FFR120: Homework 2 by Richard Blücher

Exercise 1

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

In [2]:
```
from functools import reduce

def phoretic_velocity(x, y, R, v0, r_c, L):
    """
    Function to calculate the phoretic velocity.

    Parameters
    ==========
    x, y : Positions.
    R : Particle radius.
    v0 : Phoretic reference velocity.
    r_c : Cut-off radius.
    L : Dimension of the squared arena.
    """

    N = np.size(x)

    vx = np.zeros(N)  # Phoretic velocity (x component).
    vy = np.zeros(N)  # Phoretic velocity (y component).

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

    for j in range(N - 1):

        # Check if replicas are needed to find the interacting neighbours.
        if np.size(np.where(replicas_needed == j)[0]):
            # Use replicas.
            xr, yr = replicas(x[j], y[j], L)
            for nr in range(9):
                dist2 = (x[j + 1:] - xr[nr]) ** 2 + (y[j + 1:] - yr[nr]) ** 2
                nn = np.where(dist2 <= r_c ** 2)[0] + j + 1

                # The list of nearest neighbours is set.
                # Contains only the particles with index > j

                if np.size(nn) > 0:
                    nn = nn.astype(int)

                    # Find interaction
                    dx = x[nn] - xr[nr]
                    dy = y[nn] - yr[nr]
```

```
                    dist = np.sqrt(dx ** 2 + dy ** 2)
                    v_p = v0 * R ** 2 / dist ** 2
                    dvx = dx / dist * v_p
                    dvy = dy / dist * v_p

                    # Contribution for particle j.
                    vx[j] += np.sum(dvx)
                    vy[j] += np.sum(dvy)

                    # Contribution for nn of particle j nr replica.
                    vx[nn] -= dvx
                    vy[nn] -= dvy

        else:
            dist2 = (x[j + 1:] - x[j]) ** 2 + (y[j + 1:] - y[j]) ** 2
            nn = np.where(dist2 <= r_c ** 2)[0] + j + 1

            # The list of nearest neighbours is set.
            # Contains only the particles with index > j

            if np.size(nn) > 0:
                nn = nn.astype(int)

                # Find interaction
                dx = x[nn] - x[j]
                dy = y[nn] - y[j]
                dist = np.sqrt(dx ** 2 + dy ** 2)
                v_p = v0 * R ** 2 / dist ** 2
                dvx = dx / dist * v_p
                dvy = dy / dist * v_p

                # Contribution for particle j.
                vx[j] += np.sum(dvx)
                vy[j] += np.sum(dvy)

                # Contribution for nn of particle j.
                vx[nn] -= dvx
                vy[nn] -= dvy

    return vx, vy


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 remove_overlap(x, y, R, L, dl, N_max_iter):
    """
    Function to remove the overlap between particles.
    Use the volume exclusion methods.
    If N_max_iter iterations are reached, then it stops.

    Parameters
    ==========
    x, y : Positions.
    R : Particle radius.
    L : Dimension of the squared arena.
    dl : Tolerance on the overlap. Must be much smaller than R.
    N_max_iter : stops if the number of iterations is larger than this.
    """

    N_part = np.size(x)
    step = 0
    running = True

    while running:

        n_overlaps = 0

        for i in np.arange(N_part):
            for j in np.arange(i + 1, N_part):
                # Check overlap.
                dx = x[j] - x[i]
                dy = y[j] - y[i]
                dist = np.sqrt(dx ** 2 + dy ** 2)

                if dist < 2 * R - dl:
                    n_overlaps += 1  # Increment overlap counter.
                    # Remove overlap.
```

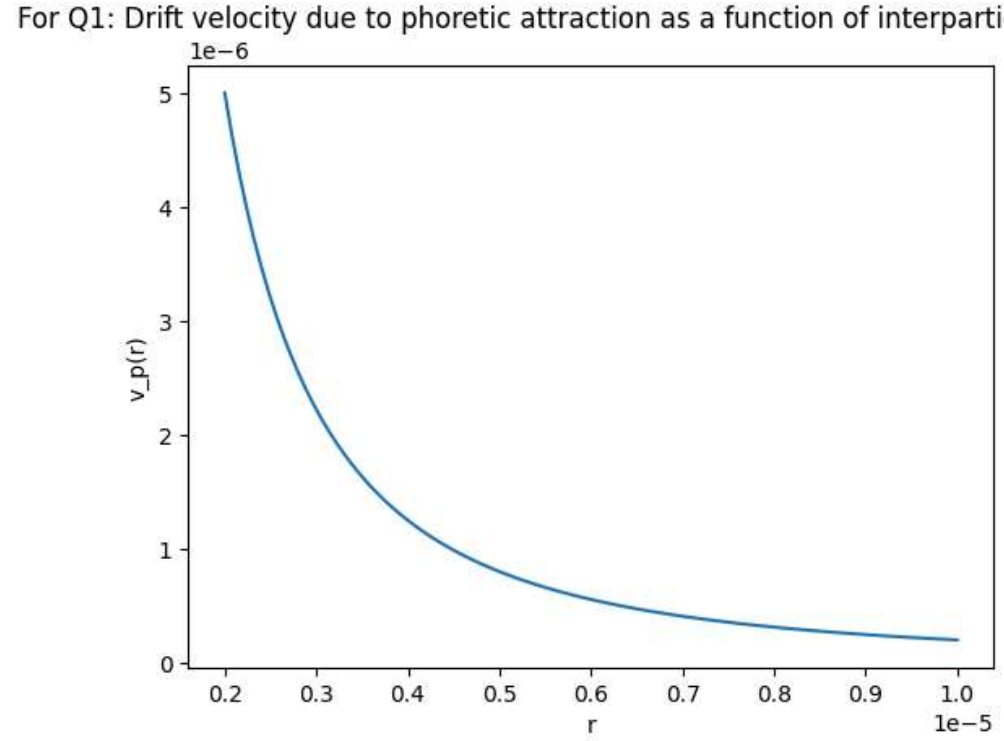
```
                    xm = 0.5 * (x[j] + x[i])
                    ym = 0.5 * (y[j] + y[i])
                    x[i] = xm - dx / dist * R
                    y[i] = ym - dy / dist * R
                    x[j] = xm + dx / dist * R
                    y[j] = ym + dy / dist * R

        step += 1

        if (step >= N_max_iter) or (n_overlaps == 0):
            running = False

    x, y = pbc(x, y, L)  # Apply periodic boundary conditions.

    return x, y
```

