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

# In[11]:


# Kenneth J Martinez

import numpy as np
import matplotlib.pyplot as plt
import math
import time


# ## Initializing Constant ##

# In[26]:


den = 0.85                 # density [1/sigma^3]: P2 (a) ~ (c)
T1 = 1.2                    # temperature [epsilon/k]: P2 (a) ~ (c)
denarray = [0.01, 0.10, 0.85]  # density [1/sigma^3]: P2 (d)
T2 = 1.5                    # temperature [epsilon/k]: P2 (d)
ratio_plt = 0.02               # ratio of iteration for test plot
ratio_tr = 0.2              # ratio of iteration to truncate

global M
M = 1                       # mass [m]
Nperdimension = 5           # Number of particles per dimension
N = Nperdimension**3
Iterations = 100000            # number of Iterations
dt = 0.001                  # Timesteps
nu = 1                      # coupling constant for thermostat
global colors
colors = plt.cm.jet([0.5, 0.7, 0.99])
r_all = np.zeros((3, N, Iterations))  # positions [sigma]
v_all = np.zeros((3, N, Iterations))  # velocities [sigma / t]
E_all = np.zeros(Iterations)          # potential energies [epsilon]
K_all = np.zeros(Iterations)          # kinetic energies [epsilon]
T_all = np.zeros(Iterations)          # temperatures [epsilon/k]

def Constants(Nperdimension,density):
  N = Nperdimension ** 3           # number of particles
  L = (N / density) ** (1 / 3)      # lattice size [sigma]
  L0 = L / (2 * Nperdimension)
  return N, L,L0


# ## AsembleCubeLattice function ##

# In[27]:


def AssembleCubeLattice(lattice_diameter, density, num_lattice_x, num_lattice_y, num_lattice_z
):
    num_particles = num_lattice_x*num_lattice_y*num_lattice_z
    lin_density=(density)**(1/3)
    box_size = 1/lin_density*np.asarray([num_lattice_x, num_lattice_y, num_lattice_z])

    pos=np.zeros([num_particles, 3])
    incr_x = lattice_diameter
    incr_y = lattice_diameter
    incr_z = lattice_diameter
    bead_index = 0
    for i in range(num_lattice_z):
        for j in range(num_lattice_y):
            for k in range(num_lattice_x):
```

```
                pos[bead_index, 0] = k*incr_x + 0.5*lattice_diameter
                pos[bead_index, 1] = j*incr_y + 0.5*lattice_diameter
                pos[bead_index, 2] = i*incr_z + 0.5*lattice_diameter
                bead_index = bead_index + 1
    return pos, box_size


# ## Function to assign initial velocities ##

# In[28]:


vels = np.zeros([125, 3])
def AssignInitialVelocities( temp, k_b, mass, num_particles ):
  for i in range(num_particles):
          # Maxwell-Boltzmann distribution for velocities is a Gaussian
          # distribution with mean = 0 and standard deviation = sqrt(kT/m).
          # randn(3,1) generates 3 random numbers from a Gaussian
          # distribution with mean = 0 and standard devition = 1. A property
          # of a variable, X, that is Gaussian distributed is that
          # multiplying by a factor A does not change its mean, but
          # multiplies its standard deviation by A. Therefore we multiply the
          # results of randn() by the desired standard deviation to draw from
          # a M-B distribution.
      vels[i,:] = np.random.randn(3)*(k_b*temp/M)**(1/2)
      # A few tricks here want to prevent the entire system having a net
      # velocity (flying ice box), so we remove center of mass velocity from each
      # particle to set center of mass velocity to 0.
      com_vel = np.zeros([1, 3])
      for i in range(num_particles):
          com_vel = com_vel + vels[i, :]
      com_vel = com_vel / num_particles
      for i in range(num_particles):
          vels[i,:] = vels[i, :] - com_vel

      # Next, rescale velocities by calculating current temperature, then multiplying
      # all velocities uniformly to get new correct temperature. Calculate temperature from
      # T = 2/3k * KE = 1/3k * m v^2
      # Could have combined with above step, but keep separate to show main
      # idea.

      cur_ke = 0.0
      for i in range(num_particles):
        cur_ke = cur_ke + 0.5*mass*np.dot(vels[i, :], vels[i, :])
      cur_temp = 0.66 * cur_ke / num_particles
      # Get scaling factor
      scale_factor = np.sqrt(temp /cur_temp)
      # multiply all velocities by scale factor to get new correct temp
      new_ke = 0.0
      for i in range(num_particles):
          vels[i,:] = vels[i,:] * scale_factor
          new_ke = new_ke + 0.5*mass*np.dot(vels[i,:], vels[i,:])

      rescale_temp = 0.66 * new_ke / num_particles
      #print('Original temp =',cur_temp ,', rescaled temp =',  rescale_temp)
      v = np.transpose(vels)
      return v


# ## Force and Potential Energy Function ##

# In[29]:


def Force_Energy(r, L):
# r: particle coordinates [sigma], dimension of (3, N)
# L: lattice size [sigma]
```

