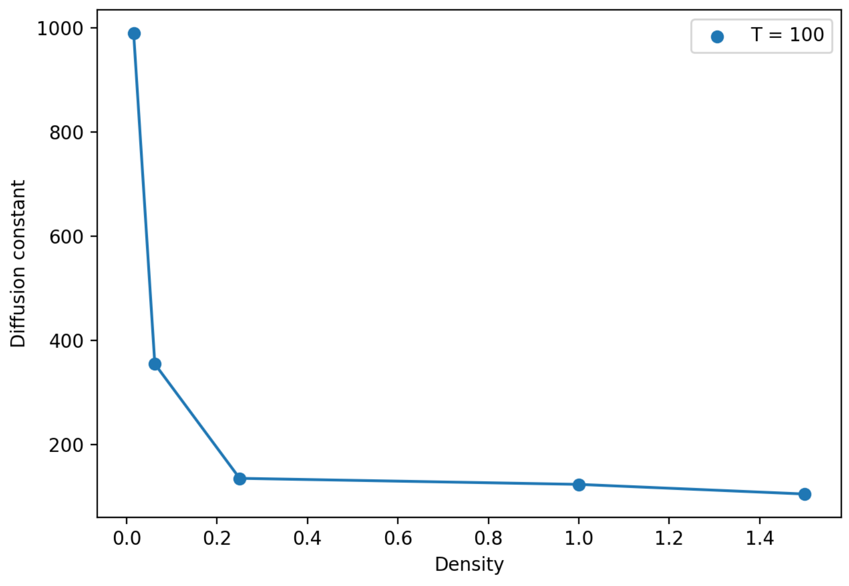

Figure 6: The relationship between diffusion constant and density

Listing 2: Original Script of Problem 2

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as am
from matplotlib import colors, cm
from numba import jit

@jit(nopython=True)
def jit_lennardJonesForce(N, L, x, y):

     f = np.zeros((2*N), np.double)
     virial = 0
     tiny = 1.0e-40
     halfL = L/2

     cutoff = L/3

     for ii  in range(N):
          for jj in range(ii+1, N):
               dx = x[ii] - x[jj]
               if( dx > halfL): dx = dx - L
               if( dx < -halfL): dx = dx + L

               dy = y[ii] - y[jj]
               if( dy > halfL): dy = dy - L
               if( dy < -halfL): dy = dy + L

               if dx**2+dy**2 > cutoff:
                    continue
```

Problem 2 continued on next page...

```
                r2inv = 1.0/(dx*dx + dy*dy + tiny)
                c = 48.0 * r2inv**7 - 24.0 * r2inv**4
                fx = dx * c
                fy = dy * c

                f[2*ii] += fx;
                f[2*ii+1] += fy;
                f[2*jj] -= fx; # Newton's 3rd law
                f[2*jj+1] -= fy;

                virial += fx*dx + fy*dy;

     return f, virial

class MD(object):

     def __init__(self, N=16, L=16, dt = 0.001, T = 100):

          self.N = N
          self.L = L
          self.t = 0.0
          self.dt = dt
          self.T = T

          self.tarray = np.array([])

          self.x = np.zeros(2*N)
          self.x_unwrap = np.zeros(2*N)
          self.x_initial = np.zeros(2*self.N)
          self.v = np.zeros(2*N)

     def force(self):

          N = self.N
          L = self.L

          x = self.x[::2]
          y = self.x[1::2]
          f, virial = jit_lennardJonesForce(N, L, x, y)

          return f

     def verletStep(self):

          a = self.force()
          self.x_unwrap += self.v * self.dt + 0.5 * self.dt**2 * a
          self.x += self.v * self.dt + 0.5 * self.dt**2 * a

          self.x = self.x % self.L
          self.v += 0.5 * self.dt * (a + self.force())

