[Jayadevan]-Ex04-Free_Energy

November 17, 2023

1 Exercise IV: Free Energies

Karthik Jayadevan

Matriculation Number: 5582876

In this exercise, we address three specific tasks involving the study of free energy landscapes in a 2D particle system within a 10 nm x 10 nm box with periodic boundary conditions.

- The first task involves setting up the simulation, creating 100 spherical particles with specified properties, and conducting a simulation for 5000 steps.
- The second task focuses on the radial distribution function (RDF) $g(r)$. The time- and particle-averaged RDF for the system simulated in the first task is calculated and displayed.
- The third task is dedicated to calculating the potential of mean force $\Delta F(r)$. Here, we first determine the system temperature from the mean kinetic energy and then calculate and plot the two-particle $\Delta F(r)$.

[1]:
```
# Extend the system path to include the custom module
# `ccs_basis`, which contains the helper functions for the
# simulation.

import sys
sys.path.append('..')

# custom module (will be added to the end of the notebook)
from ccs_basis import *

# set the custom plotting style
plt.style.use('../ccs_ex.mplstyle')
```

1.1 Task I: Simulation

First the 2D simulation box with 10nm x 10 nm is set up with the following properties: - 100 spherical particles, - Radius $r = 0.2$ nm, - mass $m = 0.2$ g/mol, - velocities of magnitude $|\vec{v}| = 0.5$ nm/ns along random directions - 5000 steps - time step length $\Delta t = 0.1$ ns

First the particle class is rewritten to introduce the property of mass to the particle:

[2]:
```
class Particle:
    def __init__(self, r, x, y, vx, vy, mass):
        self.r = r
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.mass = mass # new property

    def __repr__(self):
        return str(f"This is a particle of mass {self.mass} at {self.x}, {self.
 ↪y} with v = {self.vx},{self.vy} nm/timestep")
```

The following function is the same simulation function from the previous exercises, with minor changes: - adds a mass of 2 units to all particles - returns the array of particle objects to be used for the temperature calculation

[3]:
```
def do_the_simulation_v3(num_particles,steps,timestep=1,box=[10,10],v_norm=0.
 ↪5,pbound=True,particle_size=0.2):

    particles = [] # list to store the particle objects
    dots = [] # list to store the markers for each particle

    fig, ax = plt.subplots()
    fig, ax = setup_fig_box(fig,ax,box,'System with {0} particles'.
 ↪format(num_particles))


    for _ in range(num_particles):
        m = 2 # g/mol
        rand_vx, rand_vy = rand_velocity(v_norm)
        particles.append(Particle(r=particle_size,
                                  x=rand_coordinate(box[0],particle_size),
                                  y=rand_coordinate(box[0],particle_size),
                                  vx=rand_vx,
                                  vy=rand_vy,
                                  mass=m))
        dots.append(ax.plot([], [], 'ro')[0])

    data_traj = np.zeros((num_particles, 4, steps))

    for i in range(steps):
        # at each time step:
        for j, p in enumerate(particles):
            move(p, timestep)
            # j - particle index
            # p - particle object
```

```
            if pbound:
                periodic_bound(p, box)
            else:
                reflect(p,box)
            # input the coordinates to trajectory:
            data_traj[j, :, i] = [p.x, p.y, p.vx, p.vy]

        # checking pairs of particles for collision
        for j in range(num_particles):
            # j - again, particle index
            for k in range(j+1, num_particles):
                # k - loop over particles other than j
                if is_colliding(particles[j], particles[k]):
                    particles[j], particles[k] =␣
↪elastic_collision(particles[j], particles[k])

    def init():
        for dot in dots:
            dot.set_data([], [])
        return dots

    def animate(i):
        for j, dot in enumerate(dots):
            x = [data_traj[j, 0, i]]
            y = [data_traj[j, 1, i]]
            dot.set_data(x, y)
        return dots

    anim = animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=steps, interval=20, blit=True)

    plt.close(fig)

    return data_traj, anim, particles
    # trajectory, animation object and array of particle objects returned
```

The simulation is executed:

[4]:
```
data,an,particles =␣
↪do_the_simulation_v3(num_particles=100,steps=5000,timestep=0.1,pbound=True)
# pbound = True enables the periodic boundary conditions

# HTML(an.to_html5_video())
```

[5]:
```
data.shape
```

[5]:
```
(100, 4, 5000)
```

1.2 Task II: Radial Distribution Function

Now, the functions to compute the RDF are defined below:

