Report of FFR120 HM2: Chapter 5-8

Siyu Hu, gushusii@student.gu.se, ID: 19950910-3702

2024-11-17

1 Exercise 1 Brownian Dynamic

1.1 Q1

x different stiffnesses 0.018849555921538756

y different stiffnesses 0.0020943951023931952

set $dt = 0.00010471975511965977$

1.2 P1

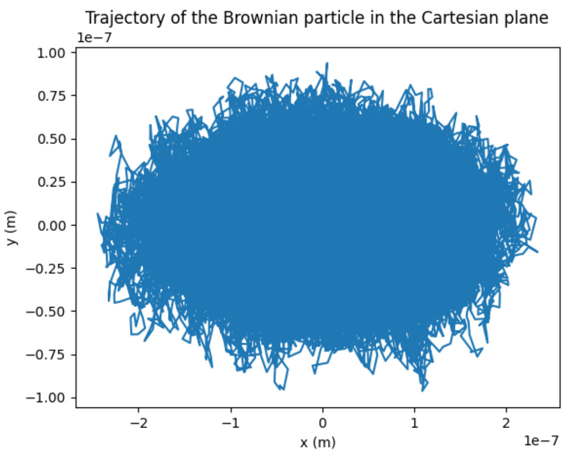


Figure 1: Trajectory

1.3 P2

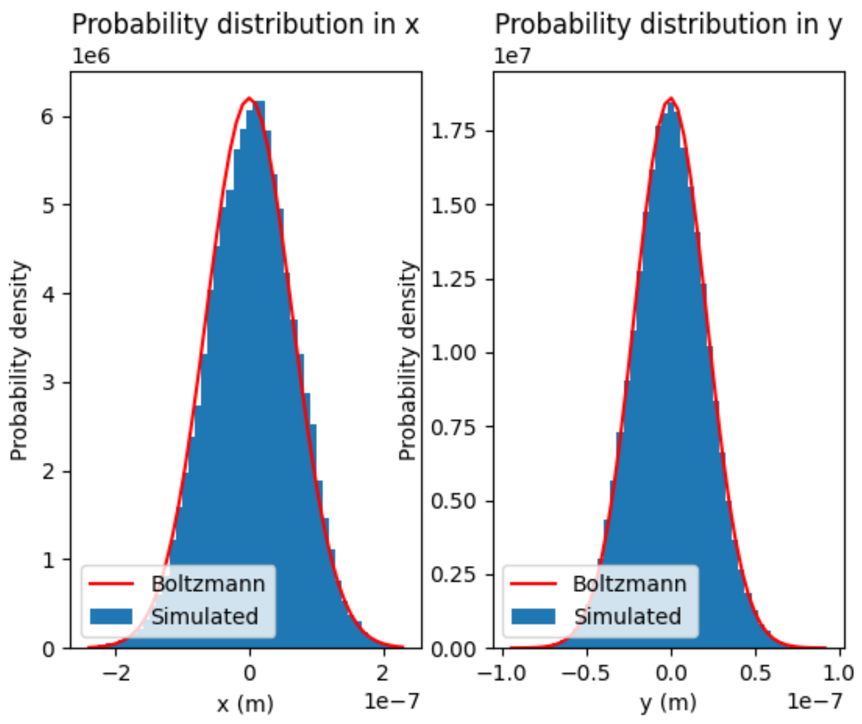


Figure 2: Probability distribution

1.4 Q2

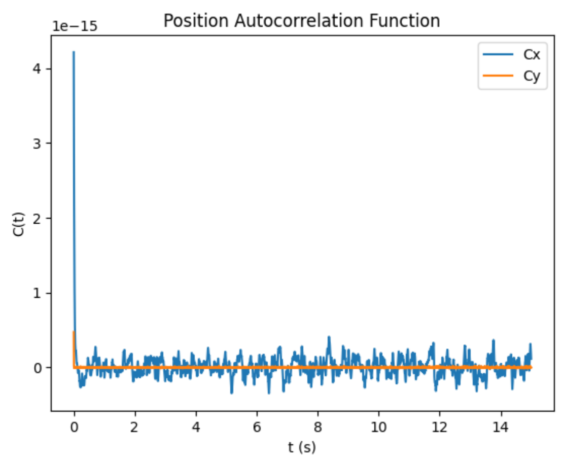
In this experiment, $(\sigma_x)^2 = 4.1966314582651 96e-15$, $(\sigma_y)^2 = 4.607489519704521e-16$, x variance is larger.

Theorical variance in a harmonic trap, $(\sigma_x)^2 = 4.1399999999999994e{-}15$, $(\sigma_y)^2 = 4.5999999999999999e{-}16$.

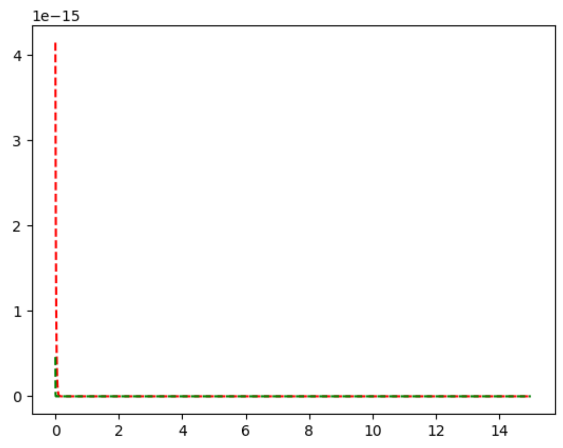
Overall, the experimental values are close to the theoretical values for both x and y variances, indicating that the experiment is in good agreement with the theoretical model for a harmonic trap.

1.5 P3 - P4

As time increases, the position autocorrelation of the particle decreases, indicating that the particle has forgotten its initial position.



(a) Postion autocorrelation



(b) Theoretical position autocorrelation

Figure 3: Postion autocorrelation

2 Exercise 2 Anomalous Diffusionl

2.1 P1

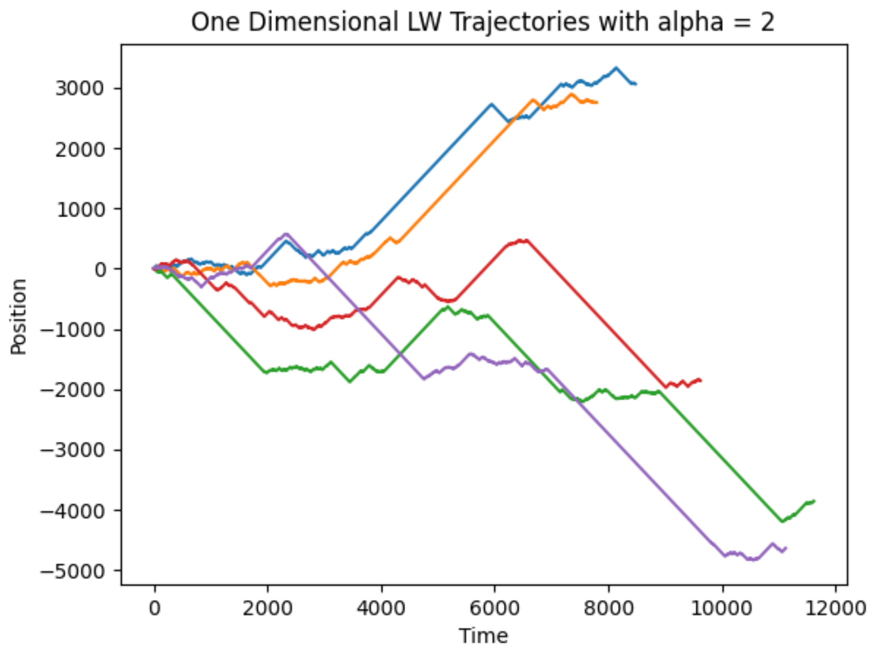


Figure 4: One Dimentional LW Trajectories

2.2 P2

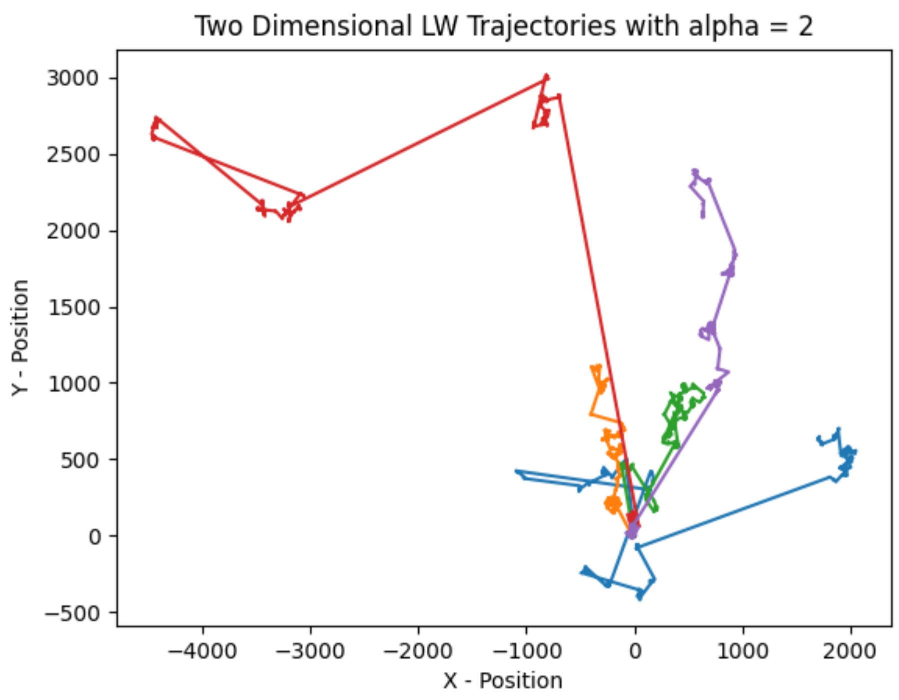


Figure 5: Two Dimensional LW Trajectories

2.3 P3 - P4

A larger value($\alpha > 1$) may cause the diffusion to deviate more from normal diffusion (Brownian motion) and exhibit superdiffusion characteristics. Specifically, MSD may grow at a faster rate than ($\alpha <= 1$), and the relationship with time may show a higher power relationship