     def evolve(self, time=10.0):
          #reset x,v,t
          self.x = np.zeros(2*self.N)
          self.x_unwrap = np.zeros(2*self.N)
          self.x_initial = np.zeros(2*self.N)
          self.v = np.zeros(2*self.N)
          self.v = np.random.random(2*self.N) - 0.5
          self.rectangularLatticePositions()
          self.t = 0.0

          steps = int(abs(time/self.dt))
          current_T = self.temperature()
          self.v *= np.sqrt(self.T/current_T)

          r_square = []
          tarray = []

          for j in range(steps):
               self.verletStep()
               self.t += self.dt
               delta_r_square = 0
               for k in range(self.N):
                    xc = self.x_unwrap[2*k]-self.x_initial[2*k]
                    yc = self.x_unwrap[2*k+1]-self.x_initial[2*k+1]
                    delta_r_square += xc**2 + yc**2
               delta_r_square = delta_r_square/self.N
               r_square.append(delta_r_square)
               tarray.append(self.t)

          return tarray, r_square

     def temperature(self):
          return self.kineticEnergy() / self.N

     def kineticEnergy(self):
          return 0.5 * (self.v * self.v).sum()

     def rectangularLatticePositions(self):

          nx = int(np.sqrt(self.N))
          ny = nx
          dx = self.L / nx
          dy = self.L / ny

          for i in range(nx):
               x = (i + 0.5) * dx
               for j in range(ny):
                    y = (j + 0.5) * dy
                    self.x[2*(i*ny+j)] = x
                    self.x[2*(i*ny+j)+1] = y
                    self.x_unwrap[2*(i*ny+j)] = x
                    self.x_unwrap[2*(i*ny+j)+1] = y
                    self.x_initial[2*(i*ny+j)] = x
```

Problem 2 continued on next page. . .

```
                    self.x_initial[2*(i*ny+j)+1] = y


def func(x, a, b):
     return a*x+b

def onetrial():
     gas = MD(N = 16, L = 16, T = 100)
     tarray, r_square = gas.evolve()
     fit = np.polyfit(tarray, r_square, 1)
     y2 = [func(i, fit[0], fit[1]) for i in tarray]
     fig = plt.figure()
     ax = fig.add_subplot(111)
     ax.scatter(tarray, r_square)
     ax.plot(tarray, y2,'r--',label='N = 16, L = 16, D = %.2f'%fit[0])
     ax.set_xlabel("time")
     ax.set_ylabel("$(\Delta r)^2$")
     ax.legend()
     plt.show()

def getD():
     gas = MD(N = 16, L = 16, T = 100)
     with open("problem2_data.txt","w") as file:
          for i in range(100):
               tarray, r_square = gas.evolve()
               fit = np.polyfit(tarray, r_square, 1)
               file.write(str(fit[0]) + "\n")



def DvsDensity():
     N = 16
     with open("problem2_density_data.txt","w") as file:
          for i in range(4):
               rho = 0.5 + i * 0.5
               L = np.sqrt(N/rho)
               gas = MD(N = N, L = L, T = 100)
               D = []
               for n in range(20):
                    tarray, r_square = gas.evolve()
                    fit = np.polyfit(tarray, r_square, 1)
                    D.append(fit[0])
               file.write(str(rho)+ " " + str(np.average(D)) + "\n")


def main():
     #onetrial()
     #getD()
     DvsDensity()