In [3]:
```
N_part = 200  # Number of active Brownian particles.

R = 1e-6  # Radius of the Brownian particle [m].
eta = 1e-3  # Viscosity of the medium.
gamma = 6 * np.pi * R * eta  # Drag coefficient.
gammaR = 8 * np.pi * R ** 3 * eta  # Rotational drag coefficient.
kBT = 4.11e-21  # kB*T at room temperature [J].
D = kBT / gamma  # Diffusion constant [m^2 / s].
DR = kBT / gammaR  # Rotational diffusion constant [1 / s].
t_r = 1 / DR  # Orientation relaxation time.
dt = 4e-2  # Time step [s].

v = 5e-6  # Self-propulsion speed [m/s].

v0 = 20e-6  # Phoretic reference speed [m/s].
r_c = 10 * R  # Cut-off radius [m].

L = 100 * R  # Side of the arena.

#print(f't_r={t_r:.3f} s')

# Initialization.

# Random position.
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]

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

# Coefficients for the finite difference solution.
c_noise_x = np.sqrt(2 * D * dt)
c_noise_y = np.sqrt(2 * D * dt)
c_noise_phi = np.sqrt(2 * DR * dt)

# Set vp = 0 from the start
vp_x = 0
vp_y = 0

# For plots
t_to_plot = [0, 5, 10, 20, 50]
N_max_steps = t_to_plot[-1]/dt
```

```
# For Q1
max_dist_per_step = (2*v+v0*(1/2))*dt
print(f'The maximum distance possible to travel in a time step with dt = {dt} is
print(f'Compare this with the radius R = {R}. The ratio d_max/R = {max_dist_per_

r_space = np.linspace(2*R, r_c, 100)
plt.plot(r_space, v0*(R**2/r_space**2))
plt.ylabel('v_p(r)')
plt.xlabel('r')
plt.title('For Q1: Drift velocity due to phoretic attraction as a function of in
```

```
The maximum distance possible to travel in a time step with dt = 0.04 is d_max =
8e-07.
Compare this with the radius R = 1e-06. The ratio d_max/R = 0.8 i.e. the particle
s' centers cent get closer to eachother than 0.2R.
```

Out[3]: `Text(0.5, 1.0, 'For Q1: Drift velocity due to phoretic attraction as a function of interparticle distance')`



Q1: We want the distance traveled every time step to be smaller than the radius R to not mess with the volume exclusion method. The maximum distance possible to travel in a time step happens when the speed is as high as possible. This happens when two particles are as close to eachother as possible (2R), making the speed $v_0\frac{R^2}{(2R)^2} = v_0\frac{1}{4}$. If the direction of the self propelling velocity v is also in the direciton of the other particle the speed of the particle is $v + v_0\frac{1}{4}$ in the direction of the other particle. If we now consider that both particles moves with this speed towards eachother they reduce their distance with $(2v + v_0\frac{1}{2})\Delta t$ every time step. So we want $(2v + v_0\frac{1}{2})\Delta t < R$. Otherwise one particle's centre clould pass the other's in one time step, making the volume exclusion method switch the particles positions (they pass through eachother).

As seen in the prints from the cell above, with $\Delta t = 0.04$ the maximum distance the particles can move towards eachother is 0.8R making it so that the particles centers can never pass. One could use a smaller Δt but the important part is that the maximum

