Q1

In [1]:
```
import numpy as np

k_B = 1.380*10**-23
T= 300
eta = 10**-3
R=10**-6
k_x = 10**-6
k_y = 9*10**-6

gamma = 6*np.pi*eta*R

D = (k_B*T)/gamma

##Q1
tau_trap_x = gamma/k_x
tau_trap_y = gamma/k_y

print("tau_trap_x:", tau_trap_x,"\ntau_trap_y:", tau_trap_y)
```

```
tau_trap_x: 0.018849555921538756
tau_trap_y: 0.0020943951023931952
```

- τ_{trap} is the relaxation time, and choosing dt much smaller than the smallest relaxation time for x and y ensures that the simulation captures the fastest dynamics in both directions.

  To be on the safe side, i also divide the smallest τ_{trap} by 10, such that it is "much smaller".

- Since $k_y > k_x$ the restoring force in the y-direction is stronger, meaning the particle relaxes faster (smaller τ_{trap}) in the y-direction.

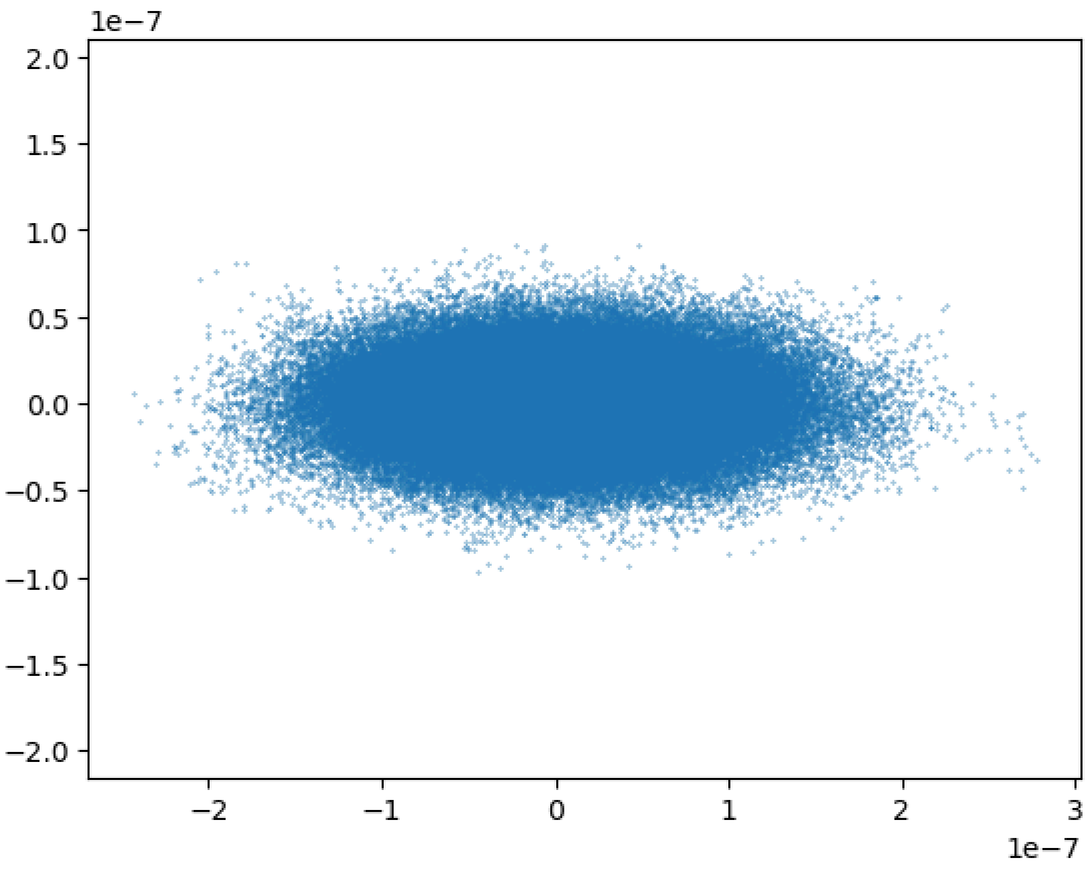
In [2]:
```
min_tau = np.min((tau_trap_x,tau_trap_y))
dt = min_tau/10
print("dt=",dt)
```

```
dt= 0.00020943951023931953
```

P1

In [3]:
```
import matplotlib.pyplot as plt
t_tot = 30
N= int(t_tot/dt)

x = np.zeros(N)
y = np.zeros(N)
w_x=np.random.normal(0,1,N)
w_y=np.random.normal(0,1,N)
for i in range(N-1):
    x[i+1] = x[i] - k_x*x[i]*dt/gamma + np.sqrt(2*k_B*T*dt/gamma)*w_x[i]
    y[i+1] = y[i] - k_y*y[i]*dt/gamma + np.sqrt(2*k_B*T*dt/gamma)*w_y[i]
```

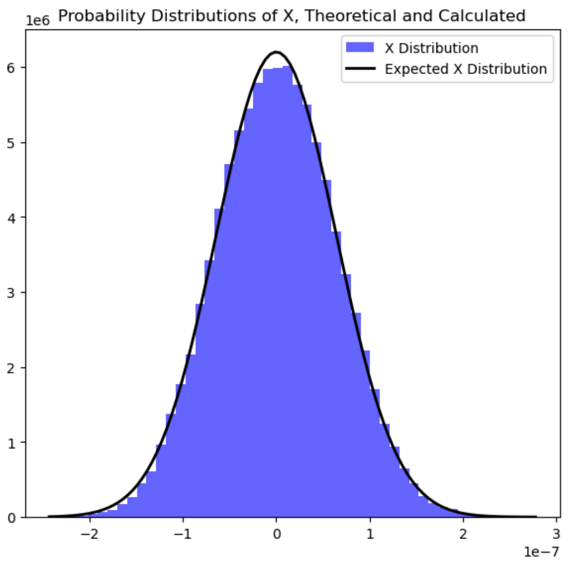
```
### P1
##scale to nm
plt.plot(x,y,'.',markersize=0.6)
plt.axis('equal')
plt.show()
```

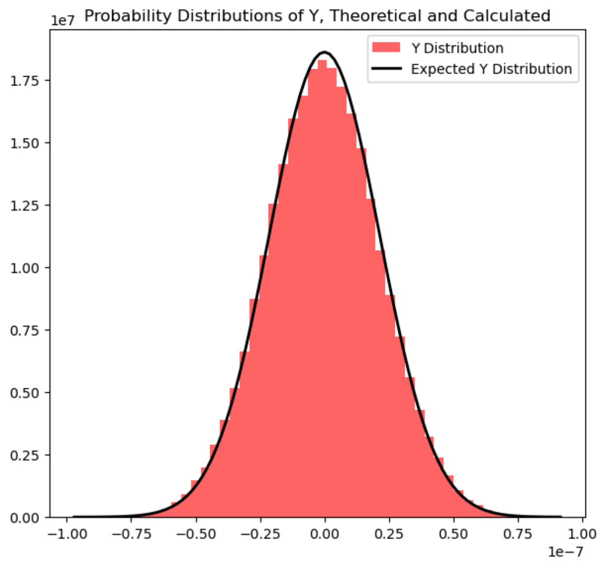


P2

In [4]:
```
#### P2
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Plot for X distribution and theoretical distribution
axes[0].hist(x, bins=50, density=True, alpha=0.6, color='blue', label='X Distrib
x_generated = np.linspace(np.min(x), np.max(x), 100)
U_x = 0.5 * k_x * x_generated**2
p_x = np.exp(-U_x / (k_B * T))
p_x /= np.sum(p_x) * (x_generated[1] - x_generated[0])  # Normalize
axes[0].plot(x_generated, p_x, color='black', label='Expected X Distribution', l
axes[0].set_title('Probability Distributions of X, Theoretical and Calculated')
axes[0].legend()

# Plot for Y distribution and theoretical distribution
axes[1].hist(y, bins=50, density=True, alpha=0.6, color='red', label='Y Distribu
y_generated = np.linspace(np.min(y), np.max(y), 100)
U_y = 0.5 * k_y * y_generated**2
p_y = np.exp(-U_y / (k_B * T))
p_y /= np.sum(p_y) * (y_generated[1] - y_generated[0])  # Normalize
axes[1].plot(y_generated, p_y, color='black', label='Expected Y Distribution', l
axes[1].set_title('Probability Distributions of Y, Theoretical and Calculated')
```

```
axes[1].legend()

plt.tight_layout()
plt.show()
```





Q2

In [5]:
```
sigma_x = np.var(x)
sigma_y = np.var(y)

harmonic_trap_x = k_B*T/k_x
harmonic_trap_y = k_B*T/k_y

print("sigma_x:",sigma_x, "vs Harmonic_trap_x:", harmonic_trap_x)
print("sigma_y:",sigma_y, "vs Harmonic_trap_y:", harmonic_trap_y)
```

```
sigma_x: 4.076958049174085e-15 vs Harmonic_trap_x: 4.1399999999999994e-15
sigma_y: 4.794308002801998e-16 vs Harmonic_trap_y: 4.599999999999999e-16
```

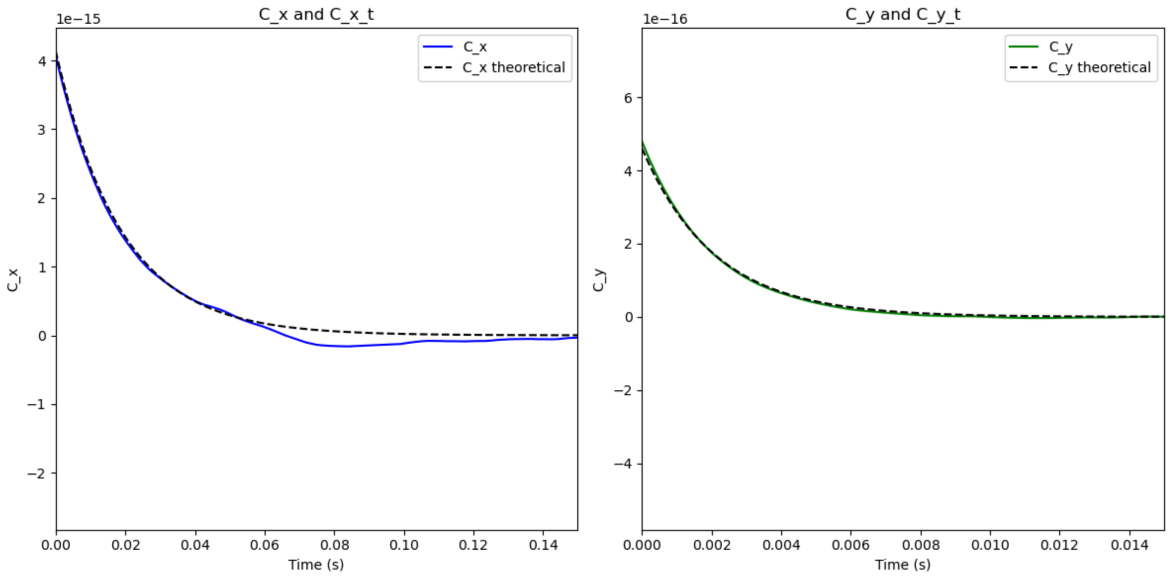
x has the larger variance because y has higher stiffness and therefore wont move as much as x, thus the variance is smaller for y than x.

The real and theoretical values seem very similar.