if __name__ == '__main__':
     main()
```

Problem 3

Problem: The transition between liquid and solid phases can be roughly tracked by monitoring the relative separation $(\Delta r)^2$ between two atoms. In a solid, this separation should remain approximately constant with time, while in a liquid $(\Delta r)^2$ will grow as the atoms undergo diffusion. For a system of particles interacting with a Lennard-Jones potential, study how the melting transition depends on density. Begin with the system initialized in a solid phase and slowly heat up by rescaling the velocities. Roughly determine the melting temperature by monitoring $(\Delta r)^2$ for a pair of atoms. You should find that the melting temperature drops as the density is reduced.

Solution: Set the number of particles $N = 1$ and the length of the box $L = 3,\ 4,\ 5,\ 6$. Now we focus on how the distance between two particles $(\Delta r)^2$ evolves when slowly heat up these systems. Figure 7 shows the evolution of $(\Delta r)^2$ in systems with different density. From each figure we can find that $(\Delta r)^2$ is almost unchanged at first, and when the system approaches to a critical temperature which is called the melting temperature, $(\Delta r)^2$ increases dramatically. Figure 8 shows how the melting temperature changes with density and we can see that the melting temperature drops as the density is reduced.

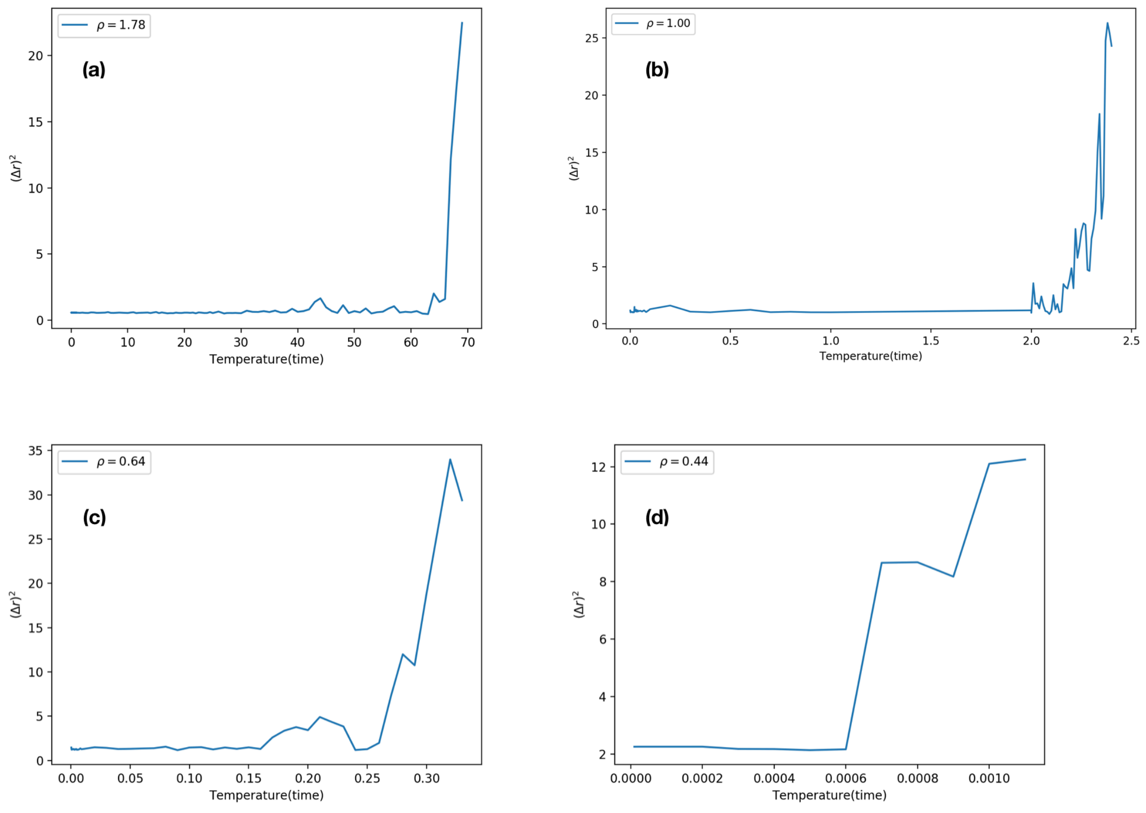

Figure 7: The evolution of $(\Delta r)^2$ with heating up the system continuously: (a): $L = 3$, (b): $L = 4$, (c): $L = 5$, (d): $L = 6$

Problem 3 continued on next page...

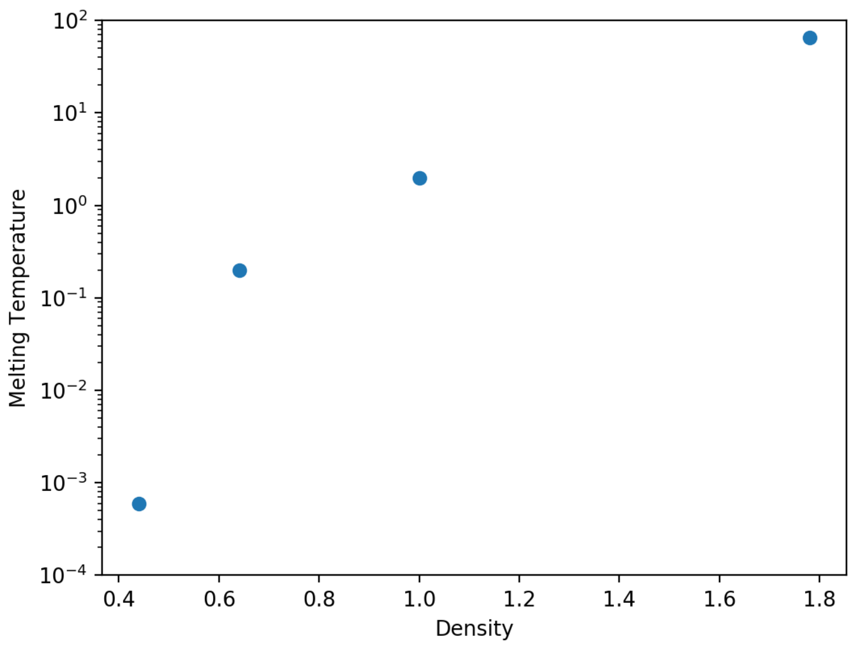

Figure 8: The relationship between the melting temperature and the density

Listing 3: Original Script of Problem 3

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as am
from matplotlib import colors, cm
from numba import jit

@jit(nopython=True)
def jit_lennardJonesForce(N, L, x, y):

     f = np.zeros((2*N), np.double)
     virial = 0
     tiny = 1.0e-40
     halfL = L/2

     cutoff = L/3

     for ii  in range(N):
          for jj in range(ii+1, N):
               dx = x[ii] - x[jj]
               if( dx > halfL): dx = dx - L
               if( dx < -halfL): dx = dx + L

               dy = y[ii] - y[jj]
               if( dy > halfL): dy = dy - L
               if( dy < -halfL): dy = dy + L
```

```
                if dx**2+dy**2 > cutoff:
                    continue

                r2inv = 1.0/(dx*dx + dy*dy + tiny)
                c = 48.0 * r2inv**7 - 24.0 * r2inv**4