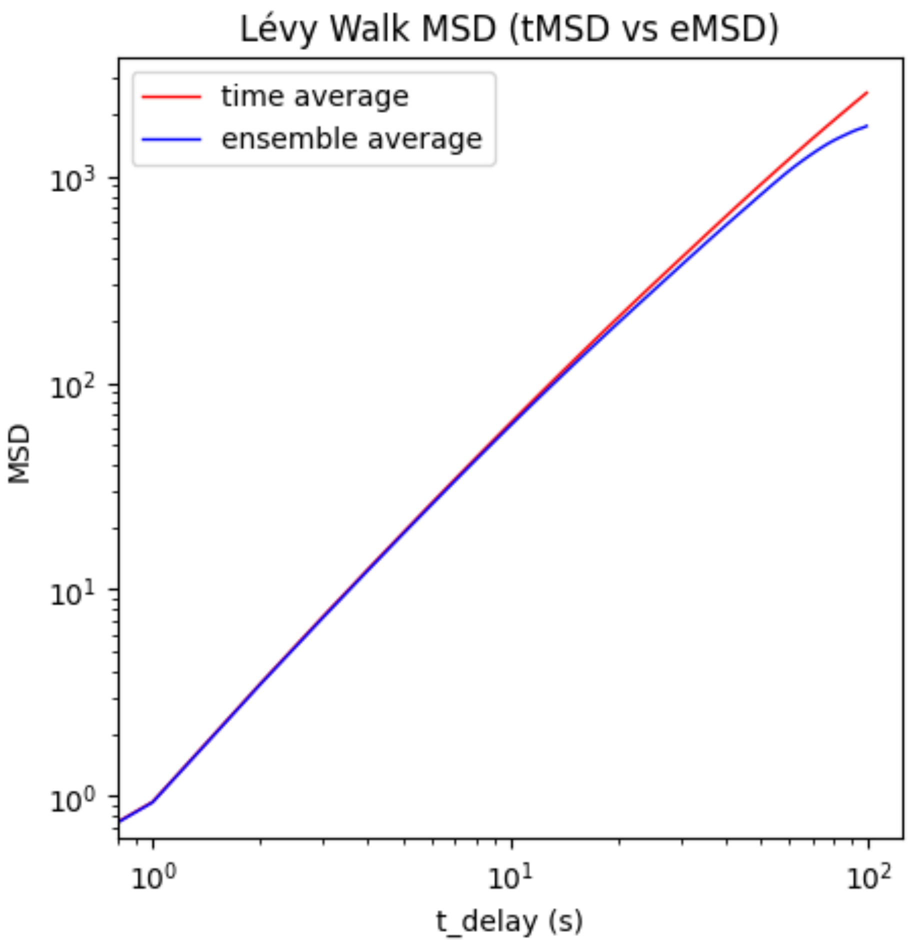


Figure 6: Mean Square Displacement for 1D LW (alpha = 2)

3 Exercise 3 Multiplicative Noise

3.1 P1

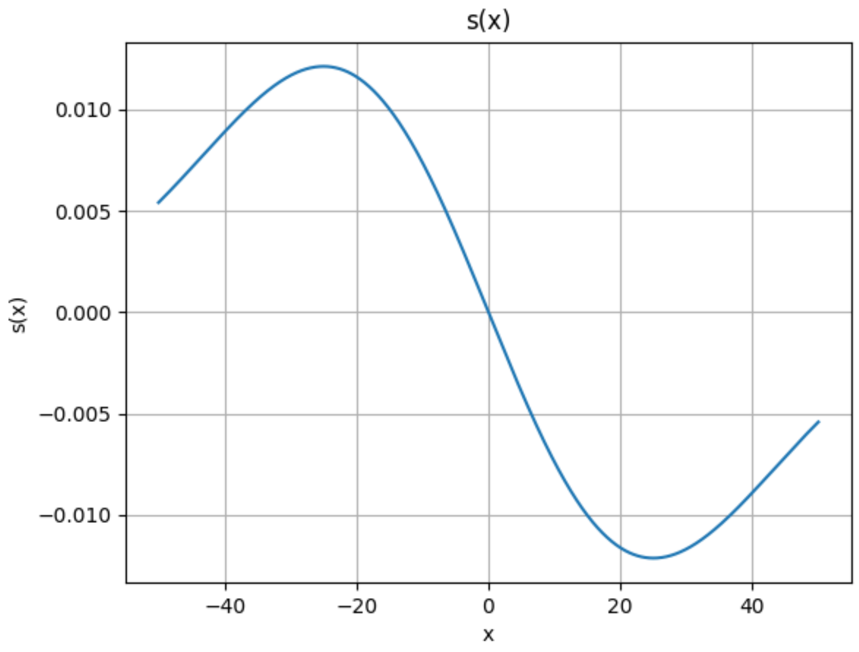


Figure 7: s(x)

3.2 P2 P3 P4

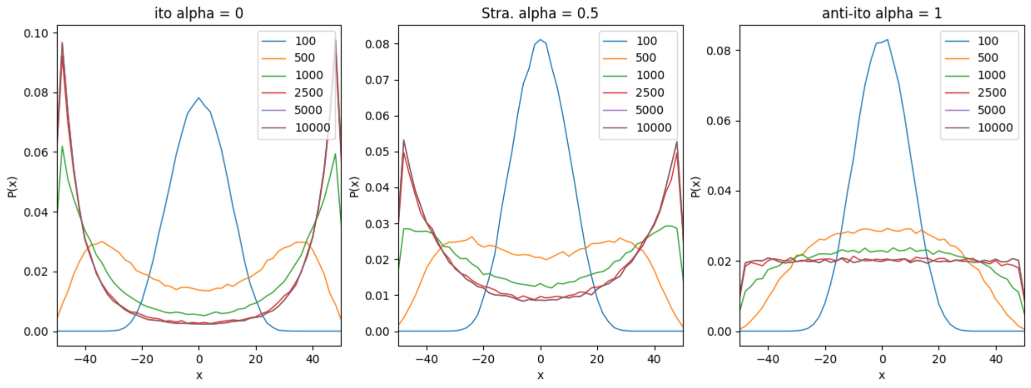


Figure 8: Final position distribution with ito, Stratonovich, anti-ito convention

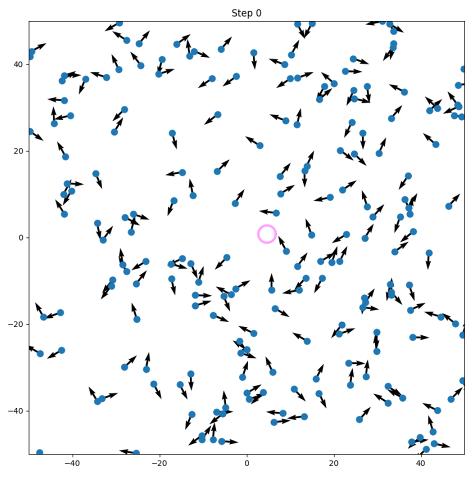
3.3 Q1

The three distribution of the final points symmetrical, because the $\sigma(x)$ noise formula is symmetrical.

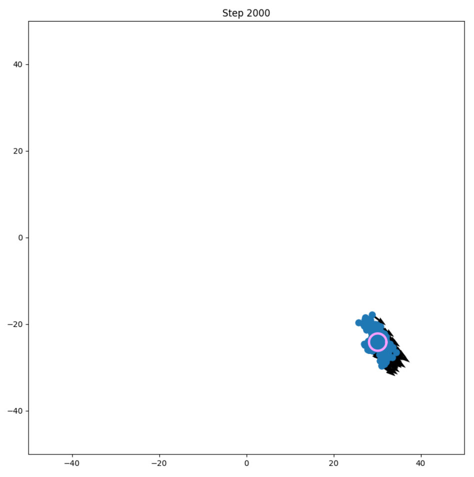
4 Exercise 4 The Vicsek Model

4.1 Task 1 P1

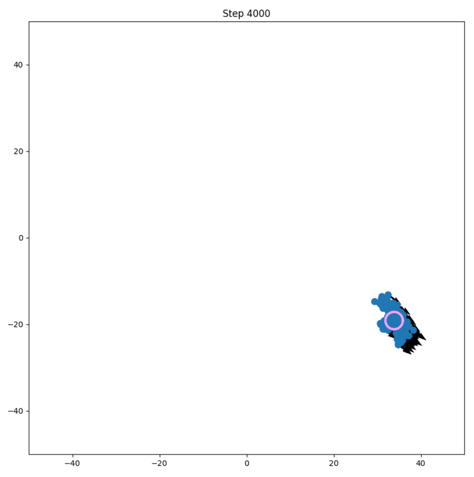
From the Figure below, one can observation that the particles will become more clustering and share the same orientation with each other.



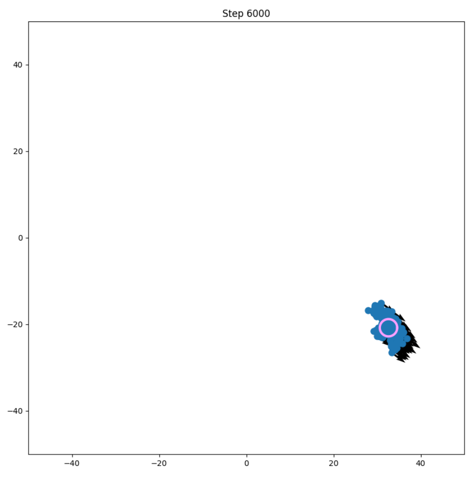
(a) a) time step = 0



(b) b) time step = 2000



(c) c) time step = 4000



(d) d) time step = 6000

Figure 9: Particles status

4.2 Task 1 P2

The global coefficient of alignment is close to 1.0 over 6000 time steps, as global coefficient of clustering is close to 0.0.