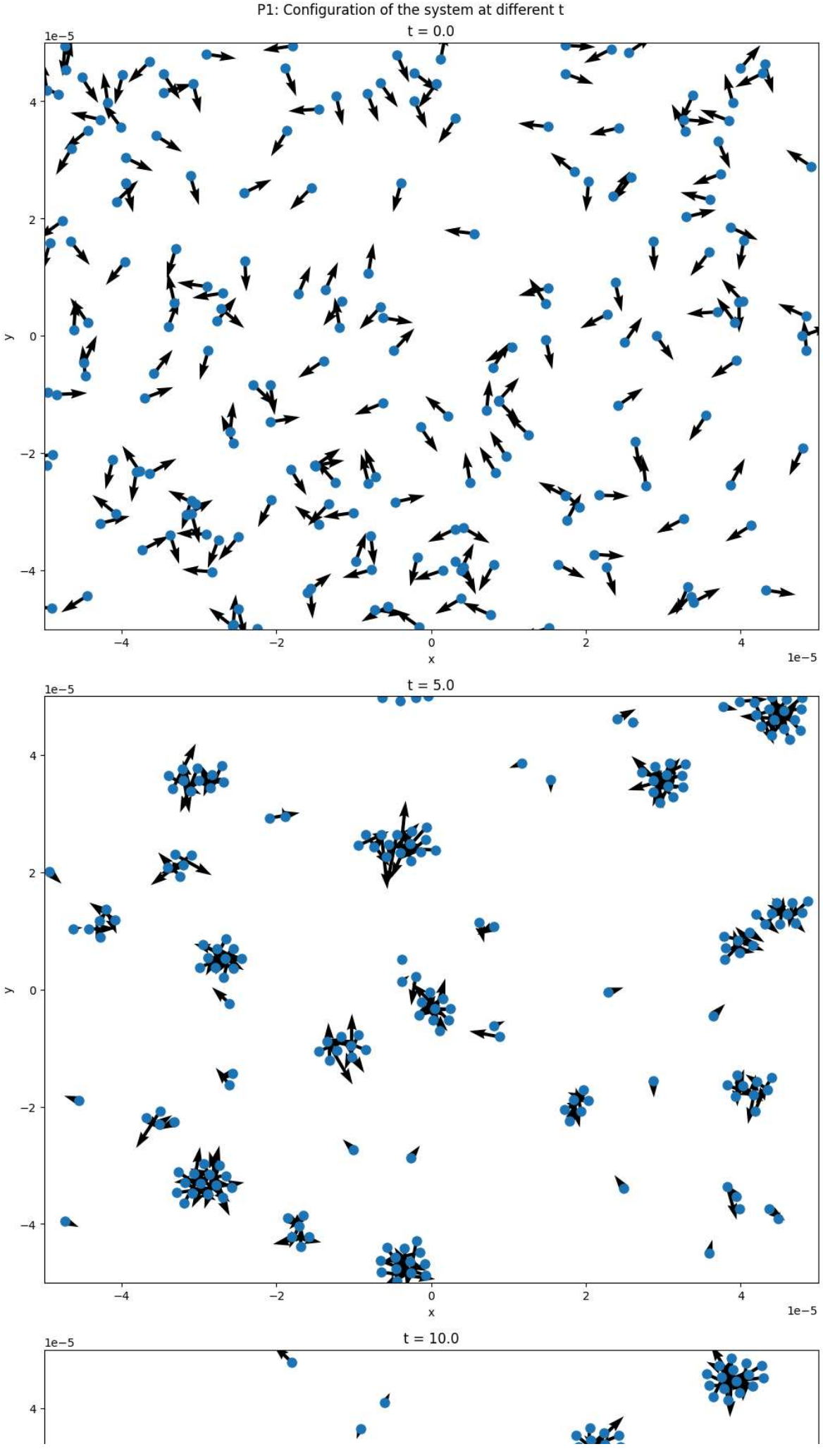
distance should be smaller than R so considering the computational cost of the simulation $\Delta t = 0.04$ is sufficient.

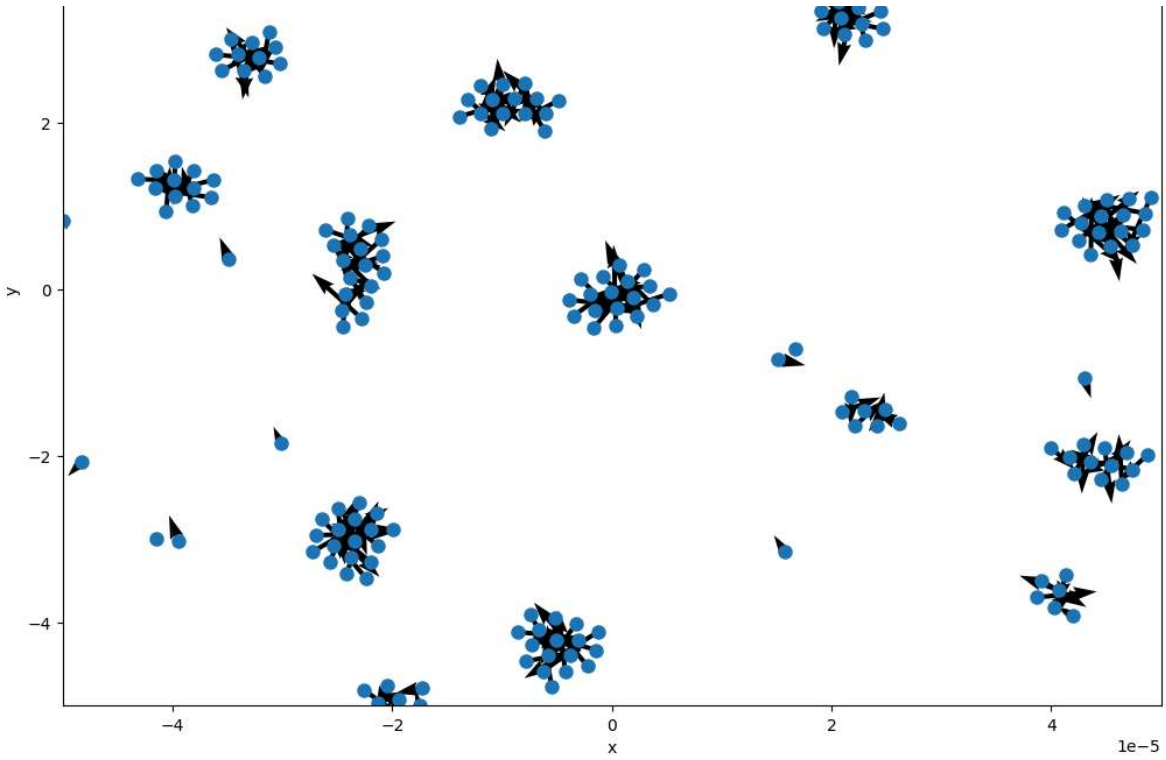
In [4]:
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *

visuals_on = False

window_size = 600

vp = 2 * R  # Length of the arrow indicating the velocity direction.
line_width = 1  # Width of the arrow line.