                fx = dx * c
                fy = dy * c

                f[2*ii] += fx;
                f[2*ii+1] += fy;
                f[2*jj] -= fx; # Newton's 3rd law
                f[2*jj+1] -= fy;

                virial += fx*dx + fy*dy;

    return f, virial

class MD(object):

    def __init__(self, N=16, L=16, dt = 0.001):

        self.N = N
        self.L = L
        self.t = 0.0
        self.dt = dt

        self.tarray = np.array([])

        self.x = np.zeros(2*N)
        self.x_unwrap = np.zeros(2*N)
        self.v = np.zeros(2*N)

    def force(self):

        N = self.N
        L = self.L

        x = self.x[::2]
        y = self.x[1::2]
        f, virial = jit_lennardJonesForce(N, L, x, y)

        return f


    def verletStep(self):

        a = self.force()
        self.x_unwrap += self.v * self.dt + 0.5 * self.dt**2 * a
        self.x += self.v * self.dt + 0.5 * self.dt**2 * a

        self.x = self.x % self.L
        self.v += 0.5 * self.dt * (a + self.force())
   def evolve(self, time=5.0):

        with open("problem3_L6.txt","w") as file:

             T_int = 0.0001
             T_final = 0.003
             dT = 0.0001
             T_step = int((T_final-T_int)/dT)+1
             Temp_array = np.linspace(T_int, T_final, T_step, endpoint=True)
             distance_array = []
             rho = self.N/(self.L)**2

             self.v = np.random.random(2*self.N) - 0.5
             self.rectangularLatticePositions()

             for i in Temp_array:

                  steps = int(abs(time/self.dt))
                  current_T = self.temperature()
                  self.v *= np.sqrt(i/current_T)

                  for j in range(steps):
                       self.verletStep()
                       self.t += self.dt
                       r = (self.x_unwrap[0]-self.x_unwrap[2])**2 +
                       (self.x_unwrap[1]-self.x_unwrap[3])**2

                  distance_array.append(r)
                  file.write(str(i)+ " " + str(r) + "\n")

        plt.plot(Temp_array, distance_array, label='$\\rho = %.2f$'%rho)
        plt.xlabel("Temperature(time)")
        plt.ylabel('$(\Delta r)^2$')
        plt.legend()
        plt.show()
        plt.close()

   def temperature(self):
        return self.kineticEnergy() / self.N

   def kineticEnergy(self):
        return 0.5 * (self.v * self.v).sum()

   def rectangularLatticePositions(self):

        nx = int(np.sqrt(self.N))
        ny = nx
        dx = self.L / nx
        dy = self.L / ny

        for i in range(nx):
             x = (i + 0.5) * dx
```