- The `calculate_distances` function computes the pairwise distances between particles in a 2D box, considering periodic boundary conditions.
  - It takes as input the positions of the particles and the size of the box.
  - The function iterates over each pair of particles, calculates the difference in their positions. The distances are stored in a matrix, from which only the upper triangle is extracted and flattened into a 1D array (indices ij represent the distance between particles i and j).
  - To take care of the periodic boundary conditions, the *minimum image convention* is employed (Source).
- The `compute_rdf` function calculates the radial distribution function (RDF) from the trajectory of particle positions.
  - It takes as input the trajectory, the size of the box, the number of bins for the histogram, and the maximum radius to consider.
  - Since periodic boundary conditions are used, the **maximal distance** r for which the RDF is to be evaluated is half the size of the box. In our case, the $r_{\max} = 5$ nm.
  - The function initializes a histogram for the RDF, calculates the bin edges, and computes the bin width.
  - For each step in the trajectory, it calculates the pairwise distances and updates the RDF histogram.
  - Finally, the RDF is normalized by the volume of the box, the particle density, the number of steps, the number of particles, and the volume of the spherical shells corresponding to each bin.
  - The function returns the radii (midpoints of the bins) and the normalized RDF.

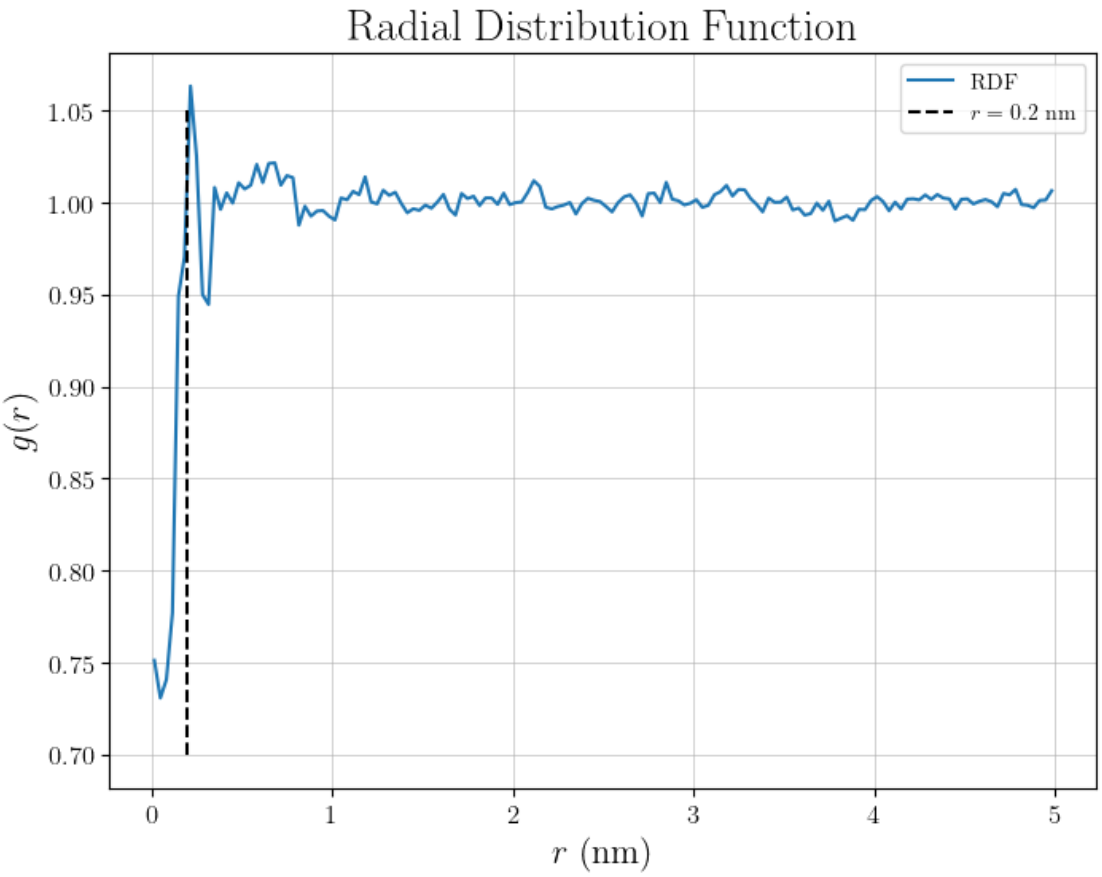
[6]:
```python
def calculate_distances(positions, box_size):
    num_particles = positions.shape[0]

    # square matrix to store the distances:
    distances = np.zeros((num_particles, num_particles))

    for i in range(num_particles):
        for j in range(i + 1, num_particles):
            delta = positions[i, :2] - positions[j, :2]
            #calculates the difference in both the x and y positions

        # Applying minimum image convention:
            # calculate how many box lengths away the particles are,
            # then adjust the initial delta by subtracting the number of box lengths times the box size.
            delta = delta - np.round(delta / box_size) * box_size
            # the shortest wrapped distance between the particles across the periodic boundaries.