N_skip = 1

fig, ax = plt.subplots(nrows=5, ncols=1, figsize=(10, 40), layout='constrained')
i_fig = 0

if visuals_on:
    tk = Tk()
    tk.geometry(f'{window_size + 20}x{window_size + 20}')
    tk.configure(background='#000000')

    canvas = Canvas(tk, background='#ECECEC')  # Generate animation window
    tk.attributes('-topmost', 0)
    canvas.place(x=10, y=10, height=window_size, width=window_size)

    particles = []
    for j in range(N_part):
        particles.append(
            canvas.create_oval(
                (x[j] - R) / L * window_size + window_size / 2,
                (y[j] - R) / L * window_size + window_size / 2,
                (x[j] + R) / L * window_size + window_size / 2,
                (y[j] + R) / L * window_size + window_size / 2,
                outline='#808080',
                fill='#808080',
            )
        )

    velocities = []
    for j in range(N_part):
        velocities.append(
            canvas.create_line(
                x[j] / L * window_size + window_size / 2,
                y[j] / L * window_size + window_size / 2,
                (x[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                (y[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                width=line_width
            )
        )

step = 0

def stop_loop(event):
    global running
    running = False
if visuals_on:
```

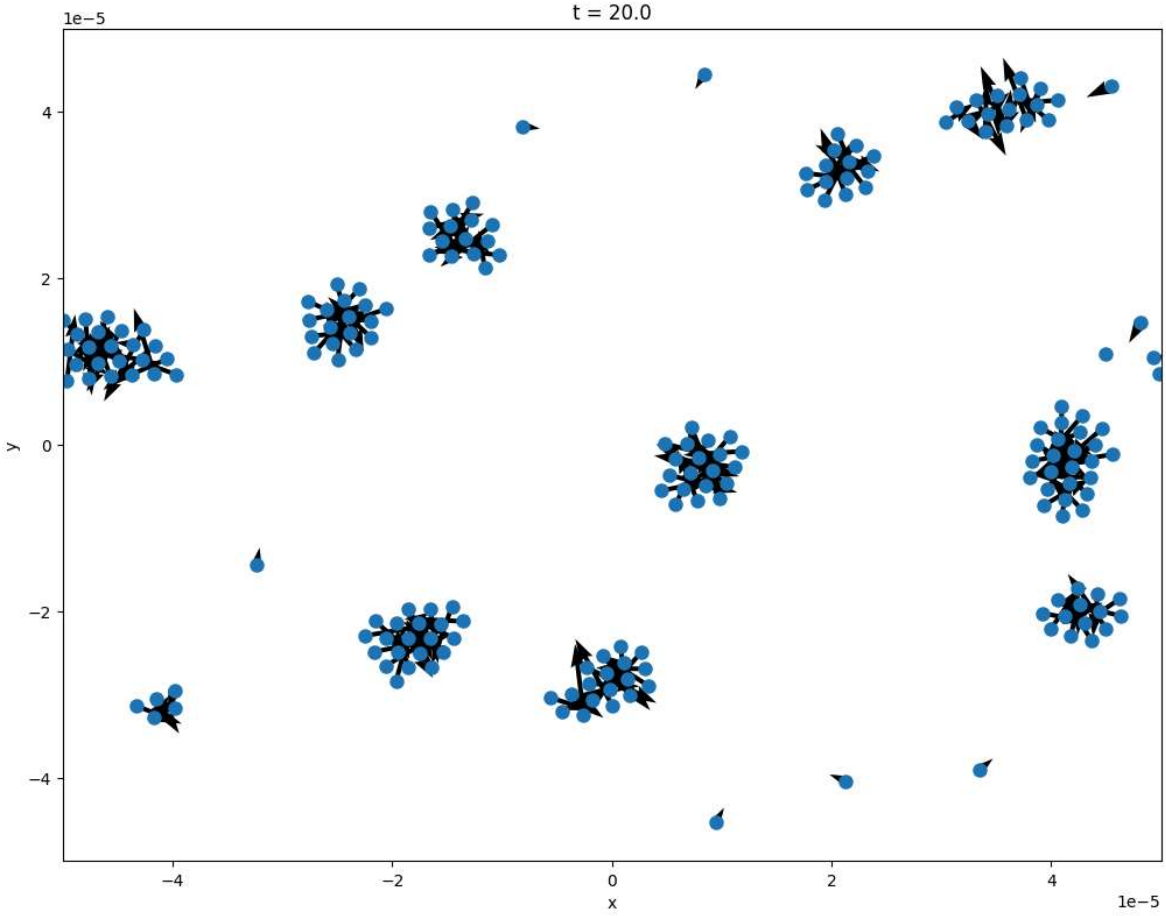
```
    tk.bind("<Escape>", stop_loop)  # Bind the Escape key to stop the loop.
running = True  # Flag to control the loop.
while running:

    # Check whether plot configuration.
    if step*dt in t_to_plot:
        #ax.clear()  # Clear previous plot.
        ax[i_fig].plot(x, y, '.', markersize=16)
        ax[i_fig].quiver(x, y, v*np.cos(phi)+vp_x, v*np.sin(phi)+vp_y)
        '''ax[i_fig].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[i_fig].set_xlim([-L / 2, L / 2])
        ax[i_fig].set_ylim([-L / 2, L / 2])
        ax[i_fig].set_title(f't = {step*dt}')
        ax[i_fig].set_xlabel('x')
        ax[i_fig].set_ylabel('y')
        #display(fig)  # Display updated plot.
        #clear_output(wait=True)  # Clear previous output.
        i_fig += 1

    # Calculate phoretic velocity.
    vp_x, vp_y = phoretic_velocity(x, y, R, v0, r_c, L)

    # Calculate new positions and orientations.
    nx = x + (v * np.cos(phi) + vp_x) * dt + c_noise_x * np.random.normal(0, 1,
    ny = y + (v * np.sin(phi) + vp_y) * dt + c_noise_y * np.random.normal(0, 1,
    nphi = phi + c_noise_phi * np.random.normal(0, 1, N_part)

    # Apply pbc.
    nx, ny = pbc(nx, ny, L)

    # Remove overlap.
    nx, ny = remove_overlap(nx, ny, R, L, dl=1e-8, N_max_iter=20)

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

    if visuals_on:
        # Update animation frame.
        if step % N_skip == 0:
            for j, particle in enumerate(particles):
                canvas.coords(
                    particle,
                    (nx[j] - R) / L * window_size + window_size / 2,
                    (ny[j] - R) / L * window_size + window_size / 2,
                    (nx[j] + R) / L * window_size + window_size / 2,
                    (ny[j] + R) / L * window_size + window_size / 2,
                )

            for j, velocity in enumerate(velocities):
                canvas.coords(
                    velocity,
                    nx[j] / L * window_size + window_size / 2,
                    ny[j] / L * window_size + window_size / 2,
                    (nx[j] + vp * np.cos(nphi[j])) / L * window_size + window_si
                    (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_si
                )

            tk.title(f'Time {step * dt:.1f} - Iteration {step}')
```

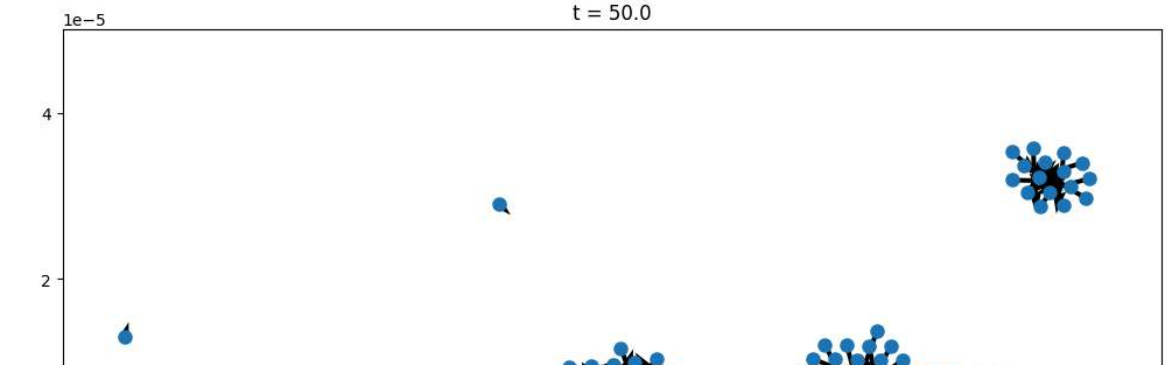
```
            tk.update_idletasks()
            tk.update()
            time.sleep(.001)  # Increase to slow down the simulation.









    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]
    if step == N_max_steps+1:
        running = False
if visuals_on:
    tk.update_idletasks()
    tk.update()
    tk.mainloop()  # Release animation handle (close window to finish).

fig.suptitle('P1: Configuration of the system at different t')
```

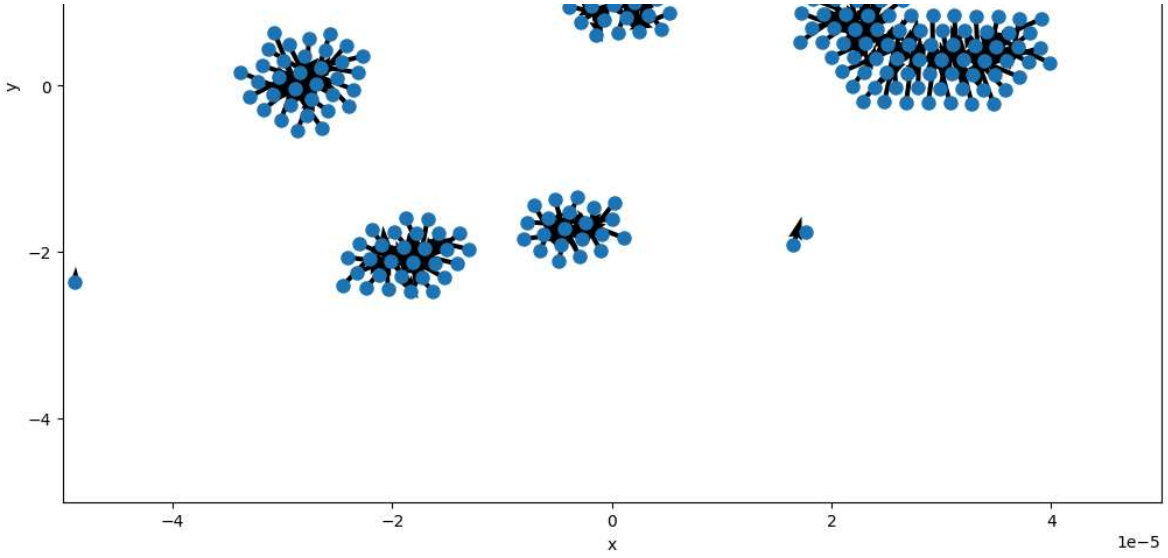
Out[4]: Text(0.5, 0.98, 'P1: Configuration of the system at different t')


P1: Configuration of the system at different t
t = 0.0
t = 5.0
t = 10.0
1e−5
x
y


2
0
−2
−4
y
−4
−2
0
2
4
x
1e−5


t = 20.0
1e−5
4
2
0
−2
−4
y
−4
−2
0
2
4
x
1e−5


t = 50.0
1e−5
4
2



Q2: At t = 0 the particles are scattred randomly with random directions. As time increases several clusters (or crystals) are formed and fewer and fewer particles are alone until almost every particle is in one of the fewer and fewer clusters. The direction of travel for the clustering particles are towards the cluster.

In [5]:
```
# Change L to 50R
N_part = 200  # Number of active Brownian particles.