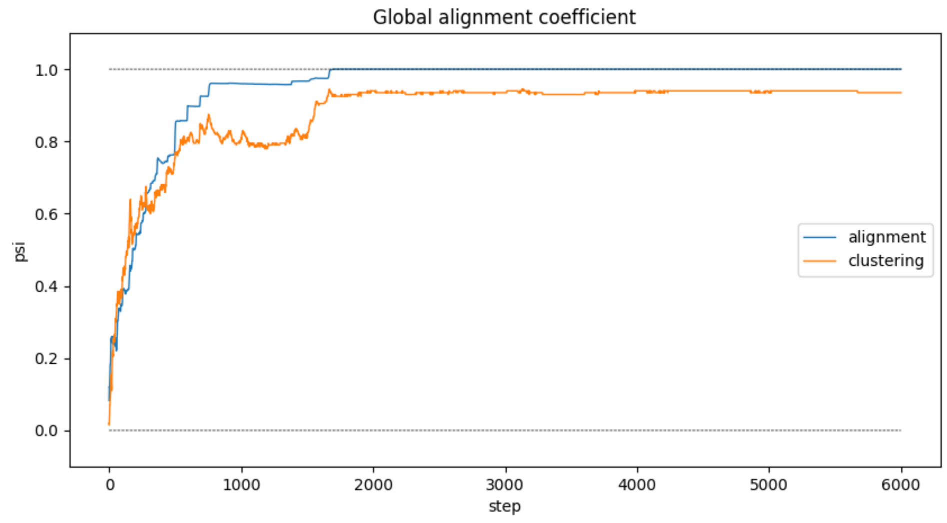
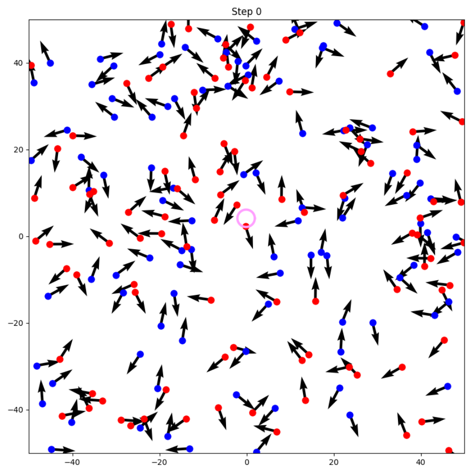


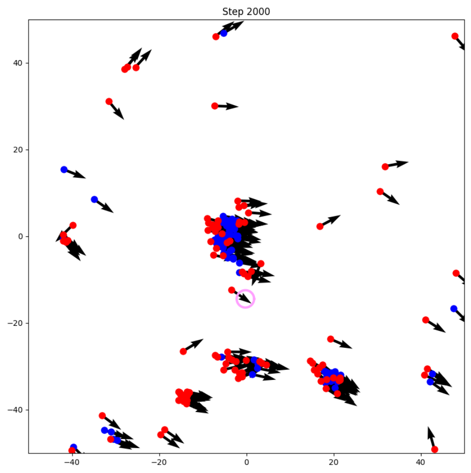
Figure 10: Enter Caption

4.3 Task 2 P3

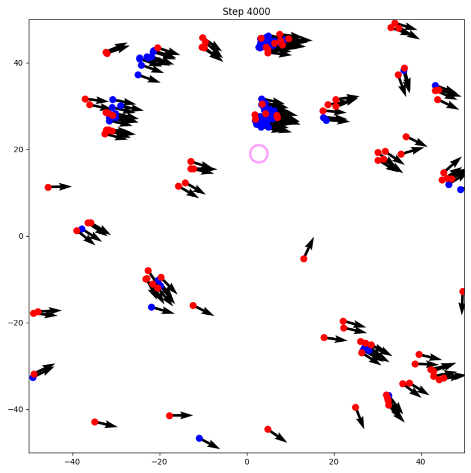
The Vicsek model with 2 subpopulation of particles.



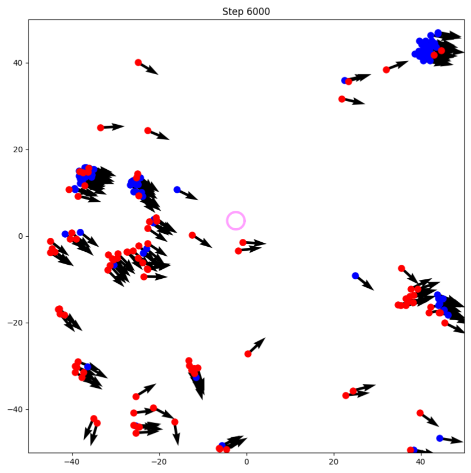
(a) a) time step = 0



(b) b) time step = 2000



(c) c) time step = 4000



(d) d) time step = 6000

Figure 11: Particles status

4.4 Task 2 P4

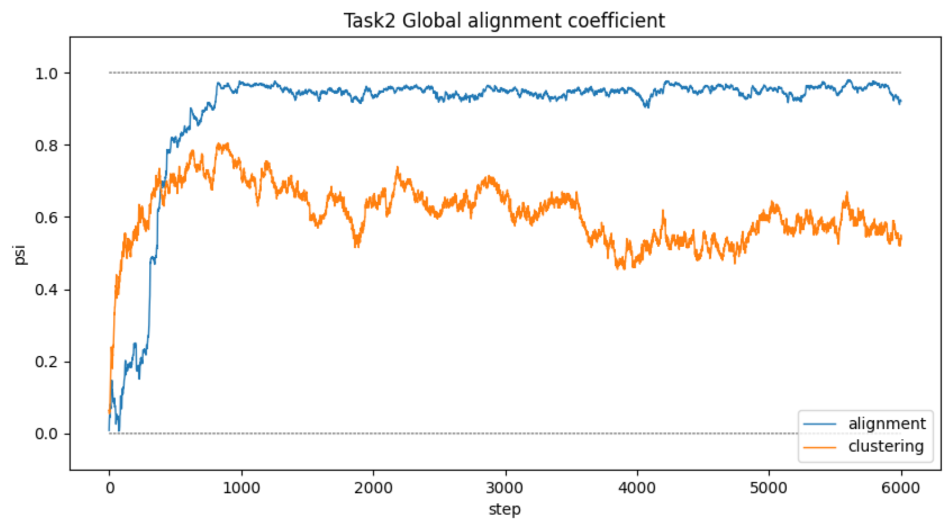


Figure 12: Enter Caption

4.5 Task 2 Q1

Comparing Figures P1 and P4, one can conclude that having a population with two distinct traits will result in a lower upper limit for the global clustering coefficient and increased fluctuations. Similarly, it will also lead to greater fluctuations in the global alignment coefficient. This indicates that the two groups of particles cannot achieve the same level of clustering as a single group within the same time frame, and their distribution within the region will remain relatively scattered.

2.1

November 18, 2024

1 Q1

define delta_t

```
[20]: import numpy as np
import matplotlib.pyplot as plt

k_b = 1.380 * 10**(- 23)  # J/K, Boltzmann Constant
T = 300  # K , Temperature
eta = 10**(- 3)  # Ns/m², viscosity
R = 10**(- 6)   # m  , radius of particle
k_x = 10**(- 6)  # N/m, sti ness
k_y = 9 * 10**(- 6)  # N/m. sti ness

gamma = 6 * np.pi * eta * R
tau_rap_x = gamma / k_x
tau_rap_y = gamma / k_y

print("x di erent sti nesses ", tau_rap_x)
print("y di erent sti nesses ", tau_rap_y)

dt_x = 0.05 * tau_rap_x
dt_y = 0.05 * tau_rap_y
dt = np.minimum(dt_x, dt_y)
print(f"timestep = {dt}" )
```

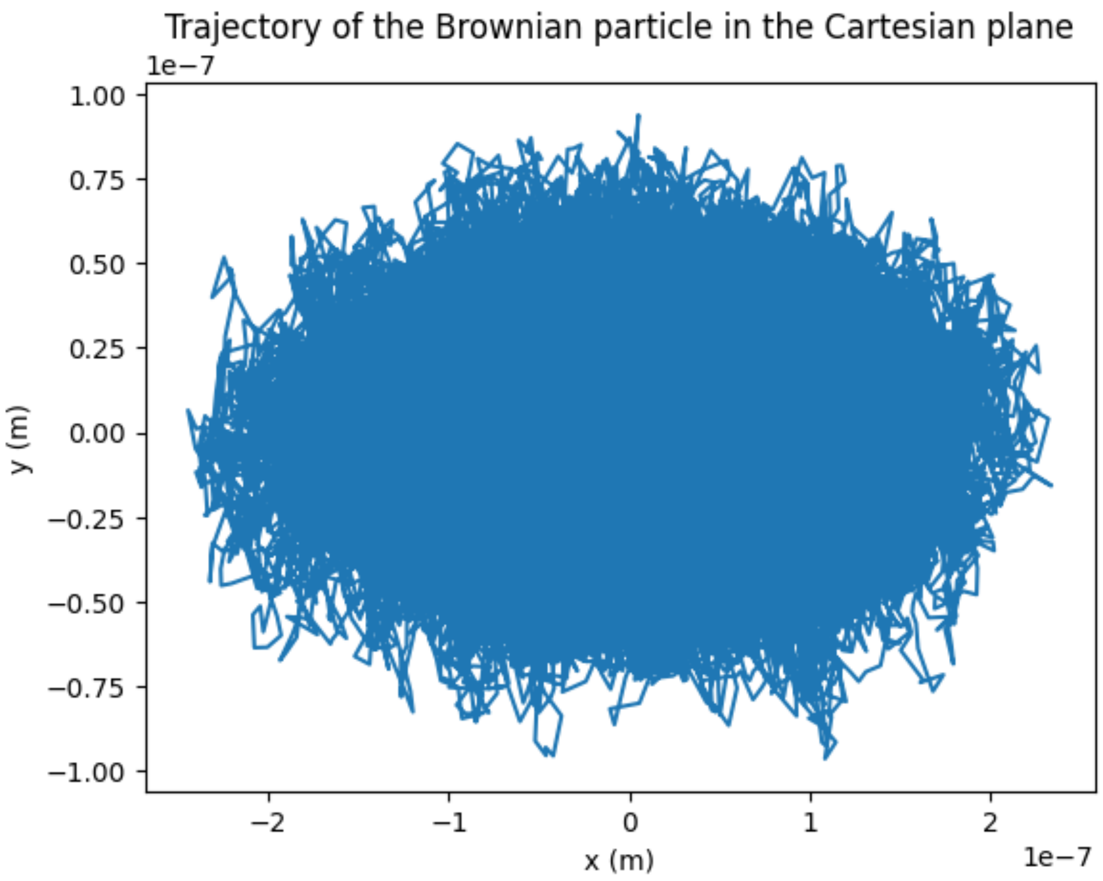
```
x di erent sti nesses  0.018849555921538764
y di erent sti nesses  0.0020943951023931952
timestep = 0.00010471975511965977
```

2 P1

Plot the trajectory of the disk in the Cartesian plane.

```
[21]: x = 0
y = 0

x_trajectory = [x]
```

```python
y_trajectory = [y]

t_total = 30
num_steps = int(t_total / dt)

for _ in range(num_steps):

    w_x = np.random.normal(0, 1)
    w_y = np.random.normal(0, 1)


    x = x - (k_x / gamma) * x * dt + np.sqrt(2 * (k_b * T / gamma) * dt) * w_x
    y = y - (k_y / gamma) * y * dt + np.sqrt(2 * (k_b * T / gamma) * dt) * w_y

    x_trajectory.append(x)
    y_trajectory.append(y)

plt.plot(x_trajectory, y_trajectory)
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Trajectory of the Brownian particle in the Cartesian plane')
plt.savefig('P1_trajactory.png')
plt.show()
```