            distance = np.sqrt(np.sum(delta**2))
```

```
            distances[i, j] = distance

    # index ij and ji represent the same distance, so only one of them is␣
 ↪required for each i,j pair
    # index ii represent the distance between the same particle, which is␣
 ↪always zero.
    # Hence we are only interested in the upper/lower triangle elements of the␣
 ↪matrix,
    # excluding the diagonal
    valid_distances = distances[np.triu_indices(num_particles, k=1)].flatten()

    # triu_indices(num_particles,k=1) generates the indices for the upper␣
 ↪triangle
    # of an array with num_particles dimensions, excluding the diagonal (k=1).

    # the distances are flattened for plotting the histogram

    return valid_distances

def compute_rdf(trajectory, box_size, num_bins, max_radius):
    # Get the number of particles and steps from the trajectory shape

    num_particles, _, num_steps = trajectory.shape

    # Initialize the RDF histogram with zeros for each bin
    rdf_histogram = np.zeros(num_bins)

    # Create equally spaced bin edges from 0 to max_radius
    bin_edges = np.linspace(0, max_radius, num_bins + 1)

    # Correctly calculate bin_width based on bin_edges
    bin_width = bin_edges[1] - bin_edges[0]

    # Loop over each step in the trajectory to accumulate distances
    for step in range(num_steps):
        positions = trajectory[:, :, step] # positions of all particles at␣
 ↪current step

        # Calculate all unique inter-particle distances
        valid_distances = calculate_distances(positions, box_size)

        # Bin the distances and add to the histogram
        hist, _ = np.histogram(valid_distances, bins=bin_edges, range=(0,␣
 ↪max_radius))
        rdf_histogram += hist
```

```
    # Normalization
    V = box_size**2 # volume (area) of simulation box
    rho = num_particles / V # particle density
    radii = 0.5 * (bin_edges[:-1] + bin_edges[1:])  # Midpoints of bins for␣
↪plostting

    # Normalize the RDF histogram by considering the number of particles,
    # the volume of the spherical shells (area of rings here) corresponding to␣
↪the bins,
    # and the particle density
    normalization = (np.pi * radii * bin_width * rho * num_steps *␣
↪(num_particles - 1))

    rdf = rdf_histogram / normalization

    return radii, rdf
```