P3

In [6]:
```
C_x = np.zeros(N)
for n in range(N):
    frac = 1/(N-n)
    C_x[n] = frac* np.sum(x[n:]*x[:N-n])

C_y = np.zeros(N)
for n in range(0,N):
    frac = 1/(N-n)
    C_y[n] = frac* np.sum(y[n:]*y[:N-n])


t = np.arange(0,N)*dt
C_x_t = (k_B * T / k_x) * np.exp(-k_x * t / gamma)
```

```
C_y_t = (k_B * T / k_y) * np.exp(-k_y * t / gamma)


# Create the figure and subplots
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Plot C_x and C_x_t on the first subplot
axes[0].plot(t, C_x, color="blue", label="C_x")
axes[0].plot(t, C_x_t, "k--", label="C_x theoretical")  # Black dashed line for
axes[0].set_xlabel("Time (s)")
axes[0].set_ylabel("C_x")
axes[0].legend()
axes[0].set_title("C_x and C_x_t")
axes[0].set_xlim([0, 0.15])

# Plot C_y and C_y_t on the second subplot
axes[1].plot(t, C_y, color="green", label="C_y")
axes[1].plot(t, C_y_t, "k--", label="C_y theoretical")  # Black dashed line for
axes[1].set_xlabel("Time (s)")
axes[1].set_ylabel("C_y")
axes[1].legend()
axes[1].set_title("C_y and C_y_t")
axes[1].set_xlim([0, 0.015])

# Show the plot
plt.tight_layout()
plt.show()
```



In [18]:
```python
import numpy as np
def regularize(x_nu, t_nu, t):
    """
    Function to 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).
    """
    x = np.zeros(np.size(t))
    m = np.diff(x_nu) / np.diff(t_nu)  # Slopes of the different increments.

    s = 0  # Position in the wanted trajectory.
    for i in range(np.size(t_nu) - 1):

        # Select the spots in x (wanted trajectory) to set.
        s_end = np.where(t < t_nu[i+1])[0][-1]

        # Assign the values of the segment.
        x[s:s_end + 1] = x_nu[i] + m[i] * (t[s:s_end + 1]-t_nu[i])

        # Update the position in the wanted trajectory.
        s = s_end + 1

    return x
```

P1

In [19]:
```python
def lw_1D(T,alpha,v):
    import numpy as np
    x = []
    t = []
    x.append(0)
    t.append(0)
    #Continue until the previous time exceeds the duration
    while t[-1]<T:
        dt = np.random.rand()**(-1/(3-alpha))
        #Cumulative sum
        t.append(t[-1] + dt)
        w = np.random.choice([-1,1])
        x.append(x[-1] + v*w*dt)
    return x,t
```

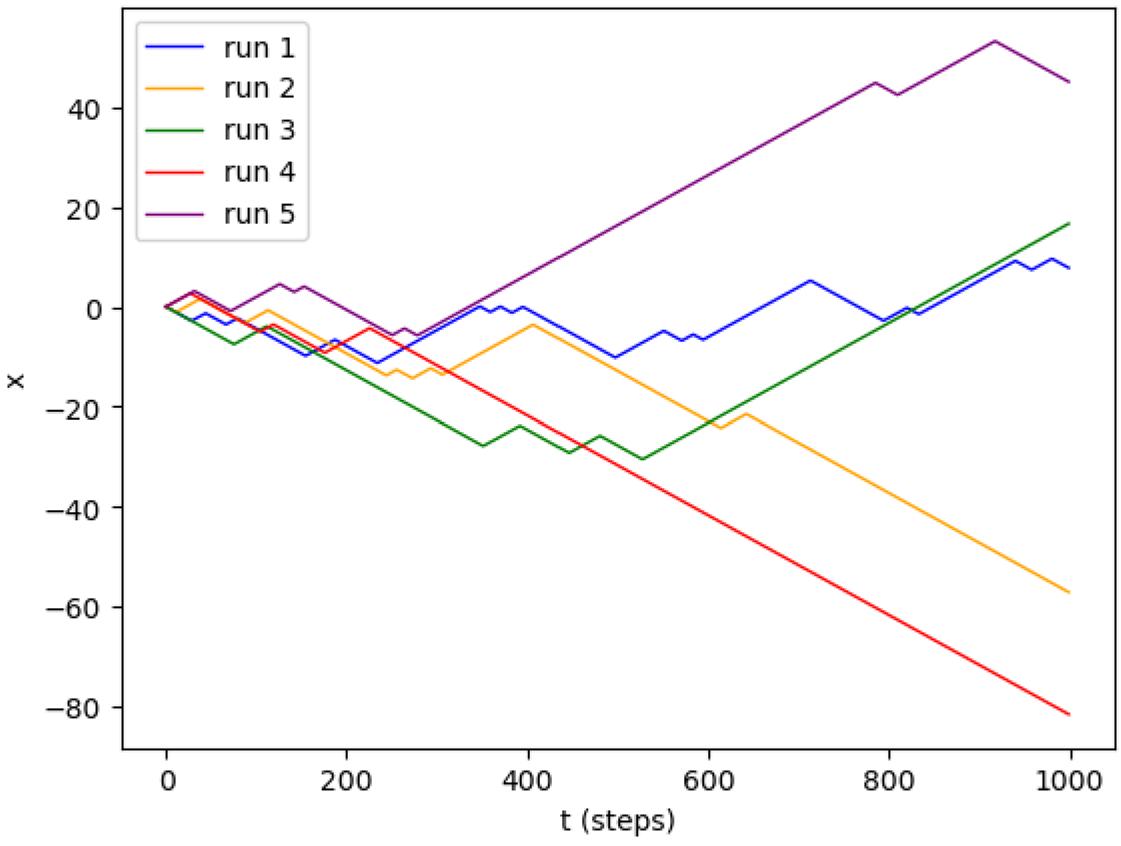
Uncomment to visualize what the regularization does.

In [ ]:
```python
duration = 100
dt = 0.1
## round up to cover all steps, also scale by dt. Used for regularization.
t = np.arange(int(np.ceil(duration / dt))) * dt
v = 1 # constant velocity
alpha = 2
runs = 5
trajectories = np.zeros((runs,len(t)))
for run in range(runs):
    x, t_sum = lw_1D(duration,alpha,v)
    x_r = regularize(x,t_sum,t)
    trajectories[run] = x_r

"""
x, t_sum = lw_1D(duration,alpha,v)
x_r = regularize(x,t_sum,t)
"""
```

```
Out[ ]: '\nx, t_sum = lw_1D(duration,alpha,v)\nx_r = regularize(x,t_sum,t)\n'
```

Uncomment to visualize what the regularization does.

```
In [21]: import matplotlib.pyplot as plt
colors = ['blue', 'orange', 'green', 'red', 'purple']
for run in range(runs):
    plt.plot(trajectories[run], '-', color=colors[run], linewidth=1, label=f'run


"""plt.plot(t_sum,x, 'o-', color='g', linewidth=1, label='Levi walk alpha=2')
plt.plot(t, x_r, '.', color='k', label='Regularized')
plt.xlim([0, 3])
plt.ylim([0, 3])"""

plt.legend()
plt.xlabel('t (steps)')
plt.ylabel('x')
plt.show()
```

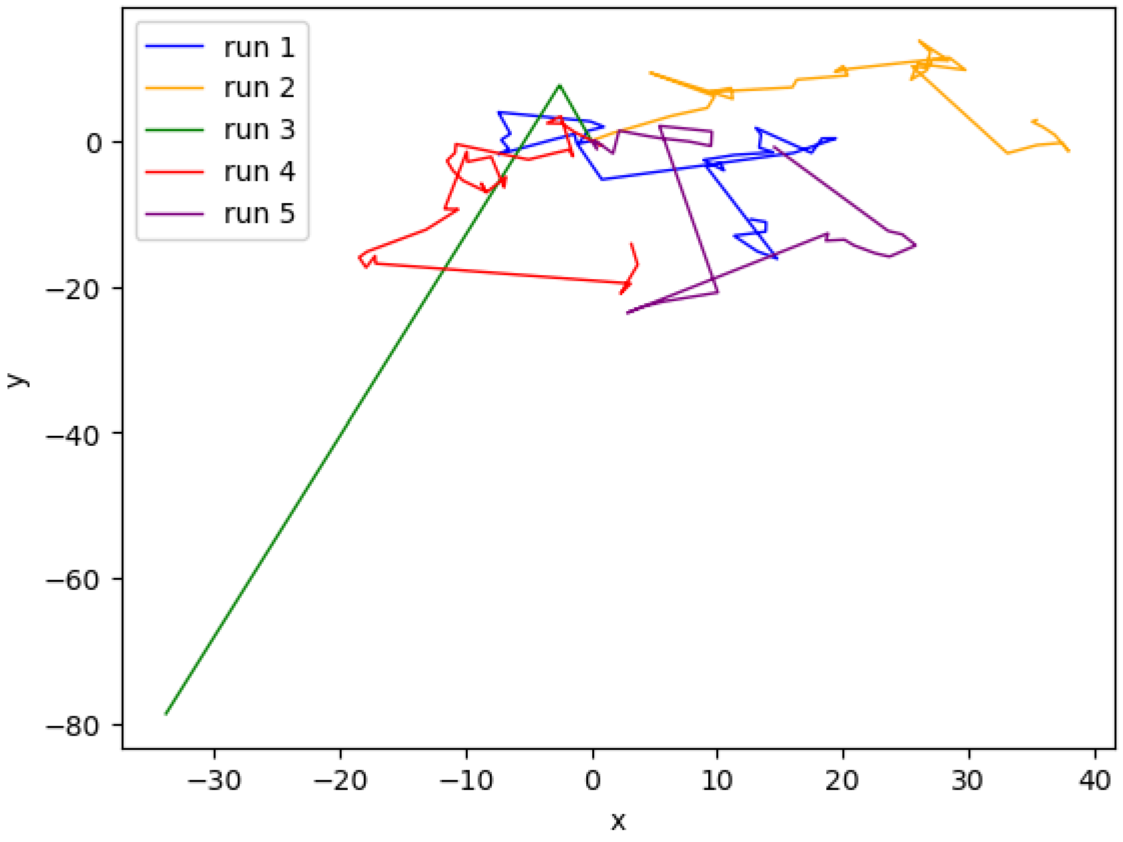


P2

In [22]:
```
def lw_2D(T,alpha,v):
    import numpy as np
    t = []
    x = []
    y = []
    x.append(0)
    y.append(0)
    t.append(0)
    while t[-1]<T:
        dt = np.random.rand()**(-1/(3-alpha))
        #Cumulative sum
        t.append(t[-1] + dt)
        phi = np.random.rand()*2*np.pi
        x.append(x[-1] + v*np.cos(phi)*dt)
        y.append(y[-1] + v*np.sin(phi)*dt)
    return x,y,t
```

In [23]:
```
duration = 100
dt = 0.1
t = np.arange(int(np.ceil(duration / dt))) * dt
v = 1 # constant velocity
alpha = 2
runs = 5
trajectories = np.zeros((2,runs,len(t))) #row 0 is x, row 1 y.
for run in range(runs):
    x,y,t_sum = lw_2D(duration, alpha, v)
    x = regularize(x,t_sum,t)
    y = regularize(y,t_sum,t)
    trajectories[0][run] = x
    trajectories[1][run] = y
```

In [24]:
```
import matplotlib.pyplot as plt
colors = ['blue', 'orange', 'green', 'red', 'purple']
for run in range(runs):
    plt.plot(trajectories[0][run],trajectories[1][run], '-', color=colors[run],
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



P3

- Time MSD: Measures how far a single particle moves over time.
- Ensemble MSD: Measures how far many particles move on average at a specific time.

In [25]:
```
def tMSD_1d(x, L):
    """
    Function to calculate the tMSD.

    Parameters
    ==========
    x : Trajectory (x component).
    L : Indicates the maximum delay (L * dt) considered.
    """

    tmsd = np.zeros(L)

    nelem = np.size(x)

    for n in range(L):
        Nmax = nelem - n
        dx = x[n:nelem] -  x[0: Nmax]
        tmsd[n] += np.mean(dx ** 2)

    return tmsd
```

In [26]:
```
def eMSD_1d(x):
    """
    Function to calculate the eMSD.