```
    F = np.zeros((3, N))  # force [epsilon / sigma]
    PE = 0                # potential energy [epsilon]

    # Calculation of force and potential energy
    r_ij = np.zeros((3, N, N))
    for k in range(3):
        for i in range(N):
            r_ij[k, i, :] = r[k, :] - r[k, i]
    r_ij = r_ij - L * np.round(r_ij / L)  # distance: periodic B.C.
    r2 = np.sum(r_ij ** 2, 0)             # square distance calculation
    r2i = np.where(r2 == 0, 0, 1 / r2)
    r6i = r2i ** 3
    F_ij = -np.tile(48 * r2i * r6i * (r6i - 0.5), (3, 1, 1)) * r_ij
    F = np.sum(F_ij, 2)                   # force calculation
    E_ij = 4 * r6i * (r6i - 1)
    PE = np.sum(E_ij) / 2                 # potential energy calculation

    return F, PE, r2
    # r2: square distance of all interactions, dimension of (N, N)


# ## Molecular dynamics for Lennard Jones Function ##

# In[30]:


def MolecularDynamics(density, T, opt = 0):
# density: particle density [1/sigma^3]
# T: temperature [epsilon/k]
# opt: 0 (default), 1 (Anderson thermostat)
    # Calling for Constants
    N, L, L0 = Constants(Nperdimension,density)
    # Matrices to save values
    r_all = np.zeros((3, N, Iterations))  # positions [sigma]
    v_all = np.zeros((3, N, Iterations))  # velocities [sigma / t]
    E_all = np.zeros(Iterations)          # potential energies [epsilon]
    K_all = np.zeros(Iterations)          # kinetic energies [epsilon]
    T_all = np.zeros(Iterations)          # temperatures [epsilon/k]
    # Anderson thermostat settings
    if opt == 1:
        p0 = nu * dt  # probability of selection
        p = np.tile(np.random.rand(N, Iterations) <= p0, (3, 1, 1))
        vm = np.random.randn(3, N, Iterations) * np.sqrt(T / M)

    # Initial positions
    L0 = L / (2 * Nperdimension)
    r0 = np.linspace(L0, L - L0, Nperdimension)
    r = np.zeros((3, N))
    for i in range(Nperdimension):
        for j in range(Nperdimension):
            for k in range(Nperdimension):
                r[:, Nperdimension * (Nperdimension * i + j) + k] = [r0[i], r0[j], r0[k]]
    rT = np.transpose(AssembleCubeLattice(L,density, 5,5,5))
    # Calling for Initial Velocities
    v = AssignInitialVelocities( T,1,M, N )

    # Initial force
    np.seterr(divide = 'ignore')
    F, _, _ = Force_Energy(r, L)



    # Iterations by velocity-Verlet integrator
    for i in range(Iterations):
        if i % 10000 == 0:
            print('Run: ' + str(i) + ' of ' + str(Iterations))
```

```
        mult = dt / (2 * M)
        v_half = v + F * mult
        r = (r + v_half * dt) % L                    # position: periodic B.C.
        F, E_all[i], r2 = Force_Energy(r, L)                # force calculation
        v = v_half + F * mult                        # update velocities
        if opt == 1:
            v = np.where(p[:, :, i], vm[:, :, i], v)  # Anderson thermostat
        r_all[:, :, i] = r
        v_all[:, :, i] = v
        K_all[i] = 0.5 * M * np.sum(v ** 2)
    np.seterr(divide = 'raise')
    E_all /= N
    K_all /= N
    T_all = K_all * (2 / 3)
    d_ij = np.sqrt(r2)
    return r_all, v_all, E_all, K_all, T_all, d_ij
    # d_ij: distance of all interactions for the last iteration


# ## Function to plot kinetic, potential, and total energy ##

# In[31]:


def plot_KE_PE_Tot(E, K, text, N_plot = 0):
# E: potential energy [epsilon]
# K: kinetic energy [epsilon]
# text: plot file name
# N_plot: length of plot (default is 0 for full length)

    # Other variables
    E_tot = E + K
    Iterations = np.size(E)
    N = range(Iterations)
    if N_plot == 0:
        N_plot = Iterations

    # Plot and save figure
    print('Plotting: ' + text)
    plt.plot(N[:N_plot], E_tot[:N_plot], '-*', color = colors[0], label = 'Total Energy')
    plt.plot(N[:N_plot], E[:N_plot], '-*', color = colors[1], label = 'Potential Energy')
    plt.plot(N[:N_plot], K[:N_plot], '-*', color = colors[2], label = 'Kinetic Energy')

    plt.title('Particle Energy: ' + text)
    plt.xlabel('# of Iterations')
    plt.ylabel('Energy [$\epsilon$]')
    plt.legend()
    plt.show()


# ## Function to plot speed distribution ##

# In[32]:


def plot_speed(v, T, text):
# v: velocity [sigma / t], dimension of (3, N)
# T: temperature [epsilon]
# text: plot file name