Problem 3 continued on next page...

```
                for j in range(ny):
                    y = (j + 0.5) * dy
                    self.x[2*(i*ny+j)] = x
                    self.x[2*(i*ny+j)+1] = y
                    self.x_unwrap[2*(i*ny+j)] = x
                    self.x_unwrap[2*(i*ny+j)+1] = y



gas = MD(N = 16, L = 6)
gas.evolve()
```

Problem 4

Problem: Consider the process of self-diffusion discussed in problem 2. The diffusion constant can be used to study melting and, in particular, to locate the melting transition. Calculate the diffusion constant for self-diffusion for a system of 16 particles in a 4 × 4 box as a function of temperature. Compare its value in the solid and liquid phases. Can the diffusion constant be used to determine when the system melts? To improve statistical accuracy, average the diffusion constants of all of the particles. Be sure to account for the periodic boundary conditions.

Solution: Set the number of particle $N = 16$ and the length of the box $L = 4$. Calculate the diffusion constant as problem 2 did in systems with different temperature. Figure 9 shows the relationship between the diffusion constant D and the temperature and we can find that D is almost unchanged at first, and when the system approaches to a critical temperature, D increases dramatically. The critical temperature is roughly at $T = 2$, which is consistent with the result got from problem 3: Figure 7 (b) is for $L = 4$ case and the melting temperature is roughly $T = 2$. So the diffusion constant can be used to determine when the system melts.

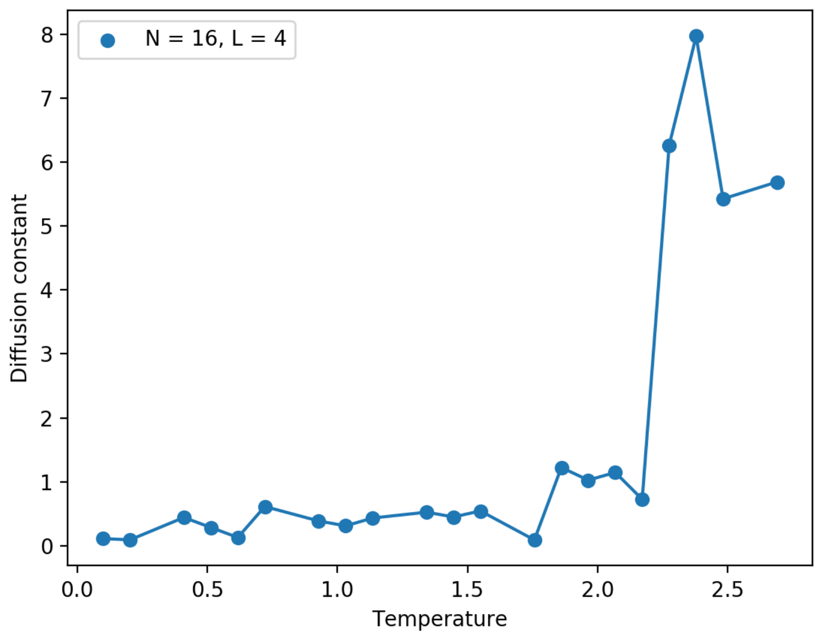

Figure 9: The relationship between the diffusion constant and the temperature

Problem 4 continued on next page...