    Parameters
    ==========
    x : Trajectories: x[n_traj, i], bidimensional array.
    """

    N_traj, N_steps = x.shape

    # emsd = np.zeros(N_steps)

    emsd = np.mean(
        (x - np.repeat(x[:, 0].reshape(N_traj, 1), N_steps, axis=1)) ** 2,
        axis=0
    )

    return emsd
```

Calculate x_t for tMSD

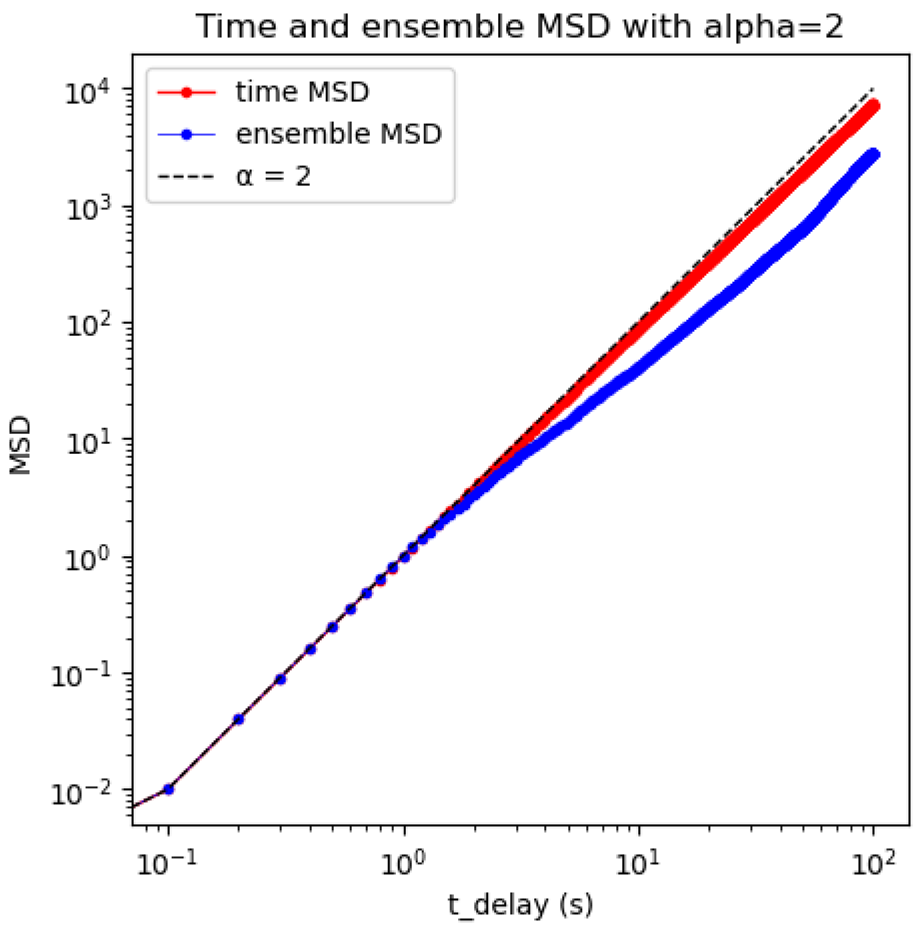
In [27]:
```
t_tot = 10000
N = 10000
dt = 0.1
v = 1

t_t = np.arange(int(np.ceil(t_tot / dt))) * dt
N_steps_t = np.size(t_t)

x_t, t_sum= lw_1D(t_tot, alpha, v)
x_t = regularize(x_t, t_sum, t_t)
```

Calculate x_e for eMSD

In [28]:
```
t_tot = 100
N = 100
dt = 0.1

# Regular sampling with dt.
t_e = np.arange(int(np.ceil(t_tot / dt))) * dt
N_steps_e = np.size(t_e)

N_traj = 100

x_e = np.zeros([N_traj, N_steps_e])

for i in range(N_traj):
    x, t_sum = lw_1D(t_tot, alpha, v)
    x_r = regularize(x,t_sum, t_e)
    x_e[i, :] = x_r
```

In [29]:
```
# Calculate eMSD
emsd = eMSD_1d(x_e)  # eMSD from ensemble trajectories.

# Calculate tMSD
tmsd = tMSD_1d(x_t, N_steps_e)  # tMSD from long trajectory.
```

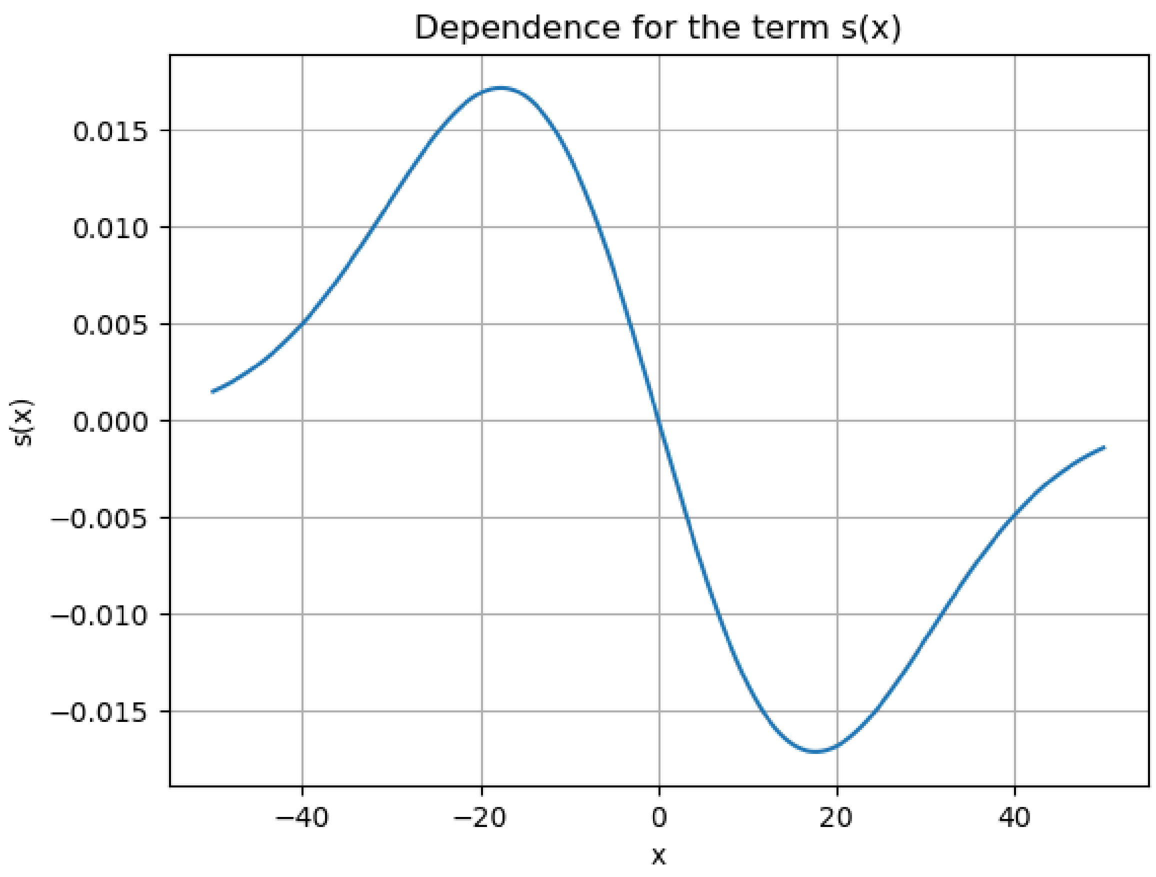
In [30]:
```
plt.figure(figsize=(5,5))
plt.loglog(t_e, tmsd, '.-', color='r', linewidth=1,
         label='time MSD')
plt.loglog(t_e, emsd, '.-', color='b', linewidth=0.5,
         label='ensemble MSD')
plt.loglog(t_e, t_e**alpha, 'k--', linewidth=1, label=f'α = {alpha}')
plt.title(f"Time and ensemble MSD with alpha={alpha}")
plt.legend()
plt.xlabel('t_delay (s)')
plt.ylabel('MSD')
plt.show()
```

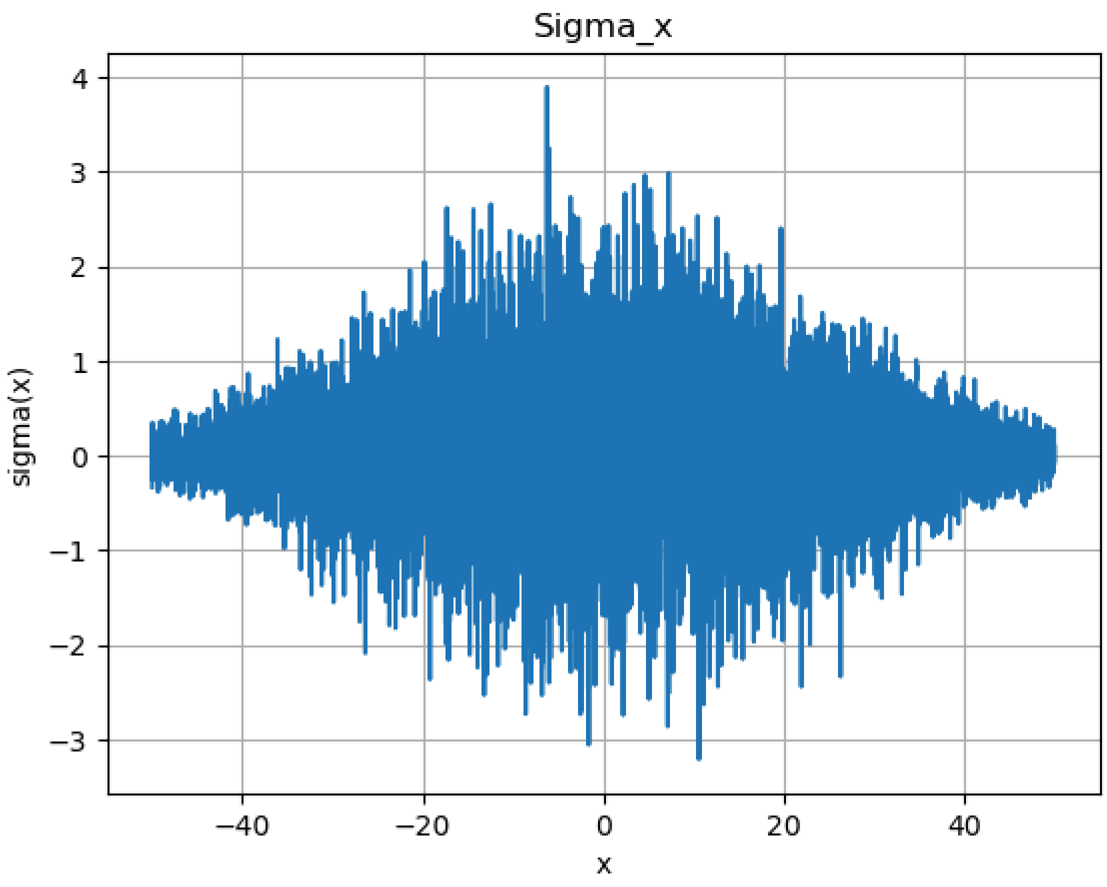


Seems to be ergodic since the time and ensemble MSD aligns closely.

P1

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
sigma0 = 1
w0 = 25
L = 100

xmin = -L/2
xmax = L/2

x = np.linspace(xmin,xmax,10000)
s_x = np.zeros(len(x))

for i in range(len(x)):
    tmp = -x[i]*(sigma0/w0)**2
    s_x[i] = tmp*np.exp(-(x[i]**2)/(w0**2))

plt.plot(x, s_x)
plt.xlabel('x')
plt.ylabel('s(x)')
plt.title('Dependence for the term s(x)')
plt.grid(True)
plt.show()


### Used to understand the equation.
rn = np.random.normal(0,1,len(x))
sigma = sigma0*np.exp(-(x**2)/(2*w0**2))*rn

plt.plot(x, sigma)
plt.xlabel('x')
plt.ylabel('sigma(x)')
plt.title('Sigma_x')
plt.grid(True)
plt.show()
```