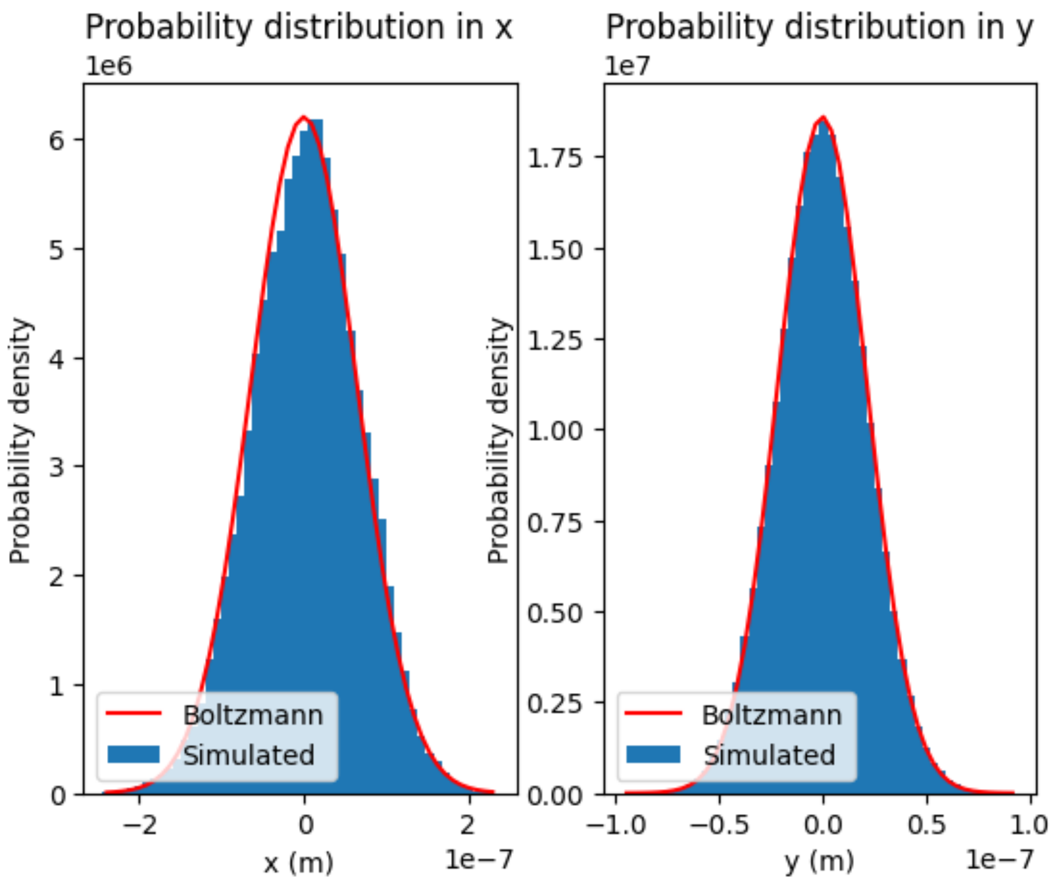


2.1 P2

Plot the probability distribution of the positions in xand in y (two separate histograms: one for xand one for y). Compare each case with the expected Boltzmann distribution

[25]:
```
# _hist : representing the probability density [use: density=True] of the␣
↪particle's x or y - positions within bins.
# _bin_edges: defining the boundaries of bins for calculating the x - position␣
↪probability distribution.

x_hist, x_bin_edges = np.histogram(x_trajectory, bins=50, density=True)
y_hist, y_bin_edges = np.histogram(y_trajectory, bins=50, density=True)

# Boltamann distibution - x axis
x_positions = (x_bin_edges[1:] + x_bin_edges[:-1]) / 2
U_x = 0.5 * k_x * x_positions ** 2
boltzmann_x = np.exp(-U_x / (k_b * T))
boltzmann_x /= np.sum(boltzmann_x) * (x_bin_edges[1] - x_bin_edges[0])
```

```python
# Boltamann distibution - y axis
y_positions = (y_bin_edges[1:] + y_bin_edges[:-1]) / 2
U_y = 0.5 * k_y * y_positions ** 2
boltzmann_y = np.exp(-U_y / (k_b * T))
boltzmann_y /= np.sum(boltzmann_y) * (y_bin_edges[1] - y_bin_edges[0])

# Plot x
plt.subplot(1, 2, 1)
plt.bar(x_positions, x_hist, width=(x_bin_edges[1] - x_bin_edges[0]),␣
 ↪label='Simulated')
plt.plot(x_positions, boltzmann_x, 'r', label='Boltzmann')
plt.xlabel('x (m)')
plt.ylabel('Probability density')
plt.title('Probability distribution in x')
plt.legend(loc='lower left')

# Plot y
plt.subplot(1, 2, 2)
plt.bar(y_positions, y_hist, width=(y_bin_edges[1] - y_bin_edges[0]),␣
 ↪label='Simulated')
plt.plot(y_positions, boltzmann_y, 'r', label='Boltzmann')
plt.xlabel('y (m)')
plt.ylabel('Probability density')
plt.title('Probability distribution in y')
plt.legend(loc='lower left')

plt.savefig('P2_Probability_distribution.png')
plt.show()
plt.pause(1)
plt.close()
```



[14]:
```
import numpy as np
sigma_x_squared = np.var(x_trajectory)
sigma_y_squared = np.var(y_trajectory)

print(f'sigma_x_squared = {sigma_x_squared}, sigma_y_squared =␣
 ↪{sigma_y_squared}')

if sigma_x_squared > sigma_y_squared:
    print("x variance is larger ")
elif sigma_y_squared > sigma_x_squared:
    print("y variance is larger")
else:
    print(" x and y variance is equal")

sigma_x_squared_theoretical = (k_b * T) / k_x
sigma_y_squared_theoretical = (k_b * T) / k_y
```

```
print(f'theorical variance in a harmonic trap, sigma_x_squared =␣
 ↪{sigma_x_squared_theoretical}, sigma_y_squared = ␣
 ↪{sigma_y_squared_theoretical}' )
```

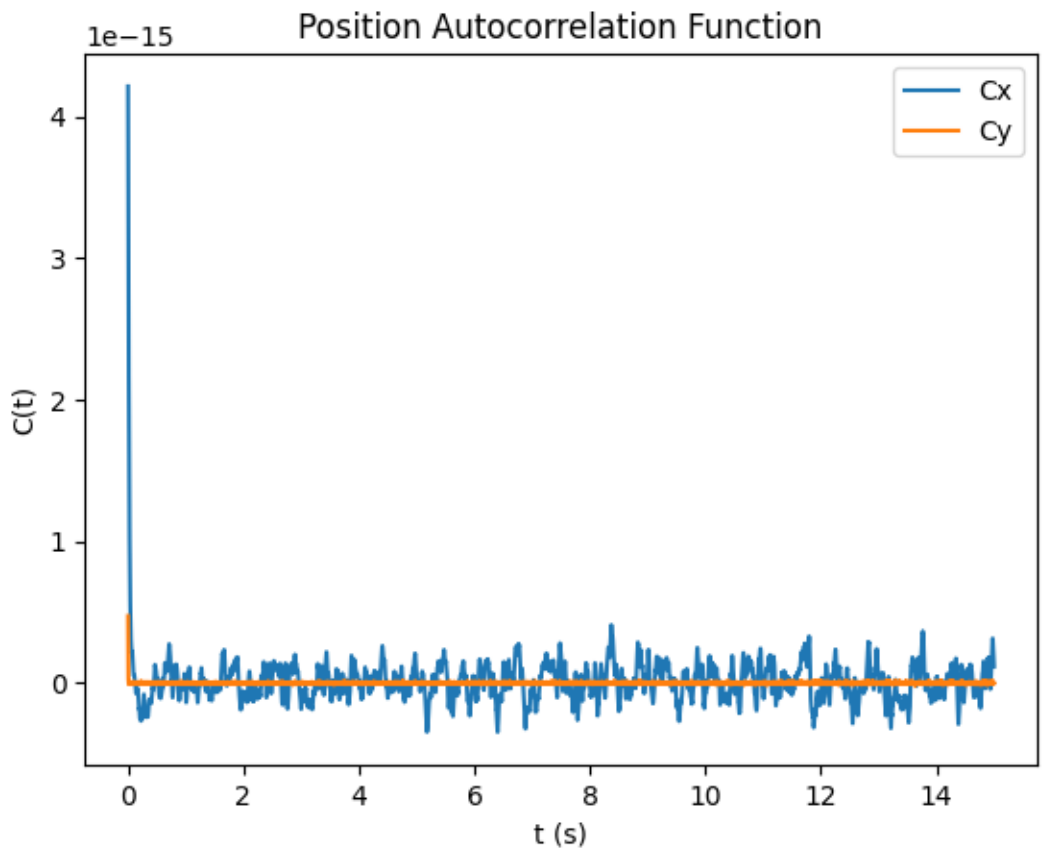
```
sigma_x_squared = 4.196631458265196e-15, sigma_y_squared = 4.607489519704521e-16
x variance is larger
theorical variance in a harmonic trap, sigma_x_squared = 4.1399999999999994e-15,
sigma_y_squared =  4.599999999999999e-16
```