    # User-defined settings
    wd = 0.2
    N_ref = 1000

    # Histogram:
    N = np.size(v, 1)
    sp = np.sqrt(np.sum(v ** 2, 0))
```

```
    max_sp = math.ceil(np.amax(sp))
    N_bar = int(max_sp / wd)
    w = np.ones(N) / (N * wd)

    # speed distribution
    sp_ref = np.linspace(0, max_sp, N_ref)
    sp2 = sp_ref ** 2
    p_ref = (M/(2*T*math.pi)) ** (3 / 2) * (4 * math.pi * sp2)          * np.exp(-M *
sp2/(2*T))

    # Histogram plotting
    plt.hist(sp, N_bar, (0, max_sp), density = True, weights = w,
         rwidth = 0.9, color = colors[2], label = text)
    plt.plot(sp_ref, p_ref, 'k-', label = 'Maxwell-Boltzmann')
    plt.title('Speed Distribution (' + str(wd) + '$\sigma/t$): ' + text)
    plt.xlabel('Speed [$\sigma/t$]')
    plt.ylabel('Probability Density')
    plt.legend()
    plt.show()


# In[ ]:




# In[33]:


def plot_radial(d_ij, density, text):
# d_ij: distance of all interactions, dimension of (N, N)
# density: density [1/sigma^3]
# text: plot file name

    # User-defined settings
    wd = 0.02

    # Histogram settings
    N = np.size(d_ij, 0)
    N_d = int(N * (N - 1) / 2)
    d = np.zeros(N_d)
    k = 0
    for i in range(N-1):
        for j in range(i+1, N):
            d[k] = d_ij[i, j]
            k = k + 1
    max_d = math.ceil(np.amax(d))
    N_bar = int(max_d / wd)
    w = np.zeros(N_d)
    for k in range(N_d):
        l = math.floor(d[k] / wd)
        vol = (4 / 3) * math.pi * ((l + 1) ** 3 - l ** 3) * (wd ** 3)
        w[k] = 1 / (density * vol * N * N_d)

    # Histogram plotting
    plt.hist(d, N_bar, (0, max_d), density = True, weights = w,
             rwidth = 0.9, color = colors[2])
    plt.title('Radial Distribution (' + str(wd) + '$\sigma$): ' + text)
    plt.xlabel('Distance [$\sigma$]')
    plt.ylabel('Probability Density')
    plt.savefig('Radial Distribution ' + text + '.jpg')
    plt.show()


# ## Plotting Simulation for part a ##
```

```
# In[34]:


# Simulation part a
print('Simulation part a')
t_start = time.time()
ra, va, Ea, Ka, Ta, _ = MolecularDynamics(den, T1)
t_end = time.time()
print(f"Runtime of P2 (a): {t_end - t_start}s")

Iterations = np.size(Ea)
N_test = int(Iterations * ratio_plt)
plot_KE_PE_Tot(Ea, Ka, 'P2 (a)')
plot_KE_PE_Tot(Ea, Ka, 'P2 (a) test', N_test)


# ## Plotting Simulation for part b ##

# In[35]:


# Simulation part b
print('\nSimulation part b')
t_start = time.time()
rb, vb, Eb, Kb, Tb, _ = MolecularDynamics(den, T1, 1)
t_end = time.time()
print(f"Runtime of P2 (b): {t_end - t_start}s")
plot_KE_PE_Tot(Eb, Kb, 'P2 (b)')
plot_KE_PE_Tot(Eb, Kb, 'P2 (b) test', N_test)


# ## Plotting Simulation for part c ##

# In[36]:


# Simulation part c
print('\nAnalysis part c')

# Calculation of ensemble-average temperature for P2 (a) ~ (b)
N_trunc = int(Iterations * ratio_tr)
print('Ensemble-average Temperature of P2 (a): ' + str(np.mean(Ta[N_trunc:])))
print('Ensemble-average Temperature of P2 (b): ' + str(np.mean(Tb[N_trunc:])))

# Plot of speed distribution for P2 (a) ~ (b)
plot_speed(va[:, :, -1], T1, 'P2 (a)')
plot_speed(vb[:, :, -1], T1, 'P2 (b)')


# ## Plotting Simulation part d and e ##

# In[39]:


# Simulation: P2 (d)
print('\nAnalysis: P2 (d)')

# Simulation for different density and T
rd1, vd1, Ed1, Kd1, Td1, d_ij1 = MolecularDynamics(denarray[0], T2)
rd2, vd2, Ed2, Kd2, Td2, d_ij2 = MolecularDynamics(denarray[1], T2)
rd3, vd3, Ed3, Kd3, Td3, d_ij3 = MolecularDynamics(denarray[2], T2)
plot_radial(d_ij1, denarray[0], 'density 0.01')
plot_radial(d_ij2, denarray[1], 'density 0.10')
plot_radial(d_ij3, denarray[2], 'density 0.85')
```

```
# In[ ]:
```