P2

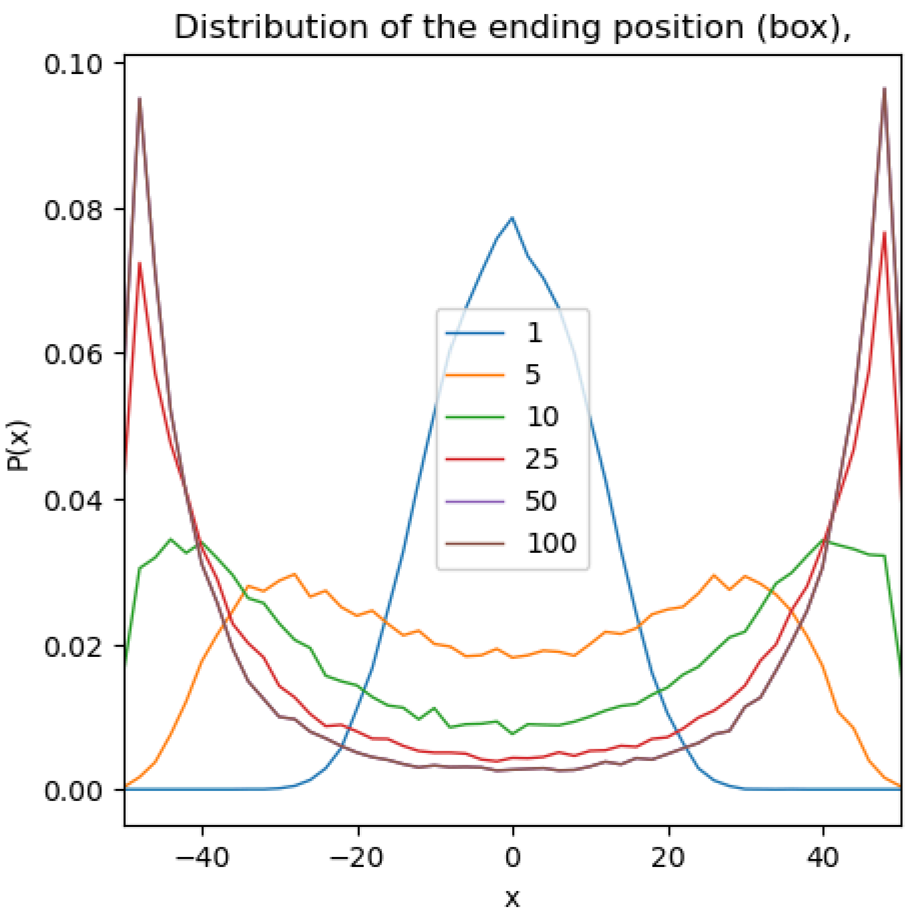
```
In [49]: alphas = np.array([0,0.5,1])
         sigma0 = 1
```

```
dt = 1  # Time step.
N_traj = 50000 # Number of independent trajectories.
t0 = 100  # Base value of the duration.
j_mult = np.array([1, 5, 10,25, 50, 100])

x0 = 0  # Initial position [m].

L = 100
x_min = - L / 2
x_max = L / 2

w0 = 25

x_finals = []

for idx,alpha in enumerate(alphas):

    x_fin = np.zeros([N_traj, np.size(j_mult)])  # Final positions.

    for j in range(np.size(j_mult)):

        # Set the number of steps to calculate further.
        if j > 1:
            N_steps = int(np.ceil((j_mult[j] - j_mult[j - 1]) * t0 / dt))
        else:
            N_steps = int(np.ceil(j_mult[j] * 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 = sigma0*np.exp((-x**2)/(2*w0**2))

            tmp = -x*(sigma0/w0)**2
            s_x = tmp*np.exp(-(x**2)/(w0**2))
            x += alpha*s_x*dt + sigma_x*np.sqrt(dt)* rn[:, step]


            # reflecting boundary conditions
            bounce_left = np.where(x < x_min)[0]  # Hitting box left end.
            x[bounce_left] = 2 * x_min - x[bounce_left]
            bounce_right = np.where(x > x_max)[0]  # Hitting box right end.
            x[bounce_right] = 2 * x_max - x[bounce_right]

        x_fin[:, j] = x

    x_finals.append(x_fin)
```

In [50]:

```
# Histogram of the final positions.
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)])  # Distributions.
```

```
for j in range(np.size(j_mult)):
    distribution = np.histogram(x_finals[0][:, j], bins=bins_edges)
    p_distr[:, j] = distribution[0] / np.sum(distribution[0])

plt.figure(figsize=(5, 5))
for j in range(np.size(j_mult)):
    plt.plot(bins, p_distr[:, j], '-', linewidth=1, label=str(j_mult[j]))
plt.title('Distribution of the ending position (box),')
plt.legend()
plt.xlabel('x')
plt.ylabel('P(x)')
plt.xlim([x_min, x_max])
plt.show()
```

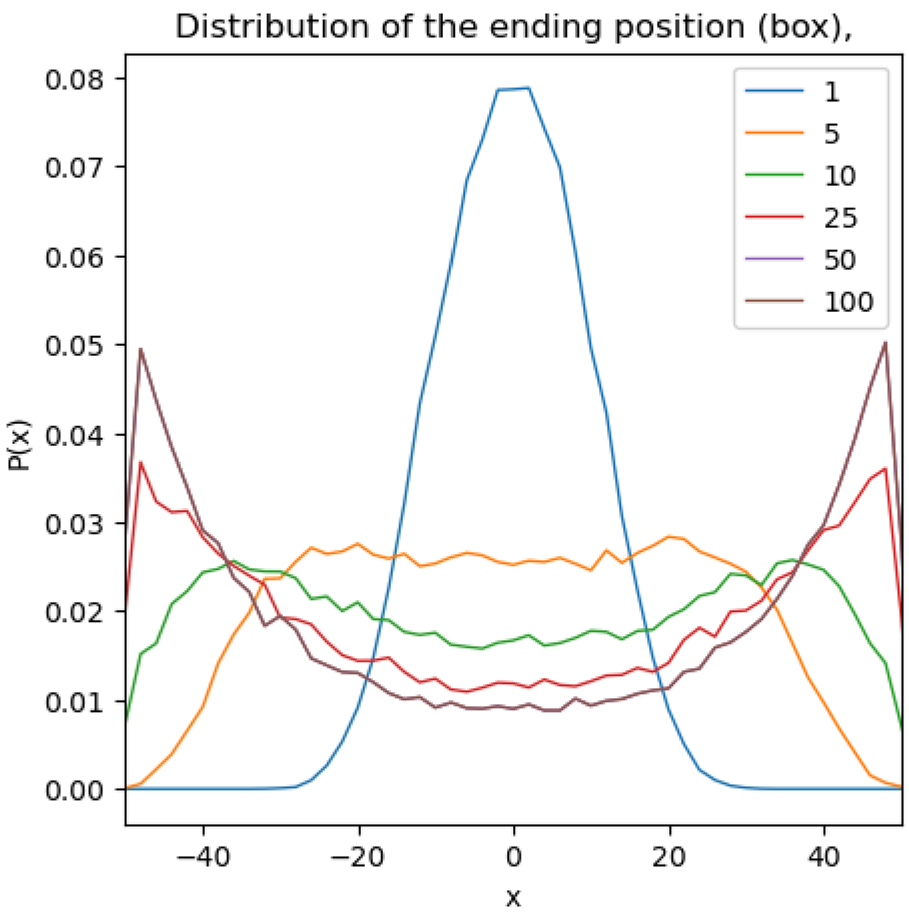


P3

In [51]:

```
# Histogram of the final positions.
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)])  # Distributions.

for j in range(np.size(j_mult)):
    distribution = np.histogram(x_finals[1][:, j], bins=bins_edges)
    p_distr[:, j] = distribution[0] / np.sum(distribution[0])

plt.figure(figsize=(5, 5))
```

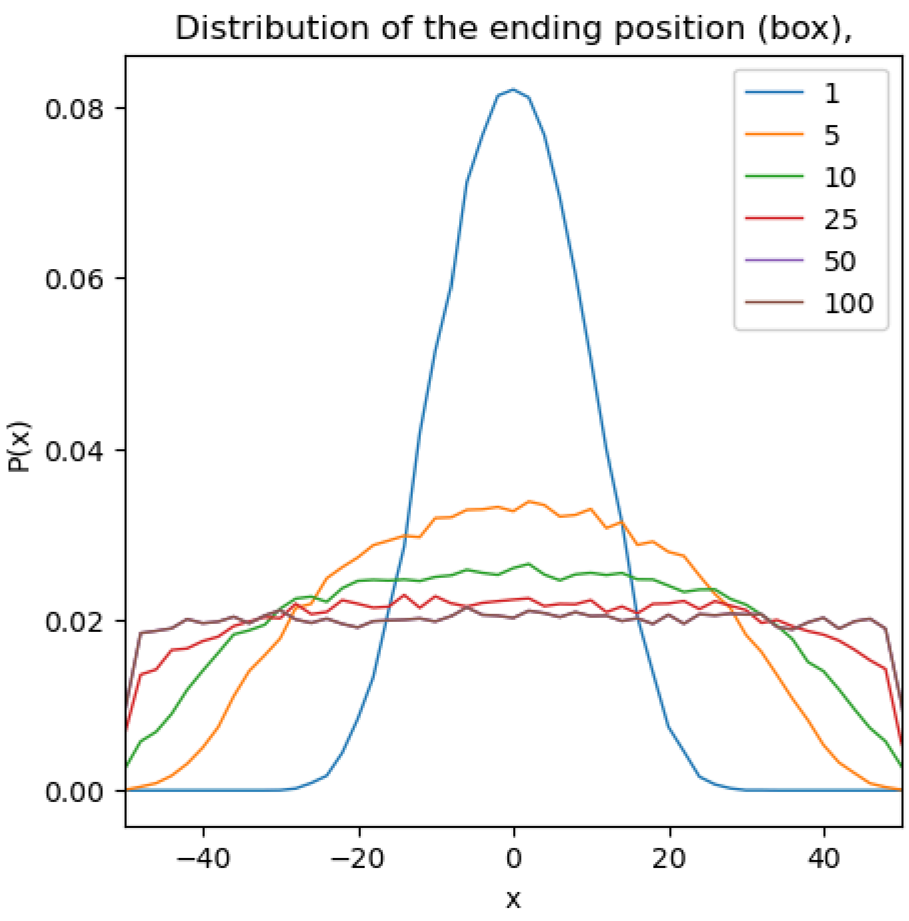
```
for j in range(np.size(j_mult)):
    plt.plot(bins, p_distr[:, j], '-', linewidth=1, label=str(j_mult[j]))
plt.title('Distribution of the ending position (box),')
plt.legend()
plt.xlabel('x')
plt.ylabel('P(x)')
plt.xlim([x_min, x_max])
plt.show()
```



P4

In [52]:

```
# Histogram of the final positions.
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)])  # Distributions.

for j in range(np.size(j_mult)):
    distribution = np.histogram(x_finals[2][:, j], bins=bins_edges)
    p_distr[:, j] = distribution[0] / np.sum(distribution[0])

plt.figure(figsize=(5, 5))
for j in range(np.size(j_mult)):
    plt.plot(bins, p_distr[:, j], '-', linewidth=1, label=str(j_mult[j]))
plt.title('Distribution of the ending position (box),')
plt.legend()
plt.xlabel('x')
plt.ylabel('P(x)')
plt.xlim([x_min, x_max])
plt.show()
```



Q1

It seems as the multiplicative noise $\sigma(x)$ follows a gaussian distribution with mean 0 and standard deviation w_0, thus the standard deviation is symmetric around x=0, causing the system to be symmetric around 0. Moreover, we have reflective boundary conditions which further contributes to the symmetry of the system.

Note: From plotting the noise-induced drift in P1, and also plotting the right-most term in equation 2. We see that they seem to counteract eachother based on the position of x. Moreover, the noise-induced drift term increase when alpha increases, so when $\alpha = 1$ we fully incorporate the drift. That is why we see higher probabilities toward the boundaries for $\alpha < 1$, since the right-most term in eq 2 is larger than the noise-induced drift.