2.2 P3 Calculate and plot the position autocorrelation function

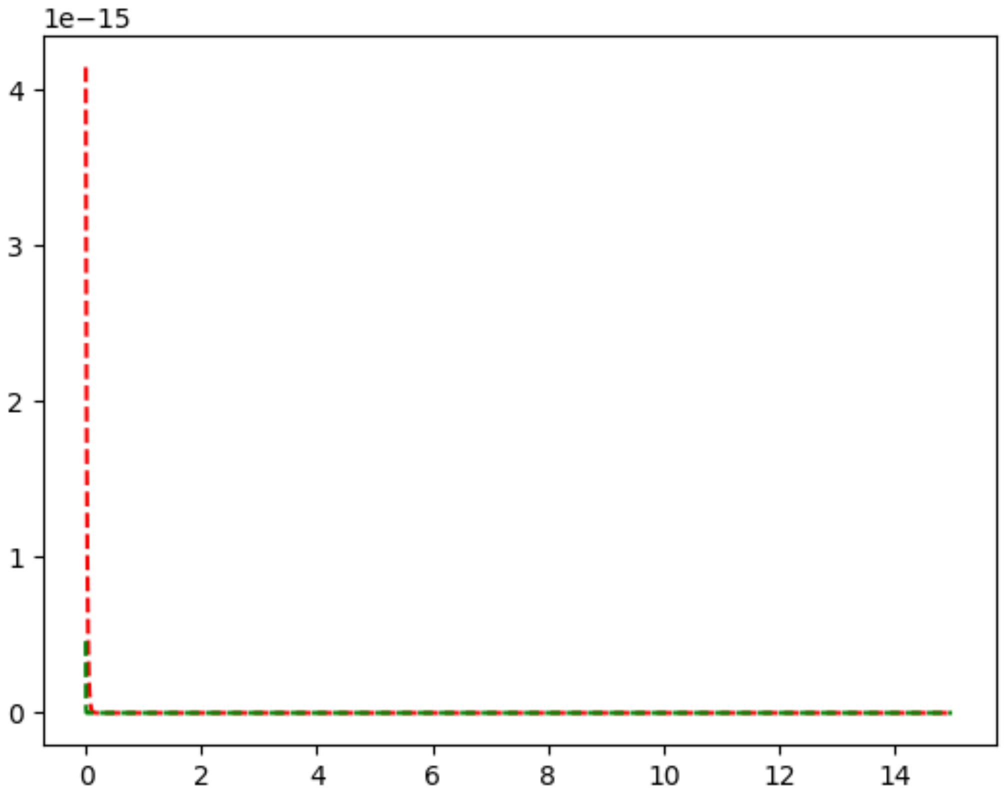
[23]:
```
def calculate_Cx(x_trajectory, n):
    x = np.array(x_trajectory)
    N = len(x)
    result = np.sum(x[n:N] * x[0:(N - n)])
    return (1 / (N - n)) * result


def calculate_Cy(y_trajectory, n):
    y = np.array(y_trajectory)
    N = len(y)
    result = np.sum(y[n:N] * y[0:(N - n)])
    return (1 / (N - n)) * result


Nx = len(x_trajectory)
Ny = len(y_trajectory)

print(f'Nx={Nx},Ny = {Ny}')
n_values = range(0, Nx // 2)

Cx_values = []
Cy_values = []

for n in n_values:

    Cx_value = calculate_Cx(x_trajectory, n)
    Cx_values.append(Cx_value)


    Cy_value = calculate_Cy(y_trajectory, n)
    Cy_values.append(Cy_value)
    #DEBUG
    if n % 5000 == 0:
        print(f'n = {n}')
```

```
plt.plot([n * dt for n in n_values], Cx_values, label='Cx')
plt.plot([n * dt for n in n_values], Cy_values, label='Cy')
plt.xlabel('t (s)')
plt.ylabel('C(t)')
plt.title('Position Autocorrelation Function')
plt.savefig('P3_position_autocorrelation_function .png')
plt.legend()
plt.show()
```

```
Nx=286479,Ny = 286479
n = 0
n = 5000
n = 10000
n = 15000
n = 20000
n = 25000
n = 30000
n = 35000
n = 40000
n = 45000
n = 50000
n = 55000
n = 60000
n = 65000
n = 70000
n = 75000
n = 80000
n = 85000
n = 90000
n = 95000
n = 100000
n = 105000
n = 110000
n = 115000
n = 120000
n = 125000
n = 130000
n = 135000
n = 140000
```



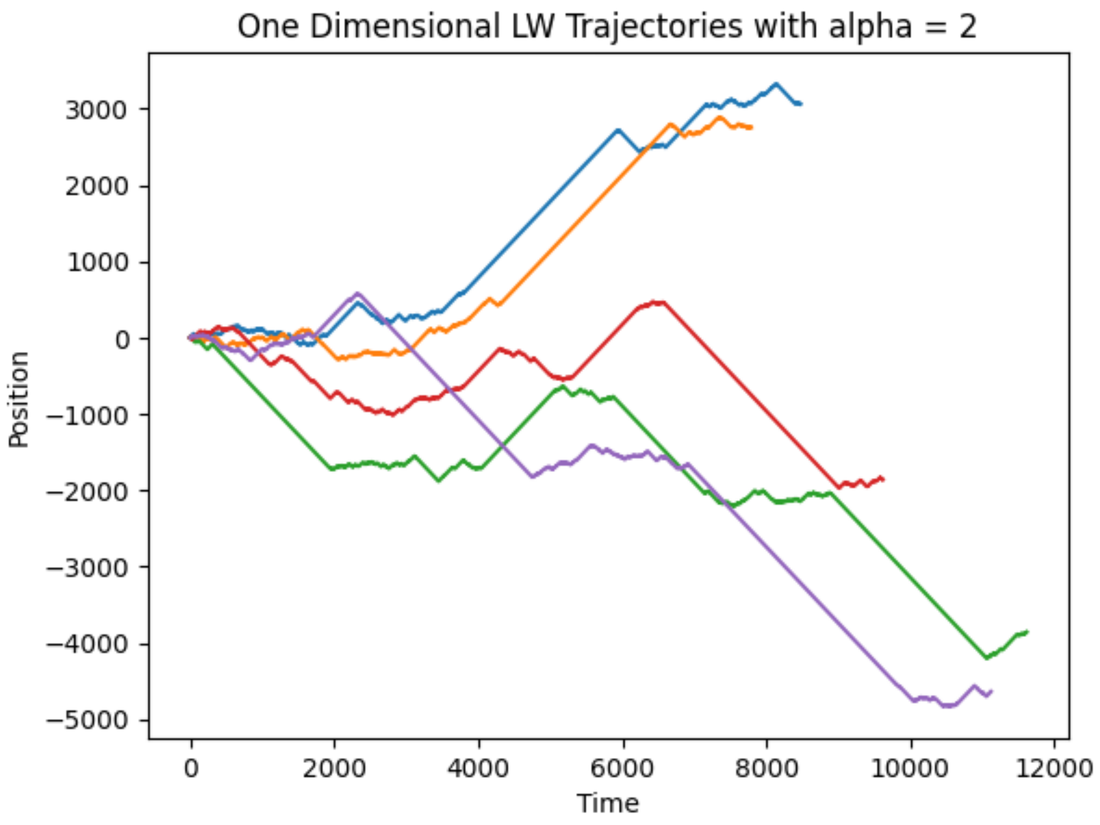
```
[24]: def theoretical_Cx(t, k_b, T, k_x, gamma):
          return (k_b * T) / k_x * np.exp(-k_x * t / gamma)

      def theoretical_Cy(t, k_b, T, k_y, gamma):
          return (k_b * T) / k_y * np.exp(-k_y * t / gamma)

      t_values = [n * dt for n in n_values]
      plt.plot(t_values, [theoretical_Cx(t, k_b, T, k_x, gamma) for t in t_values],␣
       ↪'r--', label='Theoretical Cx')
      plt.plot(t_values, [theoretical_Cy(t, k_b, T, k_y, gamma) for t in t_values],␣
       ↪'g--', label='Theoretical Cy')
      plt.savefig('P4_Theoretical_position_autocorrelation_function .png')
```


1e−15
4
3
2
1
0
0
2
4
6
8
10
12
14

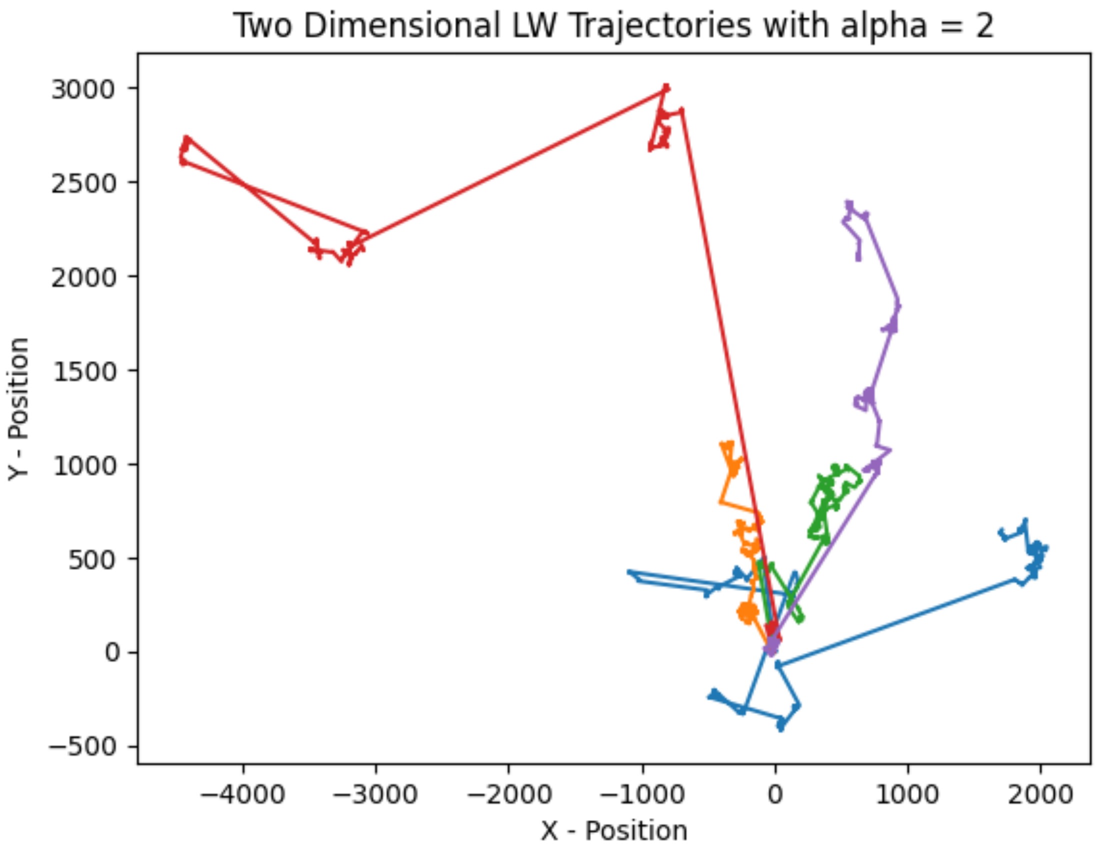
2.2

November 19, 2024

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt


     alpha = 2
     v = 1
     def generate_1D_trajectory():
         t = 0
         r = np.random.uniform(0, 1, size=1000)
         delta_t = r ** (-1 / (3 - alpha))
         w = np.random.choice([-1, 1], size=1000)
         x = np.zeros(1000)
         all_t = [t]
         for i in range(len(delta_t) - 1):
             x[i + 1] = x[i] + w[i] * v * delta_t[i]
             t += delta_t[i]
             all_t.append(t)
         return x, all_t


     for _ in range(5):
         x, t = generate_1D_trajectory()
         plt.plot(t, x)

     plt.xlabel('Time')
     plt.ylabel('Position')
     plt.title('One Dimensional LW Trajectories with alpha = 2')
     plt.savefig('One_Dimensional_LW_Trajectories.png')
     plt.show()
```



[20]:
```python
import numpy as np
import matplotlib.pyplot as plt


alpha = 2
v = 1


def generate_2D_trajectory():
    r = np.random.uniform(0, 1, size=1000)
    delta_t = r ** (-1 / (3 - alpha))