[7]:
```
radii, rdf = compute_rdf(data, box_size=10, num_bins=150, max_radius=5)
```

[21]:
```
plt.figure(figsize=(8, 6))
plt.plot(radii, rdf,'-', label="RDF")
plt.xlabel(r"$r$ (nm)")
plt.ylabel(r"$g(r)$")
plt.title("Radial Distribution Function")
plt.grid(True)
plt.vlines(x=[0.2],ymin=0.7,ymax=1.05,colors='k',linestyles='--',label=r'$r=0.
↪2$ nm')
plt.legend()
plt.savefig('rdf.png')
```



- The RDF has an initial peak close to the particle radius.
- This peak suggests a higher probability of finding another particle at a distance approximately equal to the particle size.

- As the distance r increases beyond this peak, the RDF value decreases, and fluctuates around the value 1 at longer distances. For $r \gg R$, $g(r) \to 1$, as expected. This indicates that particles are not ordered, and are isotropic, resembling ideal gas.
- For a hard-sphere model, we expect the RDF to be zero for $r < R$ where R is the particle radius.

1.3 Task III: Potential of mean force

[9]:
```
import scipy.constants as sc
```

For a 2D simulation (i.e., two degrees of freedom), according to equipartition theorem, we have

$$\bar{E}_K = \frac{1}{2}kT + \frac{1}{2}kT = kT$$

So the temperature can be calculated as:

$$T = \bar{E}_K/k$$

[10]:
```
k = sc.k
amu = sc.m_u # 1 atomic mass unit in kg
E = 0

for p in particles:
    E += 0.5*p.mass*amu*(p.vx**2 + p.vy**2)

E /= len(particles)
T = E/k
print('Temperature of box:',T)
```

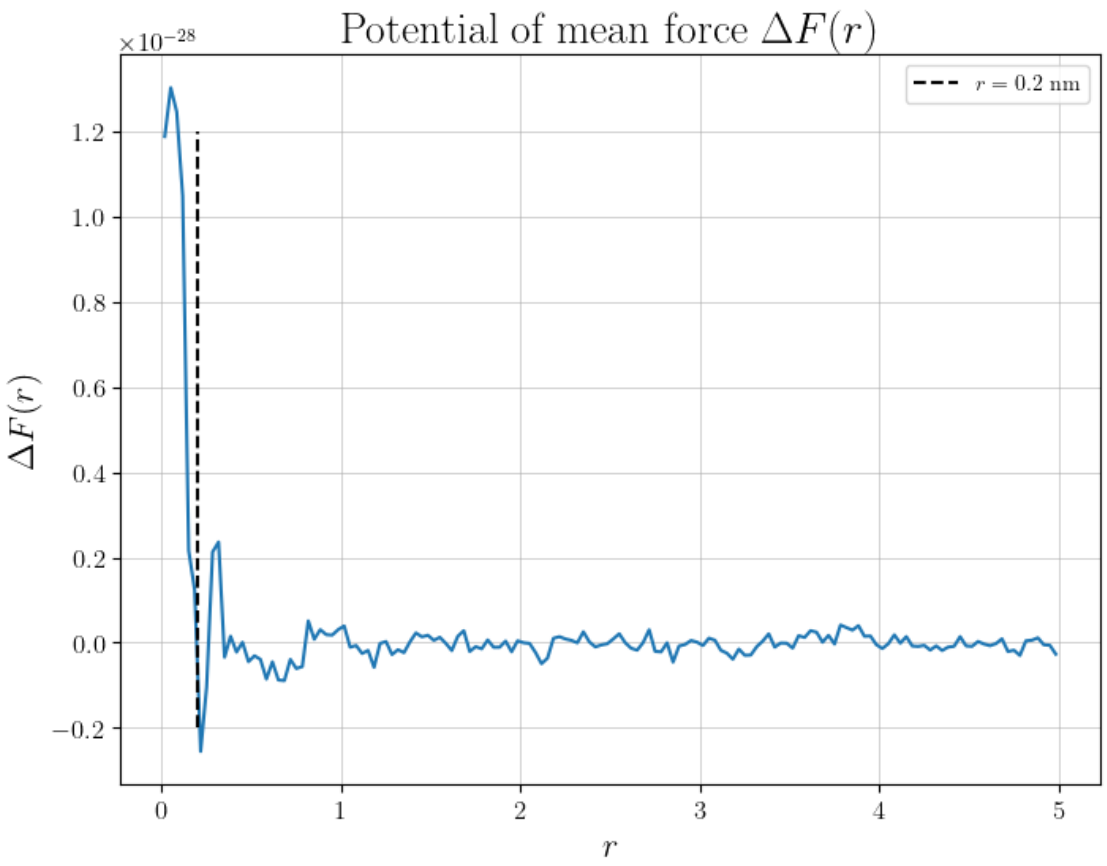
```
Temperature of box: 3.0068088750290575e-05
```

Hence the temperature of the system is obtained as $ 3 ×10^{-5} K$.

Now, the potential of mean force can be calculated as:

$F = -kT \ln g(r)$

[32]:
```
x = int(5/0.01)
# empty array:
F = np.ones(x)

# calculating F:
F = -k*T*np.log(rdf)

# Plotting
plt.figure(figsize=(8,6))
plt.plot(radii,F)
plt.title(r'Potential of mean force $\Delta F(r)$')
plt.xlabel(r'$r$')
plt.ylabel(r'$\Delta F(r)$')
plt.vlines(x=0.2,ymin=-0.2e-28,ymax=1.
 ↪2e-28,colors='k',linestyles='--',label=r'$r=0.2$ nm')
plt.legend()
plt.savefig('meanForce.png')
```



The minima of $\Delta F(r)$ represent stable states. One can see that the minima occurs close to 0.2, which is the particle radius. As r increases, this function gets closer to zero. Here, as a result of the behaviour of $g(r)$, we observe $\Delta F(r) \to 0$ for $r \gg R$. This shows that particles are non interacting on longer ranges.