Task 1

P1

In [4]:
```
import math
import numpy as np

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
```

In [5]:
```
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
```

In [6]:
```
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
```

In [7]:
```
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 is 2.
    # Vertices are listed consecutively.

    A = 0

    for i in range(N-1):
        # Below is the formula of the area of a triangle given the vertices.
        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
```

In [8]:
```
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
```

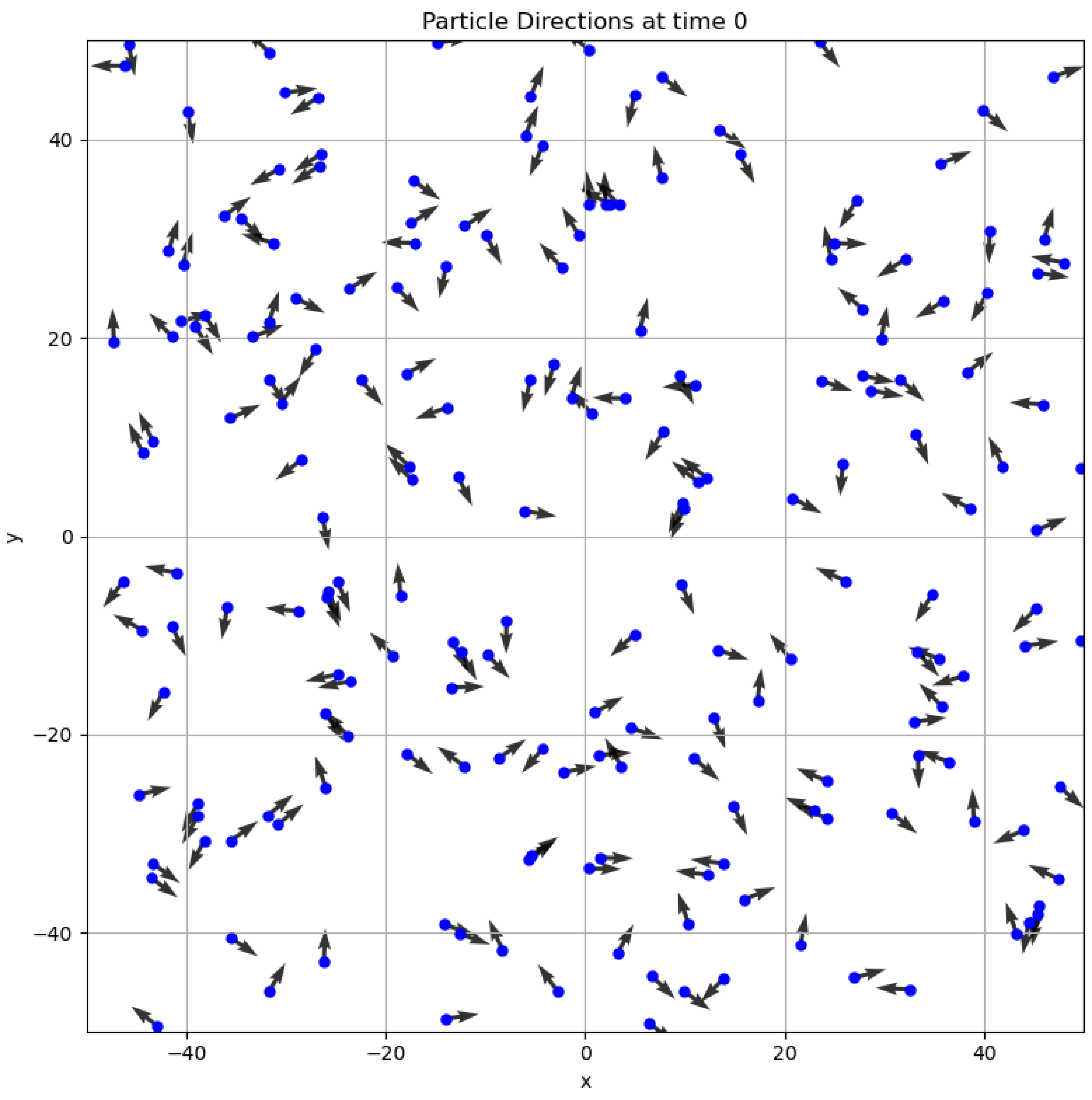
In [72]:
```
N_part = 200
L = 100
v = 1
Rf = 2
eta = 0.01
dt = 1

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

theta = 2 * (np.random.rand(N_part) - 0.5) * np.pi  # in [-pi, pi]
```

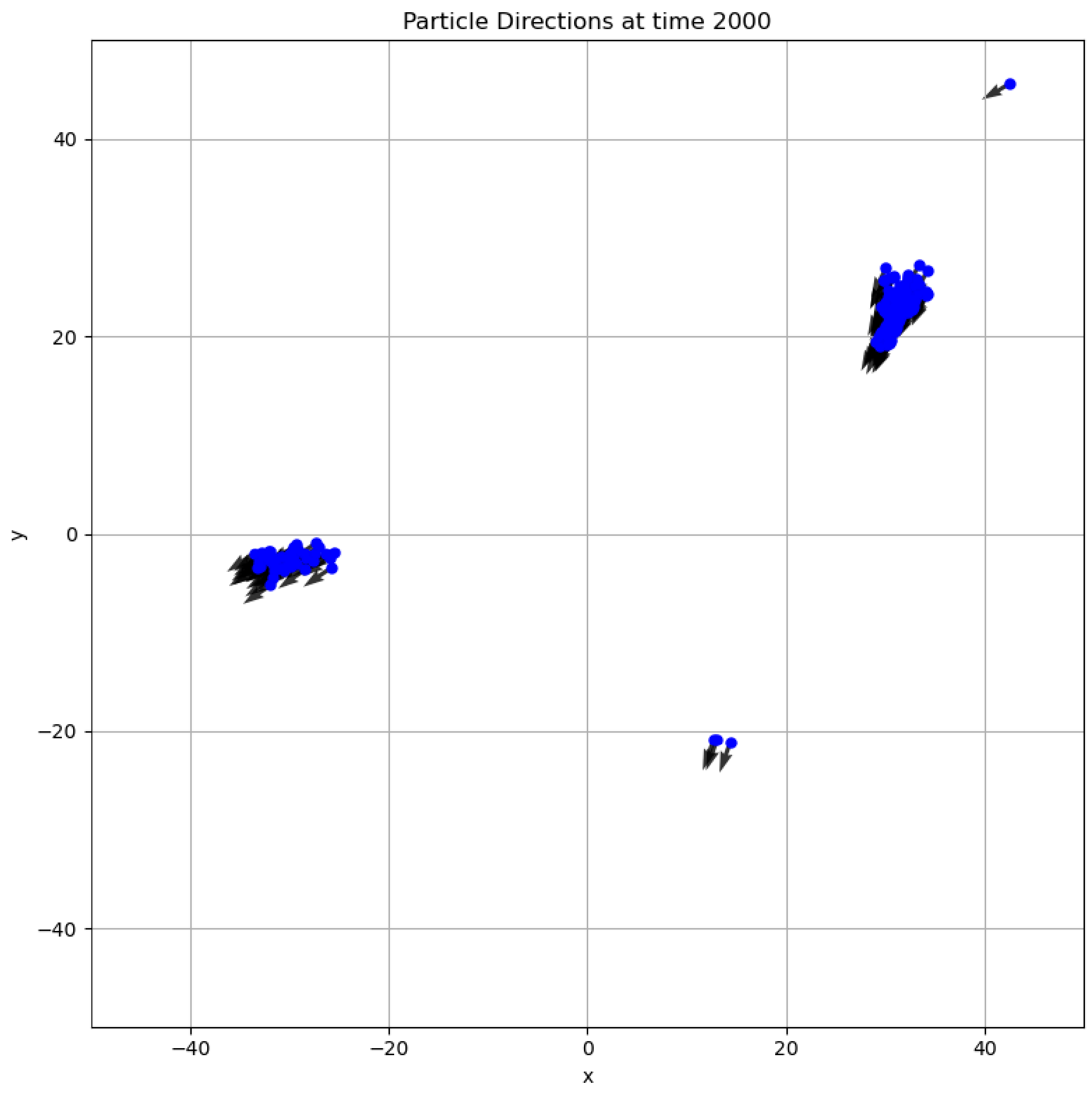
In [ ]:
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *


step = 0
t_tot = 6000

config_x1 = []
config_y1 = []
config_theta1 = []
```

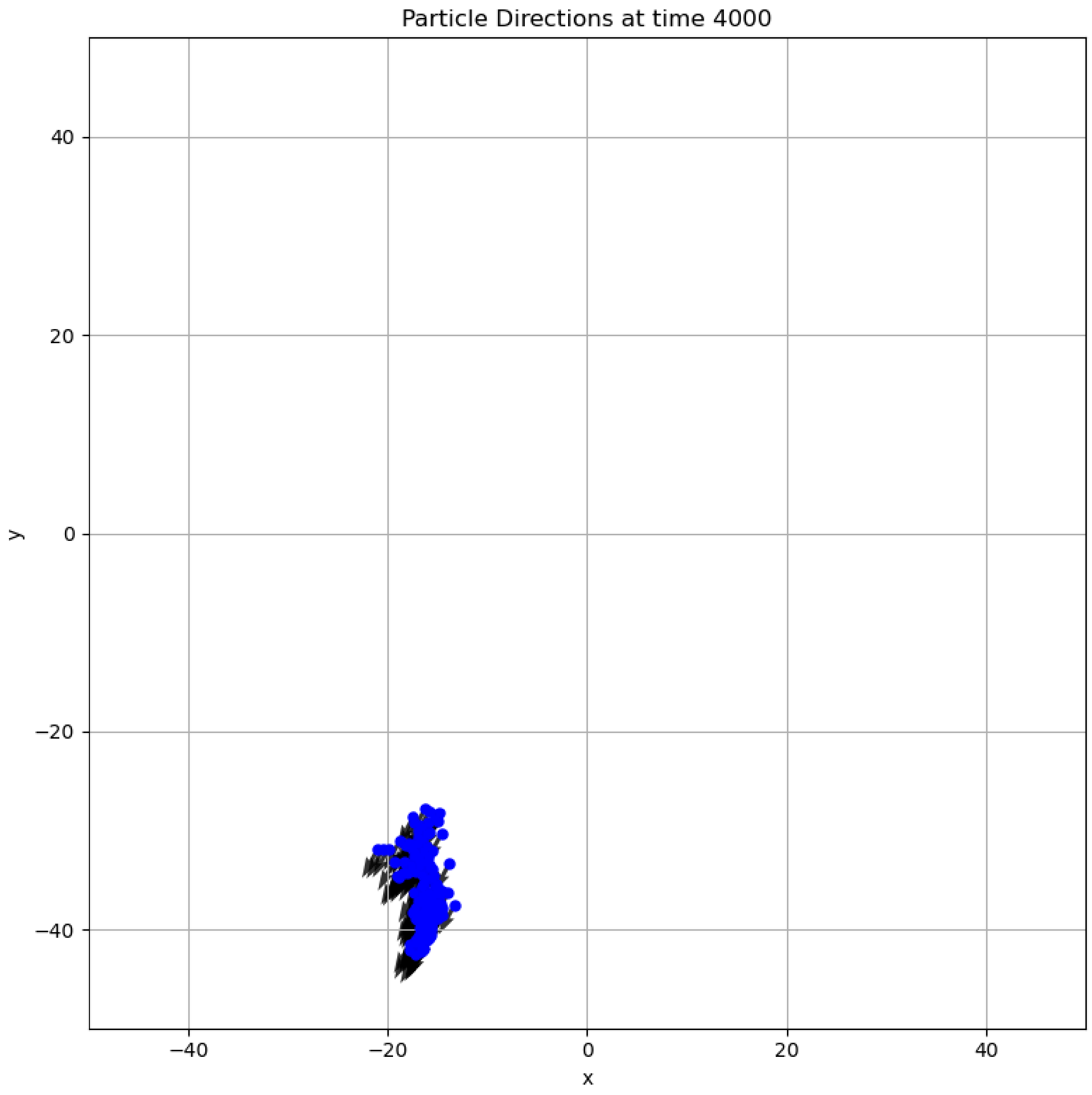
```
psi = np.zeros(t_tot+1)
c = np.zeros(t_tot+1)

while step <= t_tot:
    psi[step] = global_alignment(theta)
    c[step] = global_clustering(x, y, Rf, L)