    w = np.random.uniform(-np.pi, np.pi, size=1000)
    phi = np.zeros(1000)
    x = np.zeros(1000)
    y = np.zeros(1000)
    all_t = [0]
    for i in range(len(delta_t) - 1):
        phi[i + 1] = phi[i] + w[i]
        x[i + 1] = x[i] + v * np.cos(phi[i]) * delta_t[i]
        y[i + 1] = y[i] + v * np.sin(phi[i]) * delta_t[i]
        all_t.append(all_t[-1] + delta_t[i])
    return x, y, all_t


for _ in range(5):
    x, y, t = generate_2D_trajectory()
    plt.plot(x, y)

plt.xlabel('X - Position')
plt.ylabel('Y - Position')
plt.title('Two Dimensional LW Trajectories with alpha = 2')
plt.savefig('Two_Dimensional_LW_Trajectories.png')
plt.show()
```



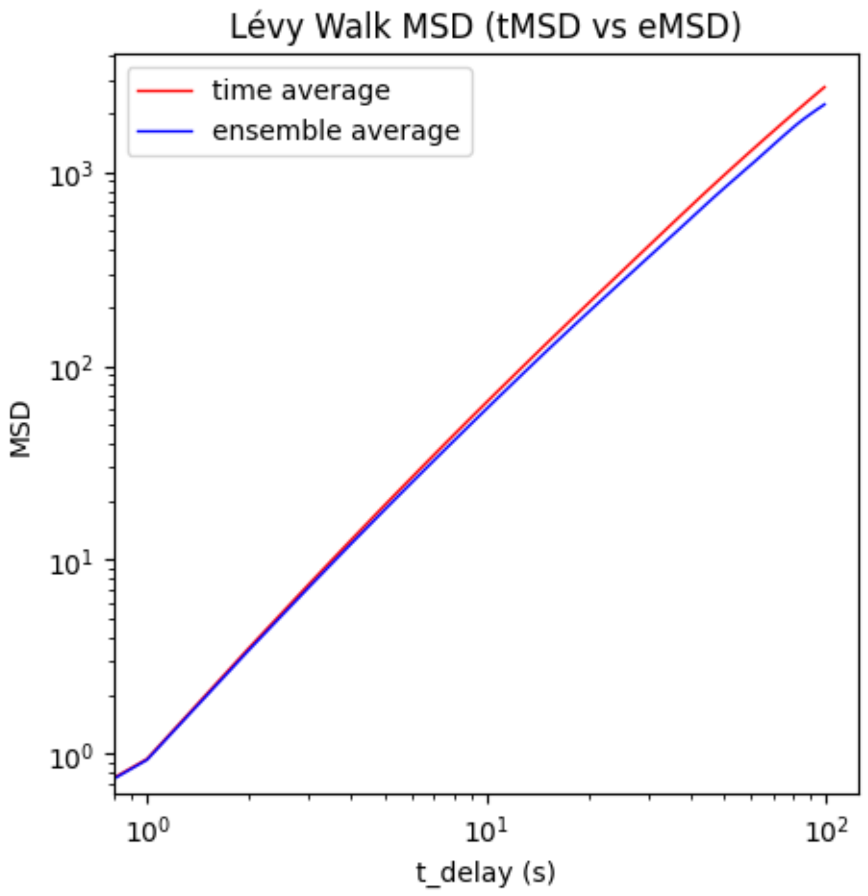
0.1 P3

plot the eMSD and tMSD for a 1-dimensional LW with = 2.

[251]:
```
import numpy as np
import matplotlib.pyplot as plt

#   1D Lévy Walk
def generate_1D_trajectory(N_steps, alpha=2, v=1):
    """
    Generate a 1D Lévy walk trajectory.

    Parameters
    ==========
    N_steps : Number of steps in the trajectory.
    alpha : Lévy index controlling the step distribution.
    v : Velocity of the walker.

    Returns
    =======
    x : Position of the walker at each step.
    all_t : Non-uniform time array.
    """
    t = 0
    r = np.random.uniform(0, 1, size=N_steps)  # Uniform random numbers
    delta_t = np.clip(r ** (-1 / (3 - alpha)), 1e-2, 1e2)  # Time intervals
    w = np.random.choice([-1, 1], size=N_steps)  # Random directions
    x = np.zeros(N_steps)  # Trajectory positions
    all_t = [t]  # Accumulated time array
    for i in range(len(delta_t) - 1):
        x[i + 1] = x[i] + w[i] * v * delta_t[i]
        t += delta_t[i]
        all_t.append(t)
    return x, np.array(all_t)

def regularize(x_nu, t_nu, t):
    """
    Regularize a time non-uniformly sampled trajectory.

    Parameters
    ==========
    x_nu : Trajectory (x component) non-uniformly sampled in time.
    t_nu : Time (non-uniform sampling).
    t : Time (wanted sampling).

    Returns
    =======
    x : Regularized trajectory.
    """
    x = np.zeros(np.size(t))
    m = np.diff(x_nu) / np.diff(t_nu)  # Slopes of the different increments.
```

```python
    s = 0  # Position in the wanted trajectory.
    for i in range(np.size(t_nu) - 1):
        s_end = np.where(t < t_nu[i + 1])[0][-1]
        x[s:s_end + 1] = x_nu[i] + m[i] * (t[s:s_end + 1] - t_nu[i])
        s = s_end + 1
    return x


def tMSD_1d(x, N_steps):
    """
    Compute the time-averaged mean squared displacement (tMSD).

    Parameters
    ==========
    x : Regularized trajectory.
    N_steps : Number of time steps.

    Returns
    =======
    tMSD : Time-averaged MSD.
    """
    tMSD = np.zeros(N_steps)
    for t in range(N_steps):
        displacements = x[t:] - x[:N_steps - t]
        tMSD[t] = np.mean(displacements**2)
    return tMSD


def eMSD_1d(x):
    """
    Compute the ensemble-averaged mean squared displacement (eMSD).

    Parameters
    ==========
    x : Ensemble of trajectories.

    Returns
    =======
    eMSD : Ensemble-averaged MSD.
    """
    N_traj, N_steps = x.shape
    eMSD = np.zeros(N_steps)
    for t in range(N_steps):
        displacements = x[:, t:] - x[:, :N_steps - t]
        eMSD[t] = np.mean(displacements**2)
    return eMSD
```

```python
alpha = 2  # Lévy index
v = 1  # Velocity of the walker

# Part I : tMSD
t_tot = 10000  # Total duration
N_steps = 10000  # Number of steps in trajectory
dt = 1  # Regularized time step

# Uniform sampling for time-averaging
t_t = np.arange(int(np.ceil(t_tot / dt))) * dt
N_steps_t = np.size(t_t)

# Generate a single Lévy Walk trajectory
x, t_nu = generate_1D_trajectory(N_steps, alpha=alpha, v=v)
x_t = regularize(x, t_nu, t_t)  # Regularize trajectory to uniform time grid

# Calculate tMSD
tmsd = tMSD_1d(x_t, N_steps_t)

# Part II: eMSD
t_tot = 100  # Total duration for each trajectory
N_steps = 1000  # Number of steps in trajectory
dt = 1  # Regularized time step

# Uniform sampling for ensemble-averaging
t_e = np.arange(int(np.ceil(t_tot / dt))) * dt
N_steps_e = np.size(t_e) # infact = t_tot / dt
N_traj = 100  # Number of trajectories

# Generate trajectories
x_e = np.zeros([N_traj, N_steps_e])
for i in range(N_traj):
    x, t_nu = generate_1D_trajectory(N_steps, alpha=alpha, v=v)
    x_r = regularize(x, t_nu, t_e)  # Regularize trajectory
    x_e[i, :] = x_r

# Calculate eMSD
emsd = eMSD_1d(x_e)


plt.figure(figsize=(5, 5))
plt.loglog(t_e, tmsd[:len(t_e)], '-', color='r', linewidth=1, label='time␣
 ↪average')
plt.loglog(t_e, emsd, '-', color='b', linewidth= 1, label='ensemble average')
plt.legend()
plt.xlabel('t_delay (s)')
plt.ylabel('MSD')
plt.title('Lévy Walk MSD (tMSD vs eMSD)')
plt.savefig('tMSD_vs_eMSD.png')
plt.show()
```

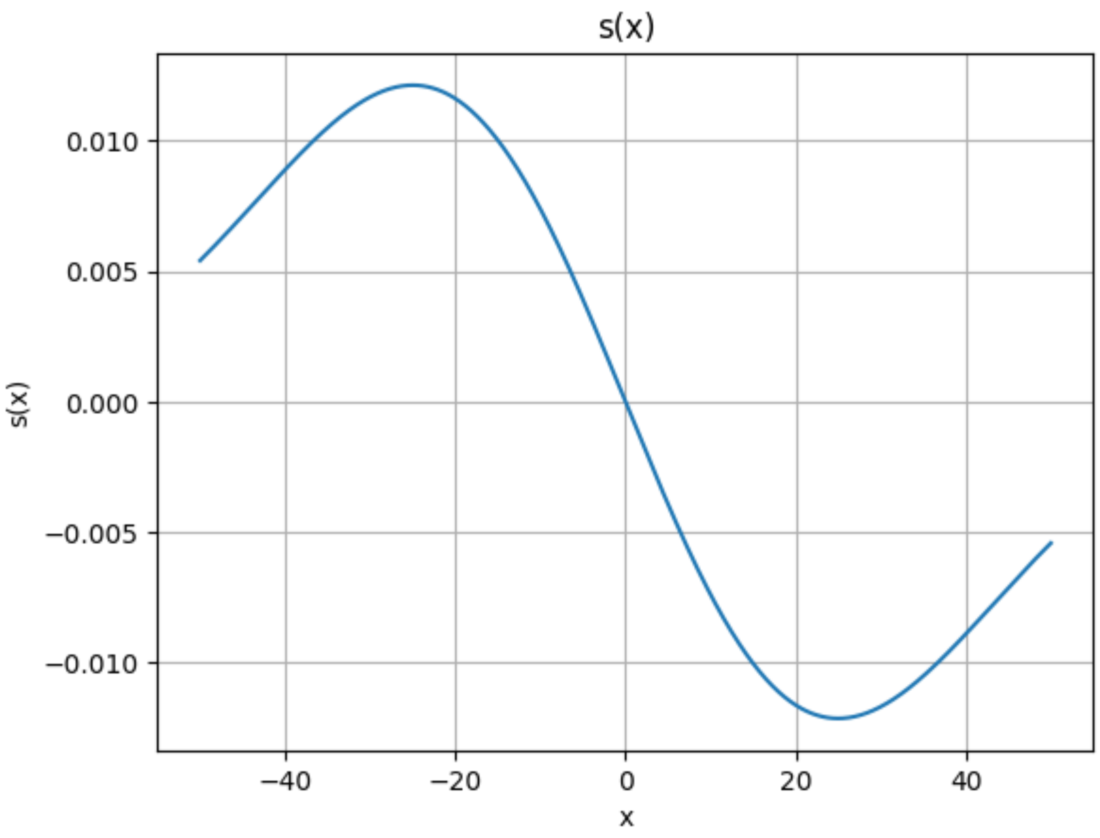


2.3_version2

November 18, 2024

0.1 P1

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

sigma0 = 1
w = 25
L = 100
times = [1, 5, 10, 25, 50, 100]


def s(x):
    return - (x / (w**2)) * (sigma0**2 / 2) * np.exp(-(x**2) / (2 * w**2))

x = np.linspace(-L/2, L/2, 1000)
s_x = s(x)

plt.plot(x, s_x)
plt.xlabel('x')
plt.ylabel('s(x)')
plt.title('s(x)')
plt.grid(True)
plt.savefig('P1.png')
plt.show()
```