R = 1e-6  # Radius of the Brownian particle [m].
eta = 1e-3  # Viscosity of the medium.
gamma = 6 * np.pi * R * eta  # Drag coefficient.
gammaR = 8 * np.pi * R ** 3 * eta  # Rotational drag coefficient.
kBT = 4.11e-21  # kB*T at room temperature [J].
D = kBT / gamma  # Diffusion constant [m^2 / s].
DR = kBT / gammaR  # Rotational diffusion constant [1 / s].
t_r = 1 / DR  # Orientation relaxation time.
dt = 4e-2  # Time step [s].

v = 5e-6  # Self-propulsion speed [m/s].

v0 = 20e-6  # Phoretic reference speed [m/s].
r_c = 10 * R  # Cut-off radius [m].

L = 50 * R  # Side of the arena.

print(f't_r={t_r:.3f} s')

# Initialization.

# Random position.
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]

x, y = remove_overlap(x, y, R, L, dl=1e-8, N_max_iter=20)

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

# Coefficients for the finite difference solution.
c_noise_x = np.sqrt(2 * D * dt)
c_noise_y = np.sqrt(2 * D * dt)
```

```
c_noise_phi = np.sqrt(2 * DR * dt)

# Set vp = 0 from the start
vp_x = 0
vp_y = 0
```

t_r=6.115 s

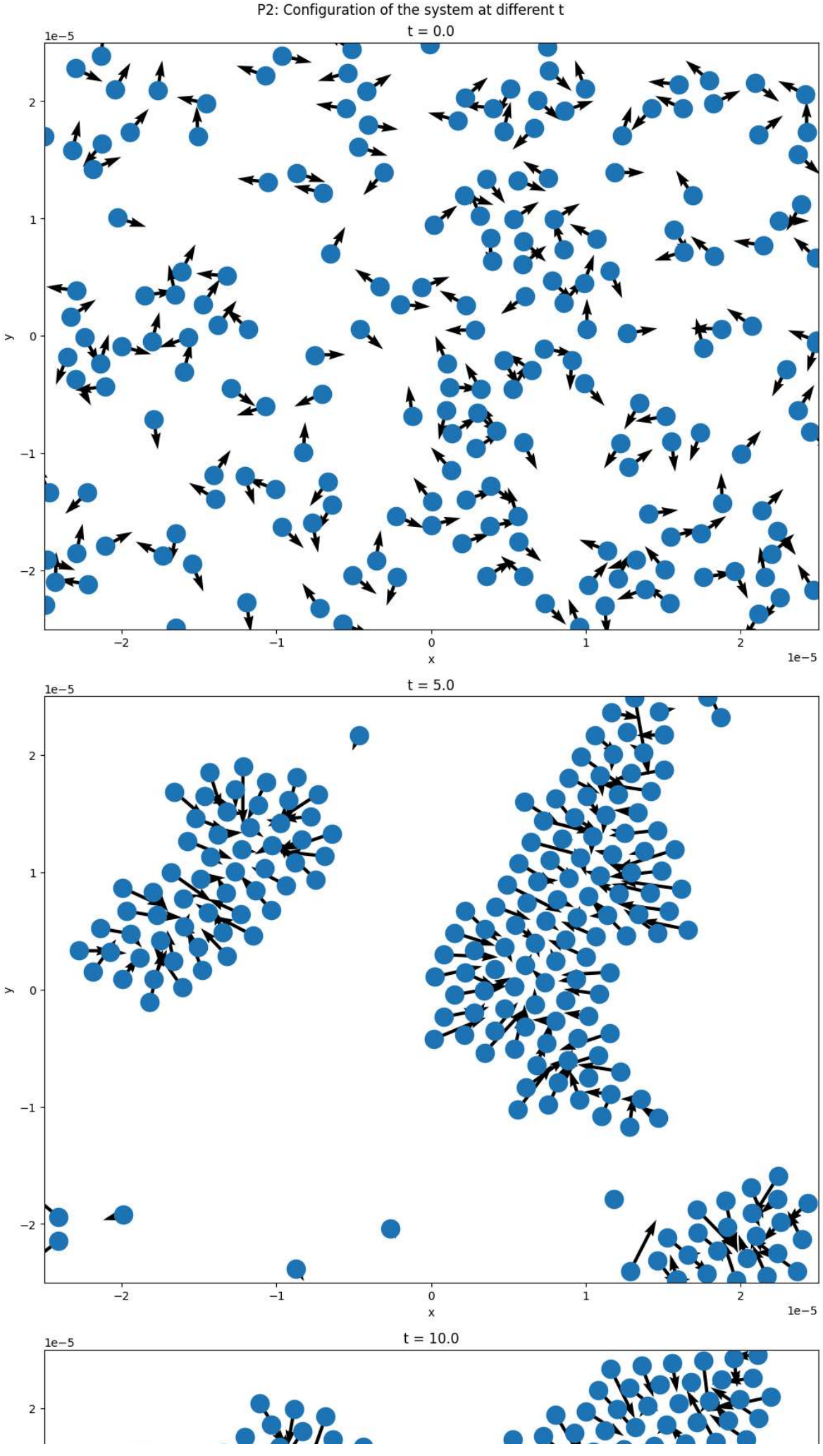
In [6]:

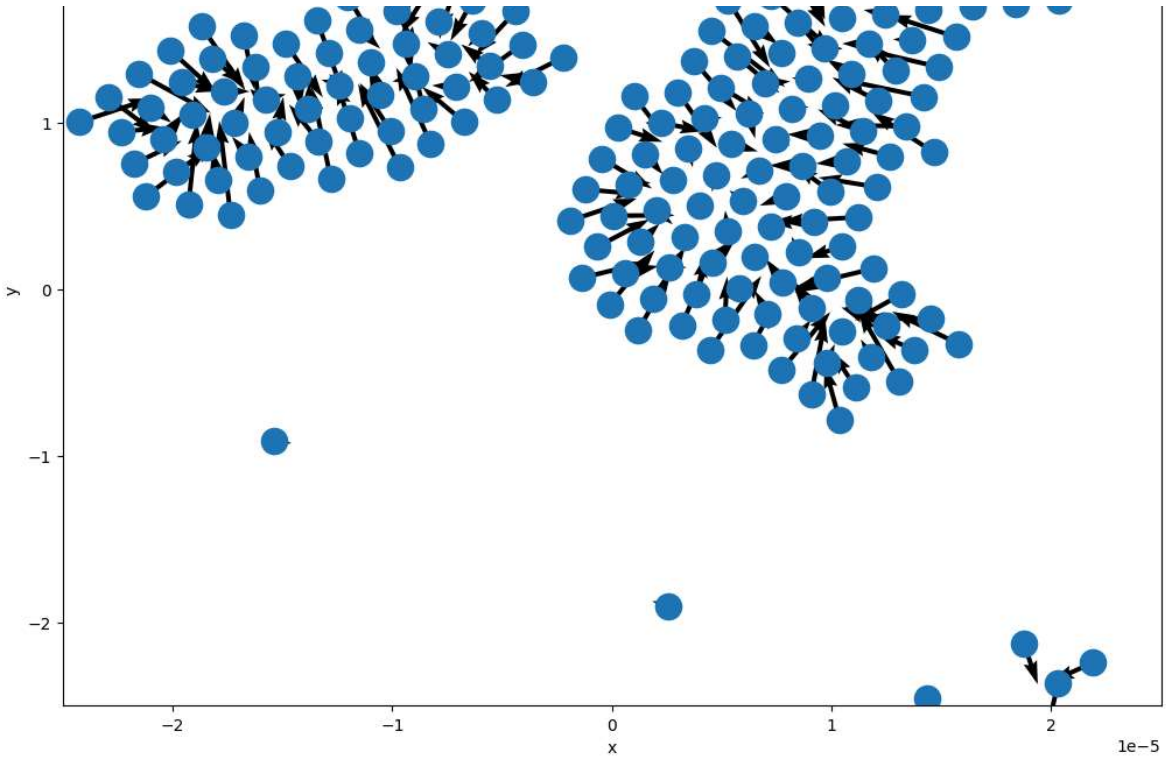
```
visuals_on = False

window_size = 600

vp = 2 * R  # Length of the arrow indicating the velocity direction.
line_width = 1  # Width of the arrow line.

N_skip = 1

fig, ax = plt.subplots(nrows=5, ncols=1, figsize=(10, 40), layout='constrained')
i_fig = 0

if visuals_on:
    tk = Tk()
    tk.geometry(f'{window_size + 20}x{window_size + 20}')
    tk.configure(background='#000000')

    canvas = Canvas(tk, background='#ECECEC')  # Generate animation window
    tk.attributes('-topmost', 0)
    canvas.place(x=10, y=10, height=window_size, width=window_size)

    particles = []
    for j in range(N_part):
        particles.append(
            canvas.create_oval(
                (x[j] - R) / L * window_size + window_size / 2,
                (y[j] - R) / L * window_size + window_size / 2,
                (x[j] + R) / L * window_size + window_size / 2,
                (y[j] + R) / L * window_size + window_size / 2,
                outline='#808080',
                fill='#808080',
            )
        )

    velocities = []
    for j in range(N_part):
        velocities.append(
            canvas.create_line(
                x[j] / L * window_size + window_size / 2,
                y[j] / L * window_size + window_size / 2,
                (x[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                (y[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                width=line_width
            )
        )