    # Calculate next theta from the rule.
    dtheta = eta * (np.random.rand(N_part) - 0.5) * dt
    ntheta = interaction(x, y, theta, Rf, L) + dtheta
    nx = x + v * np.cos(ntheta)
    ny = y + v * np.sin(ntheta)

    # Reflecting boundary conditions.
    nx, ny = pbc(nx, ny, L)

    if(step == 0 or step == 2000 or step == 4000 or step == 6000):
        config_x1.append(nx[:])
        config_y1.append(ny[:])
        config_theta1.append(ntheta[:])

    step += 1
    if step % 100 == 0:
        print(step)

    x[:] = nx[:]
    y[:] = ny[:]
    theta[:] = ntheta[:]
```

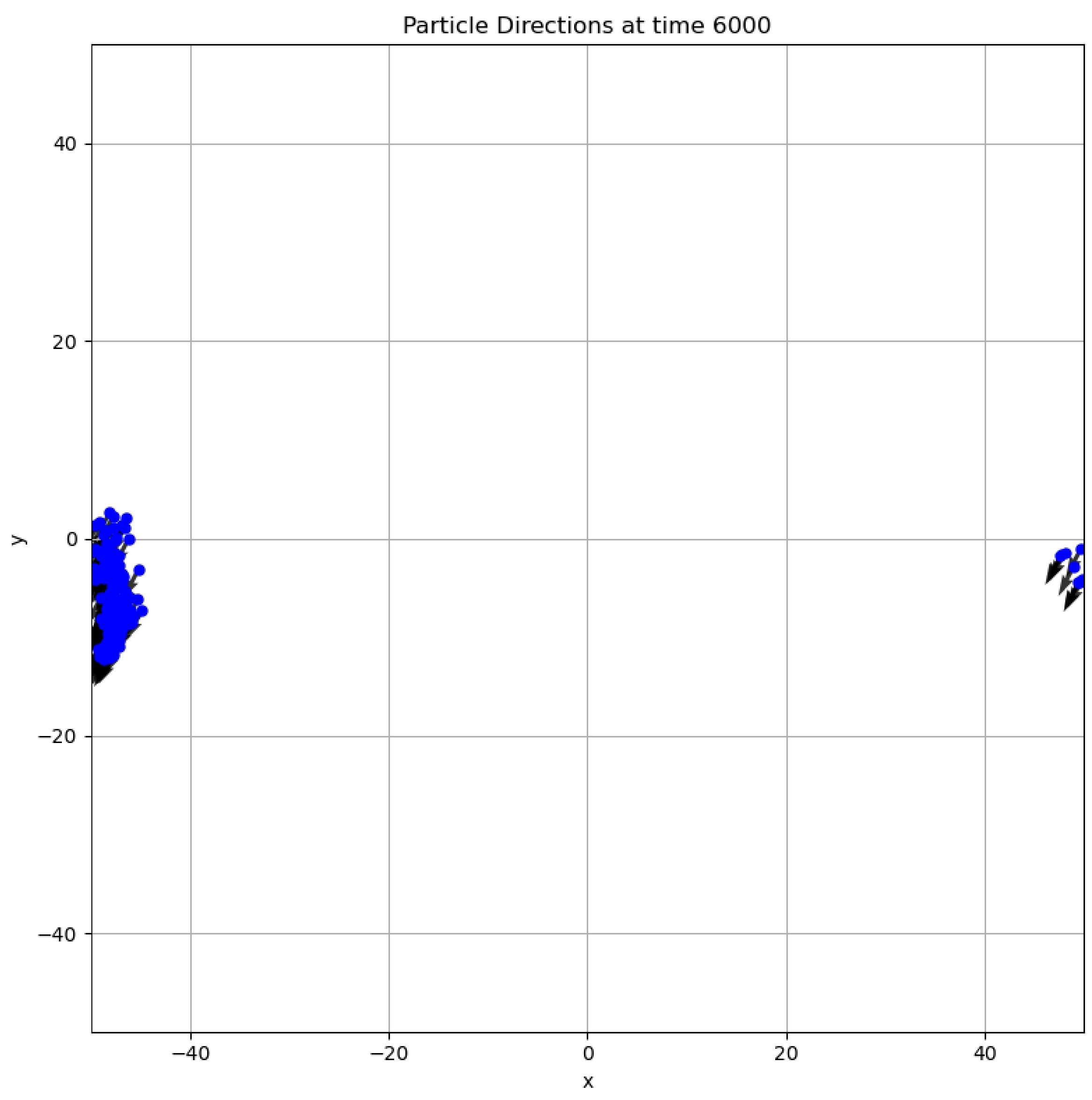
```
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
4400
4500
4600
4700
4800
4900
5000
5100
5200
5300
5400
5500
5600
5700
5800
5900
6000
```

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


        # Prepare time steps corresponding to snapshots
        time_steps = [0, 2000, 4000, 6000]

        # Plot arrows representing particle directions at each snapshot
        for i, time in enumerate(time_steps):
            plt.figure(figsize=(8, 8))
            x_positions_p1 = config_x1[i]
            y_positions_p1 = config_y1[i]
            angles = config_theta1[i]

            u_p1 = np.cos(angles)
            v_p1 = np.sin(angles)

            plt.plot(x_positions_p1, y_positions_p1, '.', color='blue', markersize=10)
            plt.quiver(x_positions_p1, y_positions_p1, u_p1, v_p1, color='black', angles
            """plt.scatter(x_positions_p1, y_positions_p1 )
            for j in range(len(x_positions_p1)):
                plt.arrow(x_positions_p1[j], y_positions_p1[j], u[j], v[j])"""
            plt.title(f'Particle Directions at time {time}')
            plt.xlabel('x')
            plt.ylabel('y')
            plt.grid()
            plt.xlim([-L/2,L/2])
            plt.ylim([-L/2,L/2])
            plt.tight_layout()
            plt.show()
```


Particle Directions at time 0
y
x
40
20
0
−20
−40
−40
−20
0
20
40

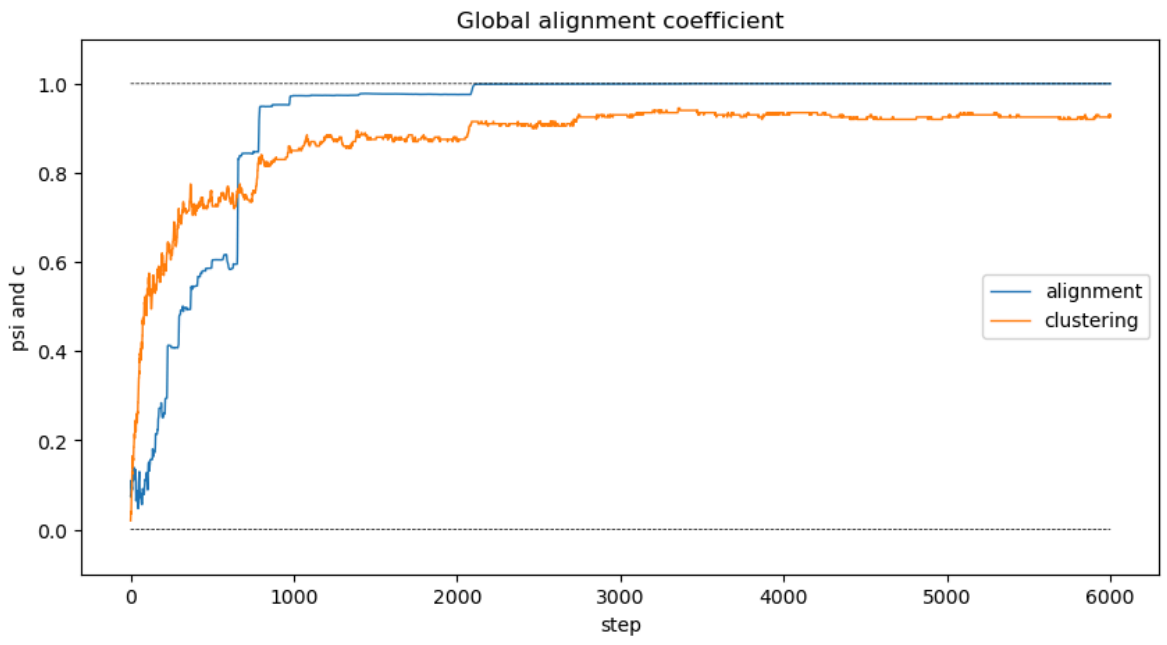

Particle Directions at time 2000
40
20
0
−20
−40
y
−40
−20
0
20
40
x


Particle Directions at time 4000
y
x
40
20
0
−20
−40
−40
−20
0
20
40



P2

```
In [75]: from matplotlib import pyplot as plt
         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 and c')
         plt.ylim([-0.1, 1.1])
         plt.show()
```



Task 2

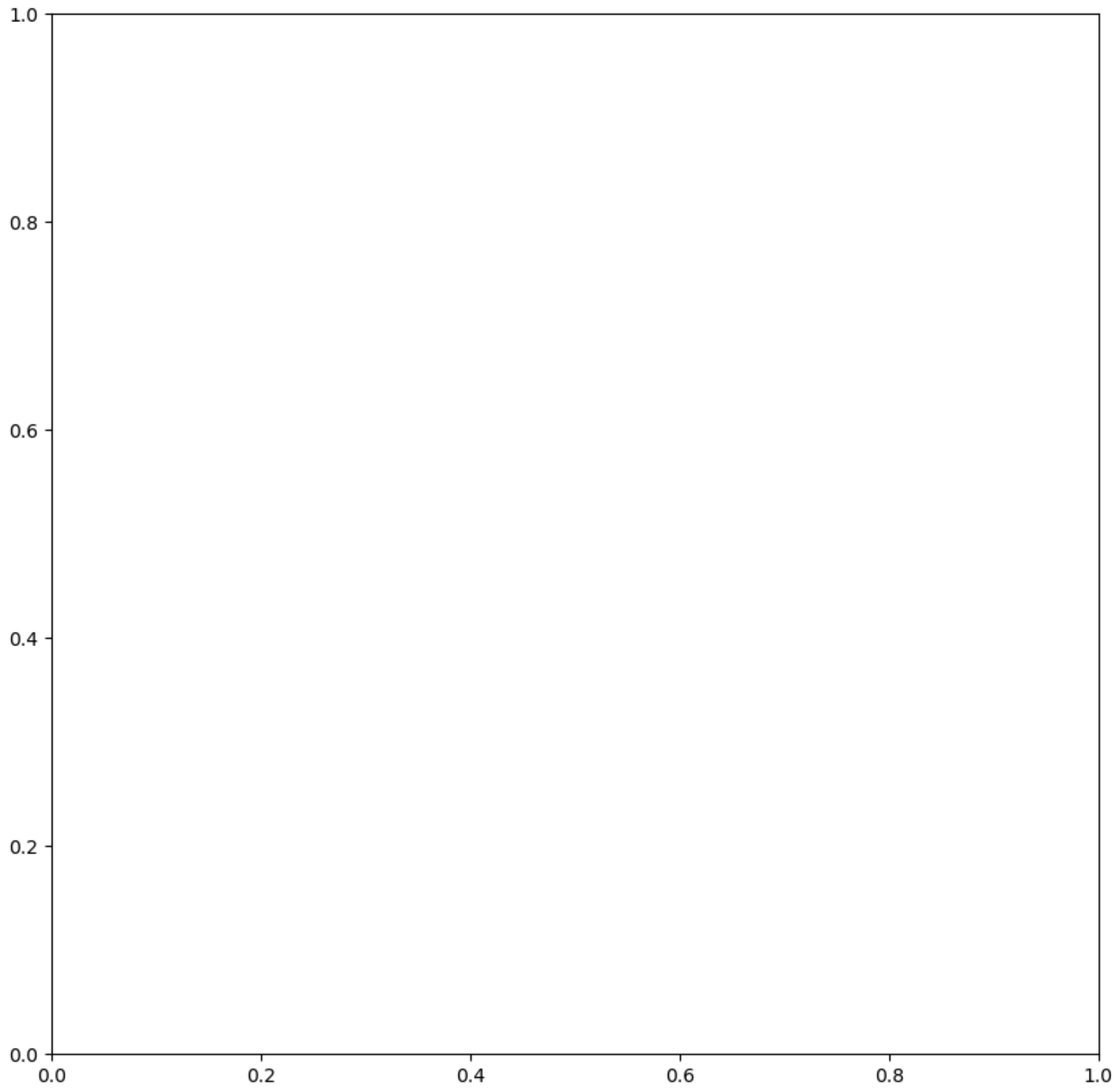
P3

In [9]:
```
import numpy as np
N_part = 200
L = 100
v = 1
Rf = 2
eta1 = 0.01
eta2 = 0.3
dt = 1

x = (np.random.rand(N_part) - 0.5) * L
y = (np.random.rand(N_part) - 0.5) * L

# Random orientation.
theta = 2 * (np.random.rand(N_part) - 0.5) * np.pi
```