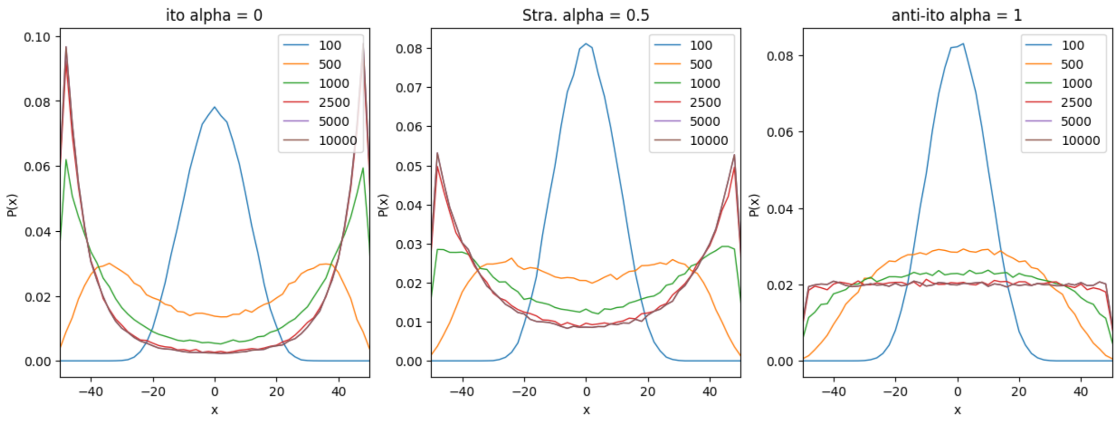
0.2 P2 P3 P4

Use different integral convention

alpha = 0: ito alpha = 0.5: Stratonovich convention alpha = 1: anti-ito

[1]:
```
import numpy as np
import matplotlib.pyplot as plt


alphas = [0, 0.5, 1]
alpha_labels = ['ito alpha = 0', 'Stra. alpha = 0.5', 'anti-ito alpha = 1']


dt = 1
N_traj = 100000
t0 = 100
j_mult = np.array([1, 5, 10, 25, 50, 100])
x0 = 0
L = 100
x_min = -L / 2
```

```python
x_max = L / 2
sigma0 = 1
w = 25


def sigma(x):
    return sigma0 * np.exp(-x ** 2 / (2 * w ** 2))


def dsigma_dx(x):
    return - (x / (w ** 2)) * sigma0 * np.exp(-x ** 2 / (2 * w ** 2))


#    alpha
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

for k, alpha in enumerate(alphas):
    x_fin = np.zeros([N_traj, np.size(j_mult)])
    for j in range(np.size(j_mult)):
        n_t0 = j_mult[j] * t0  #
        N_steps = int(np.ceil(n_t0 / dt))

        rn = np.random.normal(0, 1, size=(N_traj, N_steps))

        if j > 1:
            x = x_fin[:, j - 1]
        else:
            x = np.zeros(N_traj)

        for step in range(N_steps):
            sigma_x = sigma(x)
            d_sigma_x = dsigma_dx(x)
            dx_spurious = alpha * sigma_x * d_sigma_x * dt
            x += dx_spurious + sigma_x * rn[:, step]
            bounce_left = np.where(x < x_min)[0]
            x[bounce_left] = 2 * x_min - x[bounce_left]
            bounce_right = np.where(x > x_max)[0]
            x[bounce_right] = 2 * x_max - x[bounce_right]

        x_fin[:, j] = x

    bin_width = 2
    bins_edges = np.arange(-L - bin_width / 2, L + bin_width / 2 +.1, bin_width)
    bins = np.arange(-L, L +.1, bin_width)
    p_distr = np.zeros([np.size(bins), np.size(j_mult)])
    for j in range(np.size(j_mult)):
        distribution = np.histogram(x_fin[:, j], bins=bins_edges)
        p_distr[:, j] = distribution[0] / np.sum(distribution[0])

    for j in range(np.size(j_mult)):
        axs[k].plot(bins, p_distr[:, j], '-', linewidth=1, label=str(j_mult[j]␣
↪* t0))
    axs[k].set_title(alpha_labels[k])
    axs[k].set_xlabel('x')
    axs[k].set_ylabel('P(x)')
    axs[k].set_xlim([x_min, x_max])
    axs[k].legend()

plt.savefig('P2_3_4.png')
plt.show()
```



2.4

November 18, 2024

0.1 Task 1

[ ]:
```python
import math
import numpy as np
from matplotlib import pyplot as plt


def replicas(x, y, L):
    """
    Function to generate replicas of a single particle.

    Parameters
    ==========
    x, y : Position.
    L : Side of the squared arena.
    """
    xr = np.zeros(9)
    yr = np.zeros(9)

    for i in range(3):
        for j in range(3):
            xr[3 * i + j] = x + (j - 1) * L
            yr[3 * i + j] = y + (i - 1) * L

    return xr, yr

def pbc(x, y, L):
    """
    Function to enforce periodic boundary conditions on the positions.

    Parameters
    ==========
    x, y : Position.
    L : Side of the squared arena.
    """

    outside_left = np.where(x < - L / 2)[0]
    x[outside_left] = x[outside_left] + L
    outside_right = np.where(x > L / 2)[0]
    x[outside_right] = x[outside_right] - L

    outside_up = np.where(y > L / 2)[0]
    y[outside_up] = y[outside_up] - L

    outside_down = np.where(y < - L / 2)[0]
    y[outside_down] = y[outside_down] + L

    return x, y

from functools import reduce

def interaction(x, y, theta, Rf, L):
    """
    Function to calculate the orientation at the next time step.

    Parameters
    ==========
    x, y : Positions.
    theta : Orientations.
    Rf : Flocking radius.
    L : Dimension of the squared arena.
    s : Discrete steps.
    """

    N = np.size(x)

    theta_next = np.zeros(N)

    # Preselect what particles are closer than Rf to the boundaries.
    replicas_needed = reduce(
        np.union1d, (
            np.where(y + Rf > L / 2)[0],
            np.where(y - Rf < - L / 2)[0],
            np.where(x + Rf > L / 2)[0],
            np.where(x - Rf > - L / 2)[0]
        )
    )

    for j in range(N):
        # Check if replicas are needed to find the nearest neighbours.
        if np.size(np.where(replicas_needed == j)[0]):
            # Use replicas.
            xr, yr = replicas(x[j], y[j], L)
            nn = []
            for nr in range(9):
                dist2 = (x - xr[nr]) ** 2 + (y - yr[nr]) ** 2
                nn = np.union1d(nn, np.where(dist2 <= Rf ** 2)[0])
        else:
            dist2 = (x - x[j]) ** 2 + (y - y[j]) ** 2
            nn = np.where(dist2 <= Rf ** 2)[0]

        # The list of nearest neighbours is set.
        nn = nn.astype(int)

        # Circular average.
        av_sin_theta = np.mean(np.sin(theta[nn]))
        av_cos_theta = np.mean(np.cos(theta[nn]))

        theta_next[j] = np.arctan2(av_sin_theta, av_cos_theta)

    return theta_next



def global_alignment(theta):
    """
    Function to calculate the global alignment coefficient.

    Parameters
    ==========
    theta : Orientations.
    """

    N = np.size(theta)

    global_direction_x = np.sum(np.sin(theta))
    global_direction_y = np.sum(np.cos(theta))

    psi = np.sqrt(global_direction_x ** 2 + global_direction_y ** 2) / N

    return psi

from scipy.spatial import Voronoi, voronoi_plot_2d

def area_polygon(vertices):
    """
    Function to calculate the area of a Voronoi region given its vertices.

    Parameters
    ==========
    vertices : Coordinates (array, 2 dimensional).
    """
    N, dim = vertices.shape

    # dim 2
    A = 0
    for i in range(N - 1):
        A += np.abs(
            vertices[-1, 0] * (vertices[i, 1] - vertices[i + 1, 1]) +
            vertices[i, 0] * (vertices[i + 1, 1] - vertices[-1, 1]) +
            vertices[i + 1, 0] * (vertices[-1, 1] - vertices[i, 1])
        )
    A *= 0.5
    return A

def global_clustering(x, y, Rf, L):
    """
    Function to calculate the global alignment coefficient.

    Parameters
    ==========
    x, y : Positions.
    Rf : Flocking radius.
    L : Dimension of the squared arena.
    """

    N = np.size(x)

    # Use the replicas of all points to calculate Voronoi for
    # a more precise estimate.
    points = np.zeros([9 * N, 2])

    for i in range(3):
        for j in range(3):
            s = 3 * i + j
            points[s * N:(s + 1) * N, 0] = x + (j - 1) * L
            points[s * N:(s + 1) * N, 1] = y + (i - 1) * L

    # The format of points is the one needed by Voronoi.
    # points[:, 0] contains the x coordinates
    # points[:, 1] contains the y coordinates

    vor = Voronoi(points)
    '''
    vertices = vor.vertices  # Voronoi vertices.
    regions = vor.regions  # Region list.
    # regions[i]: list of the vertices indices for region i.
    # If -1 is listed: the region is open (includes point at infinity).
    point_region = vor.point_region  # Region associated to input point.
    '''

    # Consider only regions of original set of points (no replicas).
    list_regions = vor.point_region[4 * N:5 * N]

    c = 0

    for i in list_regions:
        indices = vor.regions[i]
        # print(f'indices = {indices}')
        if len(indices) > 0:
            if np.size(np.where(np.array(indices) == -1)[0]) == 0:
                # Region is finite.
                # Calculate area.
                A = area_polygon(vor.vertices[indices,:])
                if A < np.pi * Rf ** 2:
                    c += 1

    c = c / N

    return c



N = 200  # Number of particles.
L = 100  # Dimension of the squared arena.
v = 1  # Speed.
Rf = 2  # Flocking radius.
eta = 0.01  # Noise.  Try values: 0.01, 0.3, 1.0, 2 * np.pi
dt = 1  # Time step.
T = 6000 # total time steps

# Random position.
x = (np.random.rand(N) - 0.5) * L  # in [-L/2, L/2]
y = (np.random.rand(N) - 0.5) * L  # in [-L/2, L/2]

# Random orientation.
theta = 2 * (np.random.rand(N) - 0.5) * np.pi  # in [-pi, pi]

time_steps = []
psi = np.zeros(T+1)  # Records the global alignment.
c = np.zeros(T+1)  # Records the global clustering.

fig, ax = plt.subplots(figsize=(10, 10))

for step in range(T + 1):
    x_center = np.mean(x)
    y_center = np.mean(y)

    # Check whether plot configuration.
    if step in [0, 2000, 4000, 6000]:
        ax.clear()  # Clear previous plot.
        ax.plot(x, y, '.', markersize=16)
        ax.quiver(x, y, np.cos(theta), np.sin(theta))
        ax.plot(x_center +Rf * np.cos(2 * np.pi * np.arange(360) / 360),
                y_center + Rf * np.sin(2 * np.pi * np.arange(360) / 360),
                '-', color='#FFA0FF', linewidth=3)
        ax.set_xlim([-L / 2, L / 2])
        ax.set_ylim([-L / 2, L / 2])
        ax.set_title(f'Step {step}')

        plt.savefig(f'particle_figure_timestep_{step}.png')

    # DEBUG
    if step % 500 == 0:
        print(f'step is {step}')

    psi[step] = global_alignment(theta)
    c[step] = global_clustering(x, y, Rf, L)
    time_steps.append(step)

    # update velocity and postion
    dtheta = eta * (np.random.rand(N) - 0.5) * dt
    theta = interaction(x, y, theta, Rf, L) + dtheta
    x = x + v * np.cos(theta)
    y = y + v * np.sin(theta)
    x, y = pbc(x, y, L)

# DEBUG
print(len(psi))
print(psi)
print(len(c))
print(c)

plt.figure(figsize=(10, 5))
plt.plot(psi, '-', linewidth=1, label='alignment')
plt.plot(c, '-', linewidth=1, label='clustering')
plt.plot(0 * psi, '--', color='k', linewidth=0.5)
plt.plot(0 * psi + 1, '--', color='k', linewidth=0.5)
plt.title('Global alignment coefficient')
plt.legend()
plt.xlabel('step')
plt.ylabel('psi')
plt.ylim([-0.1, 1.1])
plt.savefig(f'Global alignment and clustering coefficient.png')
```