Listing 4: Original Script of Problem 4

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as am
from matplotlib import colors, cm
from numba import jit

@jit(nopython=True)
def jit_lennardJonesForce(N, L, x, y):

     f = np.zeros((2*N), np.double)
     virial = 0
     tiny = 1.0e-40
     halfL = L/2

     cutoff = L/3

     for ii  in range(N):
          for jj in range(ii+1, N):
               dx = x[ii] - x[jj]
               if( dx > halfL): dx = dx - L
               if( dx < -halfL): dx = dx + L

               dy = y[ii] - y[jj]
               if( dy > halfL): dy = dy - L
               if( dy < -halfL): dy = dy + L

               if dx**2+dy**2 > cutoff:
                    continue

               r2inv = 1.0/(dx*dx + dy*dy + tiny)
               c = 48.0 * r2inv**7 - 24.0 * r2inv**4
               fx = dx * c
               fy = dy * c

               f[2*ii] += fx;
               f[2*ii+1] += fy;
               f[2*jj] -= fx; # Newton's 3rd law
               f[2*jj+1] -= fy;

               virial += fx*dx + fy*dy;

     return f, virial

class MD(object):

     def __init__(self, N=16, L=16, dt = 0.001):

          self.N = N
          self.L = L
          self.t = 0.0
          self.dt = dt
```

Problem 4 continued on next page. . .

```
        self.tarray = np.array([])

        self.x = np.zeros(2*N)
        self.x_unwrap = np.zeros(2*N)
        self.x_initial = np.zeros(2*self.N)
        self.v = np.zeros(2*N)

    def force(self):

        N = self.N
        L = self.L

        x = self.x[::2]
        y = self.x[1::2]
        f, virial = jit_lennardJonesForce(N, L, x, y)

        return f

    def verletStep(self):

        a = self.force()
        self.x_unwrap += self.v * self.dt + 0.5 * self.dt**2 * a
        self.x += self.v * self.dt + 0.5 * self.dt**2 * a

        self.x = self.x % self.L
        self.v += 0.5 * self.dt * (a + self.force())


    def evolve(self, time=10.0):

        T_int = 0.1
        T_final = 3
        dT = 0.1
        T_step = int((T_final-T_int)/dT)+1
        Temp_array = np.linspace(T_int, T_final, T_step, endpoint=True)
        D = []

        for i in Temp_array:
            #reset x,v,t
            self.x = np.zeros(2*self.N)
            self.x_unwrap = np.zeros(2*self.N)
            self.x_initial = np.zeros(2*self.N)
            self.v = np.zeros(2*self.N)
            self.v = np.random.random(2*self.N) - 0.5
            self.rectangularLatticePositions()
            self.t = 0.0

            steps = int(abs(time/self.dt))
            current_T = self.temperature()
            self.v *= np.sqrt(i/current_T)

            r_square = []
            tarray = []
```

Problem 4 continued on next page. . .