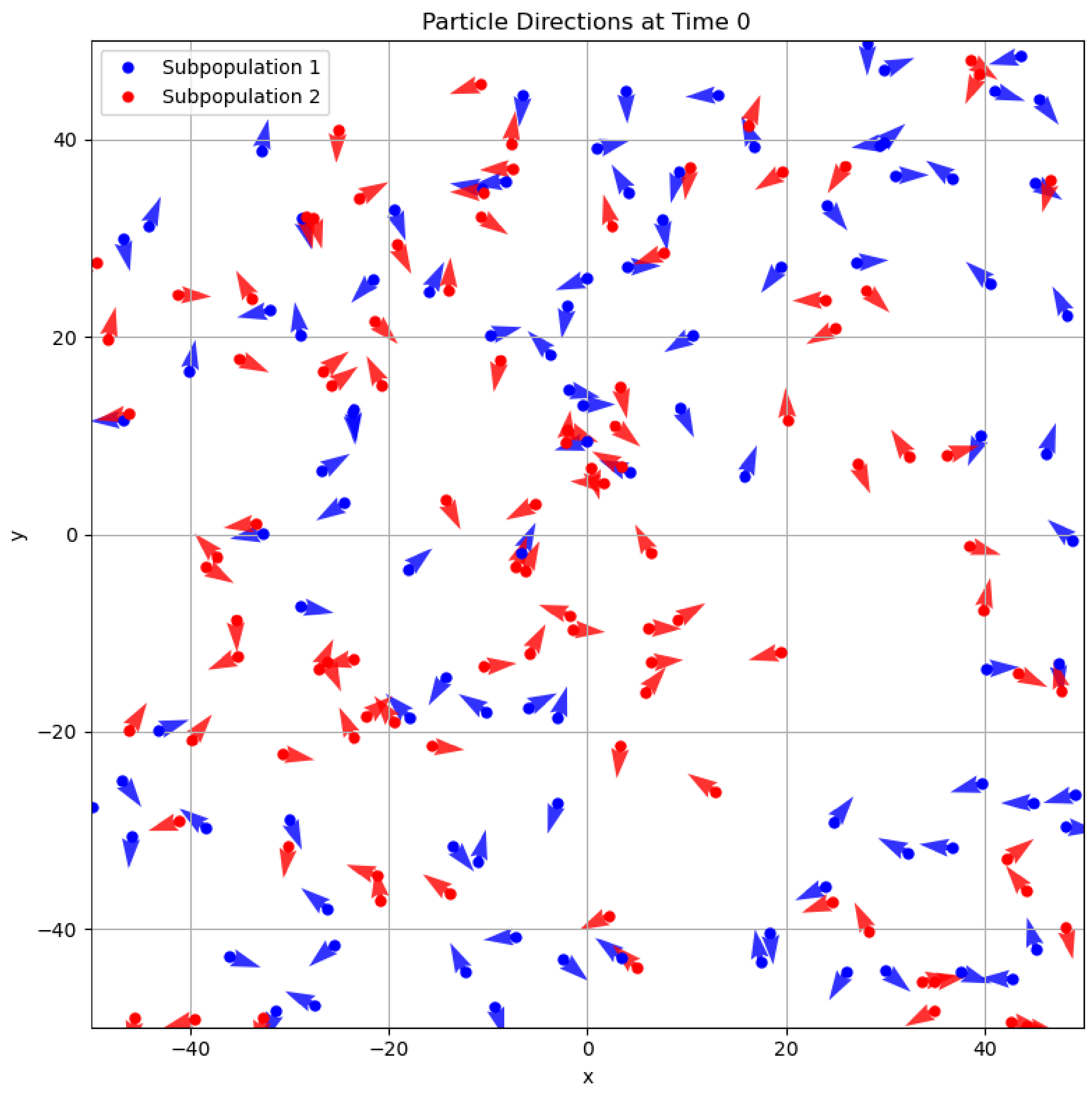
In [10]:
```
from IPython.display import clear_output
from matplotlib import pyplot as plt
import time

step = 0
t_tot = 6000
half_N = N_part // 2

config_x = []
config_y = []
config_theta = []

psi2 = np.zeros(t_tot+1)
c2 = np.zeros(t_tot+1)
fig, ax = plt.subplots(figsize=(10, 10))

while step <= t_tot:
```

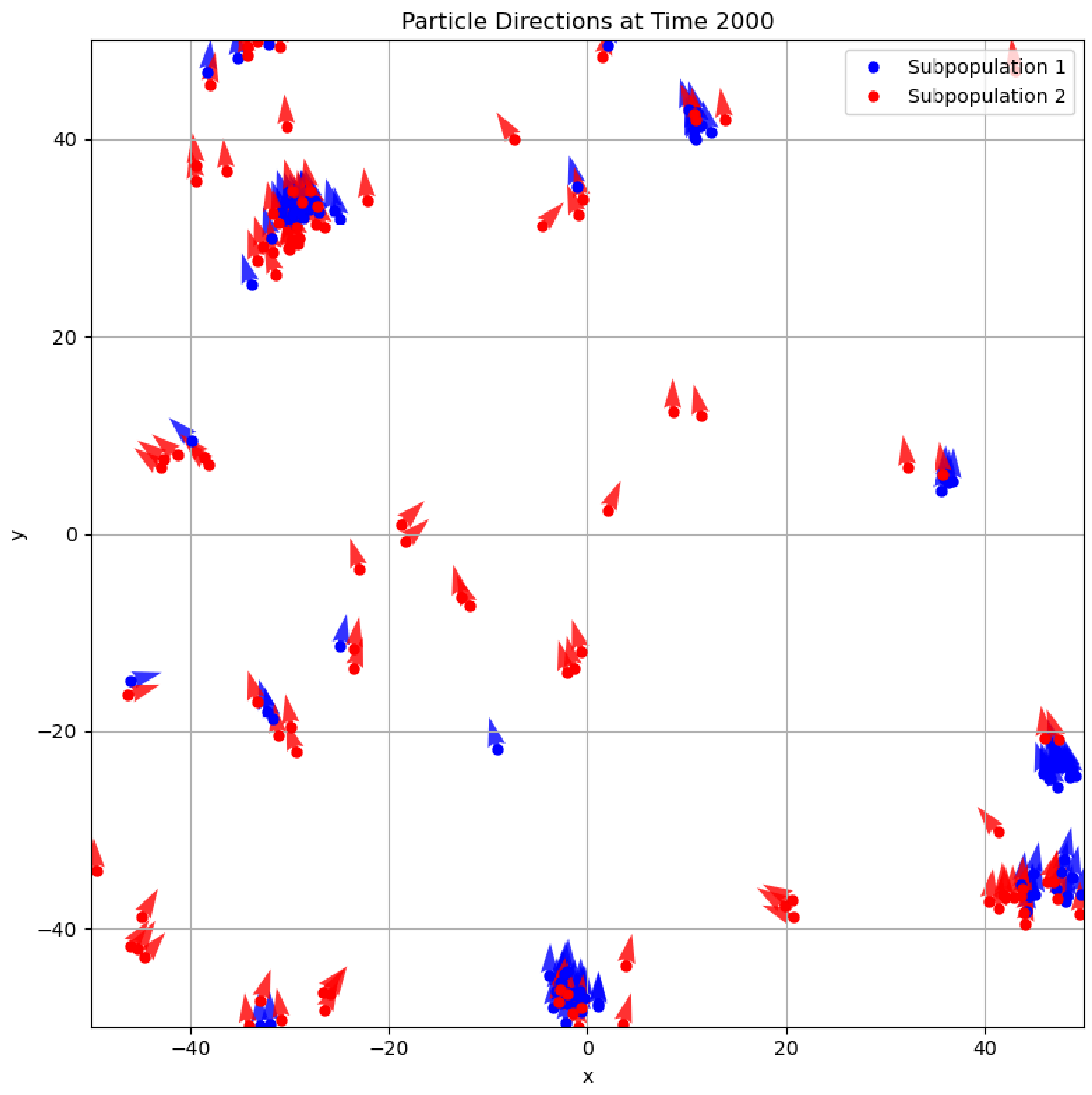
```python
    """ if step % 1 == 0:
        ax.clear()  # Clear previous plot.

        # Plot first subpopulation
        ax.plot(x[:half_N], y[:half_N], '.', markersize=10, color='blue', label=
        ax.quiver(x[:half_N], y[:half_N], np.cos(theta[:half_N]), np.sin(theta[:

        # Plot second subpopulation
        ax.plot(x[half_N:], y[half_N:], '.', markersize=10, color='red', label='
        ax.quiver(x[half_N:], y[half_N:], np.cos(theta[half_N:]), np.sin(theta[h

        # Plot boundary (if needed)
        ax.plot(Rf * np.cos(2 * np.pi * np.arange(360) / 360),
                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}')
        ax.legend(loc='upper right')  # Add legend to distinguish subpopulations

        display(fig)  # Display updated plot.
        clear_output(wait=True)  # Clear previous output."""

    psi2[step] = global_alignment(theta)
    c2[step] = global_clustering(x, y, Rf, L)

    dtheta = np.zeros(N_part)

    dtheta[:half_N] = eta1 * (np.random.rand(half_N) - 0.5) * dt
    dtheta[half_N:] = eta2 * (np.random.rand(N_part-half_N) - 0.5) * dt

    ntheta = interaction(x, y, theta, Rf, L) + dtheta
    nx = x + v * np.cos(ntheta)
    ny = y + v * np.sin(ntheta)

    # Reflecting boundary conditions.
    nx, ny = pbc(nx, ny, L)

    if(step == 0 or step == 2000 or step == 4000 or step == 6000):
        config_x.append(nx[:])
        config_y.append(ny[:])
        config_theta.append(ntheta[:])

    step += 1
    if step % 100 == 0:
        print(step)

    x[:] = nx[:]
    y[:] = ny[:]
    theta[:] = ntheta[:]
```