0.2 Task 2

[ ]:
```
import math
import numpy as np
from matplotlib import pyplot as plt


def replicas(x, y, L):
    """
    Function to generate replicas of a single particle.

    Parameters
    ==========
    x, y : Position.
    L : Side of the squared arena.
    """
    xr = np.zeros(9)
    yr = np.zeros(9)

    for i in range(3):
        for j in range(3):
            xr[3 * i + j] = x + (j - 1) * L
            yr[3 * i + j] = y + (i - 1) * L

    return xr, yr


def pbc(x, y, L):
    """
    Function to enforce periodic boundary conditions on the positions.

    Parameters
    ==========
    x, y : Position.
    L : Side of the squared arena.
    """

    outside_left = np.where(x < -L / 2)[0]
    x[outside_left] = x[outside_left] + L

    outside_right = np.where(x > L / 2)[0]
    x[outside_right] = x[outside_right] - L
    outside_up = np.where(y > L / 2)[0]
    y[outside_up] = y[outside_up] - L

    outside_down = np.where(y < -L / 2)[0]
    y[outside_down] = y[outside_down] + L

    return x, y


def interaction(x, y, theta, Rf, L):
    """
    Function to calculate the orientation at the next time step.

    Parameters
    ==========
    x, y : Positions.
    theta : Orientations.
    Rf : Flocking radius.
    L : Dimension of the squared arena.
    s : Discrete steps.
    """

    N = np.size(x)
    theta_next = np.zeros(N)

    # Preselect what particles are closer than Rf to the boundaries.
    replicas_needed = np.unique(np.concatenate((
        np.where(y + Rf > L / 2)[0],
        np.where(y - Rf < -L / 2)[0],
        np.where(x + Rf > L / 2)[0],
        np.where(x - Rf > -L / 2)[0]
    )))

    for j in range(N):
        # Check if replicas are needed to find the nearest neighbours.
        if np.size(np.where(replicas_needed == j)[0]) > 0:
            # Use replicas.
            xr, yr = replicas(x[j], y[j], L)
            nn = []
            for nr in range(9):
                dist2 = (x - xr[nr]) ** 2 + (y - yr[nr]) ** 2
                nn = np.unique(np.concatenate((nn, np.where(dist2 <= Rf **␣
 ↪2)[0])))
        else:
            dist2 = (x - x[j]) ** 2 + (y - y[j]) ** 2
            nn = np.where(dist2 <= Rf ** 2)[0]
        # The list of nearest neighbours is set.
        nn = nn.astype(int)

        # Circular average.
        av_sin_theta = np.mean(np.sin(theta[nn]))
        av_cos_theta = np.mean(np.cos(theta[nn]))

        theta_next[j] = np.arctan2(av_sin_theta, av_cos_theta)

    return theta_next


def global_alignment(theta):
    """
    Function to calculate the global alignment coefficient.

    Parameters
    ==========
    theta : Orientations.
    """

    N = np.size(theta)

    global_direction_x = np.sum(np.sin(theta))
    global_direction_y = np.sum(np.cos(theta))

    psi = np.sqrt(global_direction_x ** 2 + global_direction_y ** 2) / N

    return psi


from scipy.spatial import Voronoi


def area_polygon(vertices):
    """
    Function to calculate the area of a Voronoi region given its vertices.

    Parameters
    ==========
    vertices : Coordinates (array, 2 dimensional).
    """
    N, dim = vertices.shape

    # dim 2
    A = 0
    for i in range(N - 1):
        A += np.abs(
            vertices[-1, 0] * (vertices[i, 1] - vertices[i + 1, 1]) +
            vertices[i, 0] * (vertices[i + 1, 1] - vertices[-1, 1]) +
            vertices[i + 1, 0] * (vertices[-1, 1] - vertices[i, 1])
        )
    A *= 0.5
    return A


def global_clustering(x, y, Rf, L):
    """
    Function to calculate the global alignment coefficient.

    Parameters
    ==========
    x, y : Positions.
    Rf : Flocking radius.
    L : Dimension of the squared arena.
    """

    N = np.size(x)

    # Use the replicas of all points to calculate Voronoi for
    # a more precise estimate.
    points = np.zeros([9 * N, 2])

    for i in range(3):
        for j in range(3):
            s = 3 * i + j
            points[s * N:(s + 1) * N, 0] = x + (j - 1) * L
            points[s * N:(s + 1) * N, 1] = y + (i - 1) * L

    vor = Voronoi(points)

    # Consider only regions of original set of points (no replicas).
    list_regions = vor.point_region[4 * N:5 * N]

    c = 0
    for i in list_regions:
        indices = vor.regions[i]
        if len(indices) > 0:
            if np.size(np.where(np.array(indices) == -1)[0]) == 0:
                # Region is finite.
                # Calculate area.
                A = area_polygon(vor.vertices[indices, :])
                if A < np.pi * Rf ** 2:
                    c += 1
    c = c / N
    return c


N = 200  # Number of particles.
L = 100  # Dimension of the squared arena.
v = 1  # Speed.
Rf = 2  # Flocking radius.
eta1 = 0.01  # Noise for first sub - population
eta2 = 0.3  # Noise for second sub - population
dt = 1  # Time step.
T = 6000  # total time steps

# Random position.
x = (np.random.rand(N) - 0.5) * L  # in [-L/2, L/2]
y = (np.random.rand(N) - 0.5) * L  # in [-L/2, L/2]

# Random orientation.
theta = 2 * (np.random.rand(N) - 0.5) * np.pi  # in [-pi, pi]

time_steps = []
psi = np.zeros(T + 1)  # Records the global alignment.
c = np.zeros(T + 1)  # Records the global clustering.

fig, ax = plt.subplots(figsize=(10, 10))

for step in range(T + 1):
    x_center = np.mean(x)
    y_center = np.mean(y)

    # Check whether plot configuration.
    if step in [0, 2000, 4000, 6000]:
        ax.clear()
        # Plot first sub - population (blue)
        ax.plot(x[0:100], y[0:100], '.', markersize=16, color='blue')
        ax.quiver(x[0:100], y[0:100], np.cos(theta[0:100]), np.sin(theta[0:
 ↪100]))
        # Plot second sub - population (red)
        ax.plot(x[100:200], y[100:200], '.', markersize=16, color='red')
        ax.quiver(x[100:200], y[100:200], np.cos(theta[100:200]), np.
 ↪sin(theta[100:200]))
        ax.plot(x_center + Rf * np.cos(2 * np.pi * np.arange(360) / 360),
                y_center + Rf * np.sin(2 * np.pi * np.arange(360) / 360),
                '-', color='#FFA0FF', linewidth=3)
        ax.set_xlim([-L / 2, L / 2])
        ax.set_ylim([-L / 2, L / 2])
        ax.set_title(f'Step {step}')
        plt.savefig(f'Task2_particle_figure_timestep_{step}.png')

    # DEBUG
    if step % 500 == 0:
        print(f'step is {step}')

    psi[step] = global_alignment(theta)
    c[step] = global_clustering(x, y, Rf, L)
    time_steps.append(step)

    # update velocity and position
    dtheta1 = eta1 * (np.random.rand(100) - 0.5) * dt
    dtheta2 = eta2 * (np.random.rand(100) - 0.5) * dt
    dtheta = np.concatenate((dtheta1, dtheta2))
    theta = interaction(x, y, theta, Rf, L) + dtheta
    x = x + v * np.cos(theta)
    y = y + v * np.sin(theta)
    x, y = pbc(x, y, L)

# DEBUG
print(len(psi))
print(psi)
print(len(c))
print(c)

plt.figure(figsize=(10, 5))
plt.plot(psi, '-', linewidth=1, label='alignment')
plt.plot(c, '-', linewidth=1, label='clustering')
plt.plot(0 * psi, '--', color='k', linewidth=0.5)
plt.plot(0 * psi + 1, '--', color='k', linewidth=0.5)
plt.title('Task2 Global alignment coefficient')
plt.legend(loc='lower right')
plt.xlabel('step')
plt.ylabel('psi')
plt.ylim([-0.1, 1.1])
plt.savefig(f'Task2 Global alignment and clustering coefficient.png')
```