```
                file = open('problem4_data_T%.2f.txt'%i, 'w')
                for j in range(steps):
                    self.verletStep()
                    self.t += self.dt
                    delta_r_square = 0
                    for k in range(self.N):
                        xc = self.x_unwrap[2*k]-self.x_initial[2*k]
                        yc = self.x_unwrap[2*k+1]-self.x_initial[2*k+1]
                        delta_r_square += xc**2 + yc**2
                    delta_r_square = delta_r_square/self.N
                    r_square.append(delta_r_square)
                    tarray.append(self.t)
                    file.write(str(self.t)+ " " + str(delta_r_square) + "\n")

                fit = np.polyfit(tarray, r_square, 1)
                D.append(fit[0])

            return Temp_array, D

    def temperature(self):
        return self.kineticEnergy() / self.N

    def kineticEnergy(self):
        return 0.5 * (self.v * self.v).sum()

    def rectangularLatticePositions(self):

        nx = int(np.sqrt(self.N))
        ny = nx
        dx = self.L / nx
        dy = self.L / ny

        for i in range(nx):
            x = (i + 0.5) * dx
            for j in range(ny):
                y = (j + 0.5) * dy
                self.x[2*(i*ny+j)] = x
                self.x[2*(i*ny+j)+1] = y
                self.x_unwrap[2*(i*ny+j)] = x
                self.x_unwrap[2*(i*ny+j)+1] = y
                self.x_initial[2*(i*ny+j)] = x
                self.x_initial[2*(i*ny+j)+1] = y


gas = MD(N = 16, L = 4)
Temp_array, D = gas.evolve()
a = len(Temp_array)
with open("problem4_TvsD.txt","w") as file:
    for i in range(a):
        file.write(str(Temp_array[i])+ " " + str(D[i]) + "\n")
```

Appendix: Code for plotting figures in this homework

Listing 5: Original Script for plotting figures

```python
import matplotlib.pyplot as plt
import numpy as np

def plot_D(fname):

     D = np.loadtxt(fname)
     plt.figure()
     plt.hist(D,bins = 15,label = "N = 16, L = 16, mean: %.2f"%np.average(D))
     plt.xlim(np.min(D),np.max(D))
     plt.xlabel("Diffusion constant")
     plt.ylabel('Count')
     plt.legend()
     plt.show()
     plt.close()

def plot_densitywithD(fname):

     a = np.loadtxt(fname)
     plt.figure()
     plt.scatter(a[:,0],a[:,1],label="T = 100")
     plt.plot(a[:,0],a[:,1])
     plt.xlabel("Density")
     plt.ylabel('Diffusion constant')
     plt.legend()
     plt.show()
     plt.close()

def plot_meltingtransition(L, fname):
     N = 16
     L = L
     rho = N/L**2
     a = np.loadtxt(fname)
     plt.figure()
     plt.plot(a[:,0], a[:,1], label='$\\rho = %.2f$'%rho)
     plt.xlabel("Temperature(time)")
     plt.ylabel('$(\Delta r)^2$')
     plt.legend()
     plt.show()
     plt.close()

def plot_melting_density(fname):
     a = np.loadtxt(fname)
     plt.figure()
     plt.scatter(a[:,0], a[:,1])
     plt.yscale('log')
     plt.ylim(0.0001,100)
     plt.xlabel("Density")
     plt.ylabel("Melting Temperature")
     plt.show()
     plt.close()
```

```
def plot_melting_r2(fname):
     a = np.loadtxt(fname)
     plt.figure()
     plt.scatter(a[:,0], a[:,1],label='N = 16, L = 4')
     plt.plot(a[:,0], a[:,1])
     plt.xlabel("Temperature")
     plt.ylabel("Diffusion constant")
     plt.legend()
     plt.show()
     plt.close()

def plot_r2(fname):
     a = np.loadtxt(fname)
     plt.figure()
     plt.scatter(a[:,0], a[:,1])
     plt.xlabel("time")
     plt.ylabel("$(\Delta r)^2$")
     plt.show()
     plt.close()

def main():
     print("this is main")
     #plot_D("problem2_data.txt")
     plot_densitywithD("problem2_density_data.txt")
     #plot_meltingtransition(6, "problem3_L6(1).txt")
     #plot_melting_density("problem3_density.txt")
     #plot_melting_r2("problem4_TvsD.txt")
     #plot_r2("problem4_data_T2.07.txt")

if __name__ == '__main__':
     main()
```