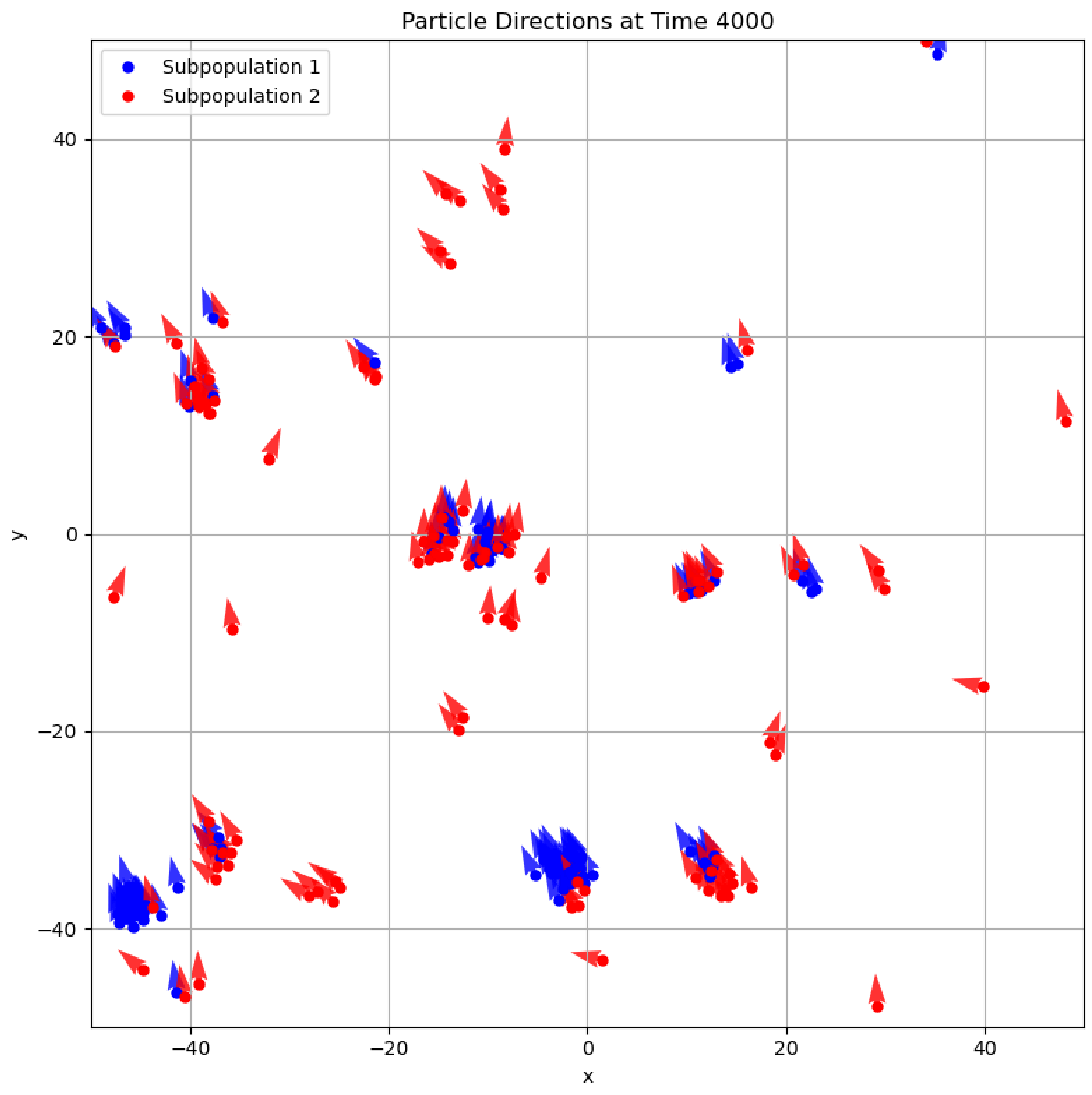
```
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
4400
4500
4600
4700
4800
4900
5000
5100
5200
5300
5400
5500
5600
5700
5800
5900
6000
```

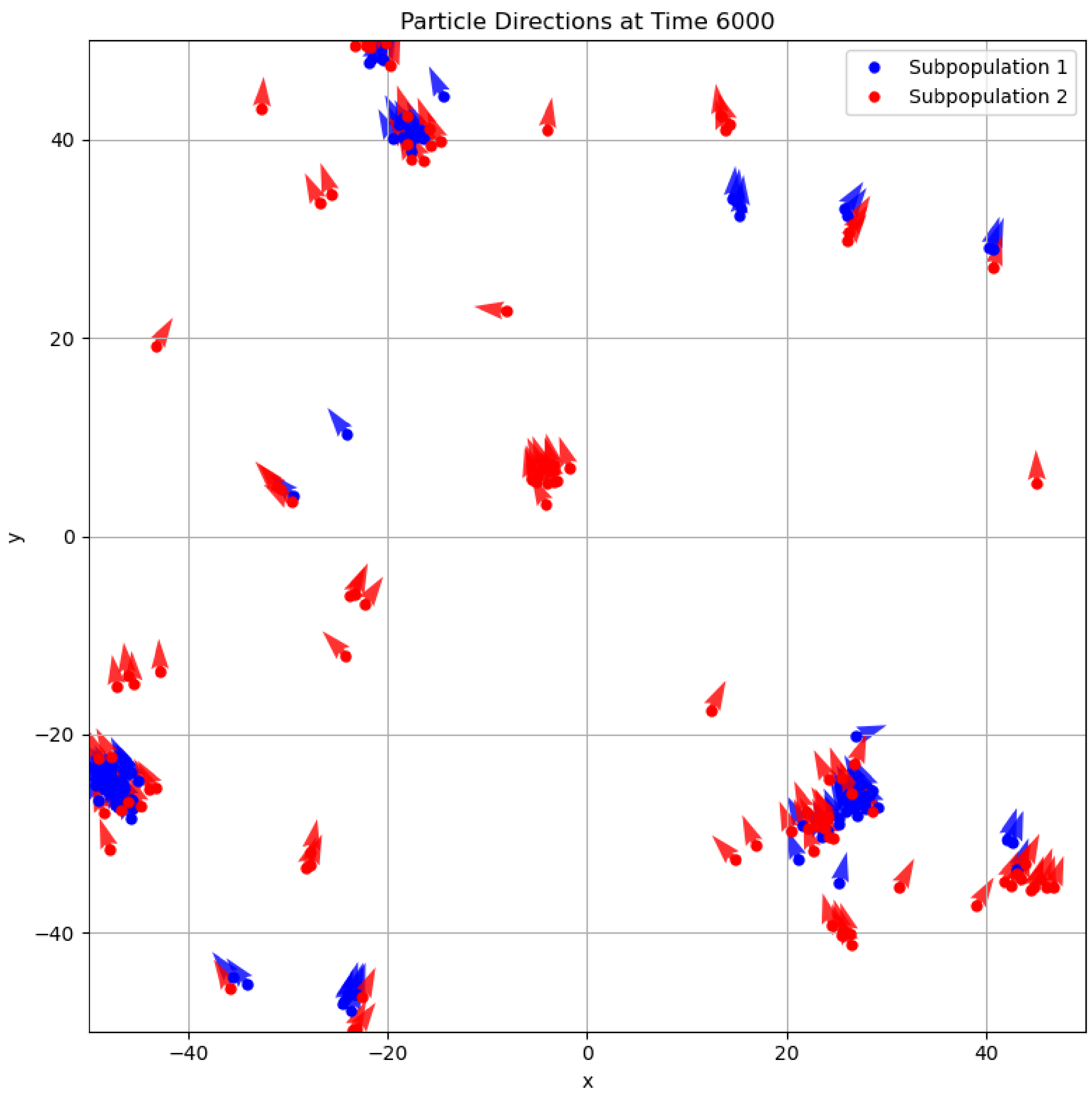


In [11]:
```
time_steps = [0, 2000, 4000, 6000]
N_half = N_part // 2

for i, time in enumerate(time_steps):
    plt.figure(figsize=(8, 8))

    x_positions = config_x[i]
    y_positions = config_y[i]
    angles = config_theta[i]

    x_positions_1, x_positions_2 = x_positions[:N_half], x_positions[N_half:]
    y_positions_1, y_positions_2 = y_positions[:N_half], y_positions[N_half:]
    angles_1, angles_2 = angles[:N_half], angles[N_half:]

    u1, v1 = np.cos(angles_1), np.sin(angles_1)
    u2, v2 = np.cos(angles_2), np.sin(angles_2)

    plt.plot(x_positions_1, y_positions_1, '.', color='blue', markersize=10, lab
    plt.plot(x_positions_2, y_positions_2, '.', color='red', markersize=10, labe

    plt.quiver(x_positions_1, y_positions_1, u1, v1, color='blue', angles='xy', 
    plt.quiver(x_positions_2, y_positions_2, u2, v2, color='red', angles='xy', s

    plt.title(f'Particle Directions at Time {time}')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.xlim([-L / 2, L / 2])
    plt.ylim([-L / 2, L / 2])
    plt.legend()
    plt.tight_layout()
    plt.show()
```



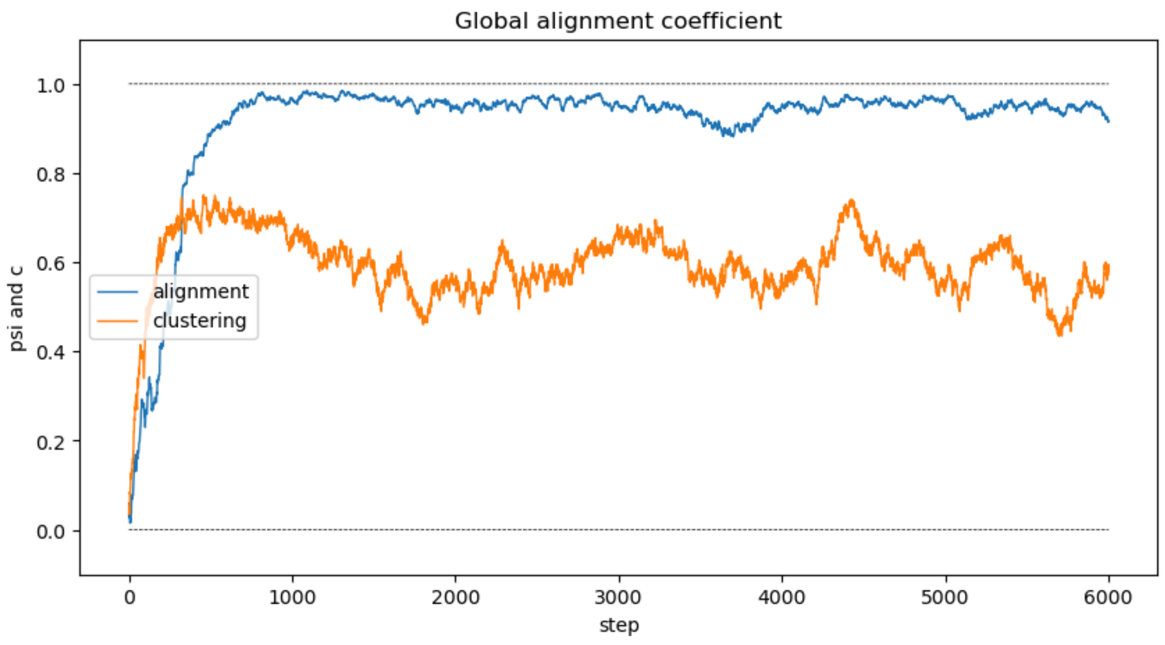

Particle Directions at Time 2000
Subpopulation 1
Subpopulation 2
x
y
−40
−20
0
20
40


Particle Directions at Time 4000
Subpopulation 1
Subpopulation 2
40
20
0
−20
−40
y
−40
−20
0
20
40
x



P4

In [12]:
```
from matplotlib import pyplot as plt
plt.figure(figsize=(10, 5))
plt.plot(psi2, '-', linewidth=1, label='alignment')
plt.plot(c2, '-', linewidth=1, label='clustering')
plt.plot(0 * psi2, '--', color='k', linewidth=0.5)
plt.plot(0 * psi2 + 1, '--', color='k', linewidth=0.5)
plt.title('Global alignment coefficient')
plt.legend()
plt.xlabel('step')
plt.ylabel('psi and c')
plt.ylim([-0.1, 1.1])
plt.show()
```



Q1

Studying the simulation, i can see that when clusters are formed, the low-noise particles remain aligned whereas the high-noise particles tend to defer from the alignment of the cluster. That is, the high-noise particles tend to move more randomly, disrupting the overall order and clustering of the system.

The low-noise particles try to form clusters, but the presence of the high-noise particles makes it harder for the entire system to achieve a coherent structure.

In summary: The presence of high-noise particles disrupts the alignment and clustering behavior demonstrated by low-noise particles, resulting in a less coherent system.