step = 0

def stop_loop(event):
    global running
    running = False
if visuals_on:
```

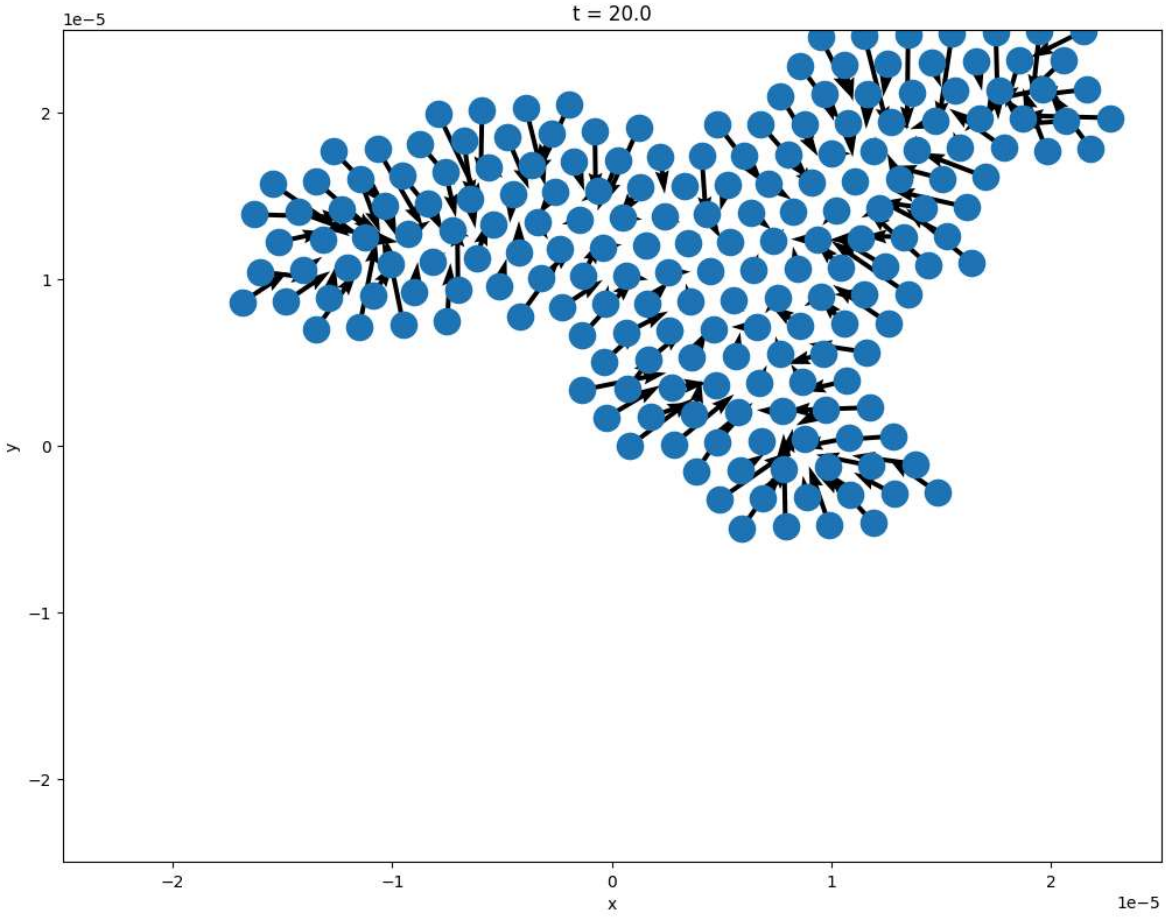
```
    tk.bind("<Escape>", stop_loop)  # Bind the Escape key to stop the loop.
running = True  # Flag to control the loop.
while running:

    # Check whether plot configuration.
    if step*dt in t_to_plot:
        #ax.clear()  # Clear previous plot.
        ax[i_fig].plot(x, y, '.', markersize=32)
        ax[i_fig].quiver(x, y, v*np.cos(phi)+vp_x, v*np.sin(phi)+vp_y)
        '''ax[i_fig].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[i_fig].set_xlim([-L / 2, L / 2])
        ax[i_fig].set_ylim([-L / 2, L / 2])
        ax[i_fig].set_title(f't = {step*dt}')
        ax[i_fig].set_xlabel('x')
        ax[i_fig].set_ylabel('y')
        #display(fig)  # Display updated plot.
        #clear_output(wait=True)  # Clear previous output.
        i_fig += 1

    # Calculate phoretic velocity.
    vp_x, vp_y = phoretic_velocity(x, y, R, v0, r_c, L)

    # Calculate new positions and orientations.
    nx = x + (v * np.cos(phi) + vp_x) * dt + c_noise_x * np.random.normal(0, 1,
    ny = y + (v * np.sin(phi) + vp_y) * dt + c_noise_y * np.random.normal(0, 1,
    nphi = phi + c_noise_phi * np.random.normal(0, 1, N_part)

    # Apply pbc.
    nx, ny = pbc(nx, ny, L)

    # Remove overlap.
    nx, ny = remove_overlap(nx, ny, R, L, dl=1e-8, N_max_iter=20)

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

    if visuals_on:
        # Update animation frame.
        if step % N_skip == 0:
            for j, particle in enumerate(particles):
                canvas.coords(
                    particle,
                    (nx[j] - R) / L * window_size + window_size / 2,
                    (ny[j] - R) / L * window_size + window_size / 2,
                    (nx[j] + R) / L * window_size + window_size / 2,
                    (ny[j] + R) / L * window_size + window_size / 2,
                )

            for j, velocity in enumerate(velocities):
                canvas.coords(
                    velocity,
                    nx[j] / L * window_size + window_size / 2,
                    ny[j] / L * window_size + window_size / 2,
                    (nx[j] + vp * np.cos(nphi[j])) / L * window_size + window_si
                    (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_si
                )

            tk.title(f'Time {step * dt:.1f} - Iteration {step}')
```

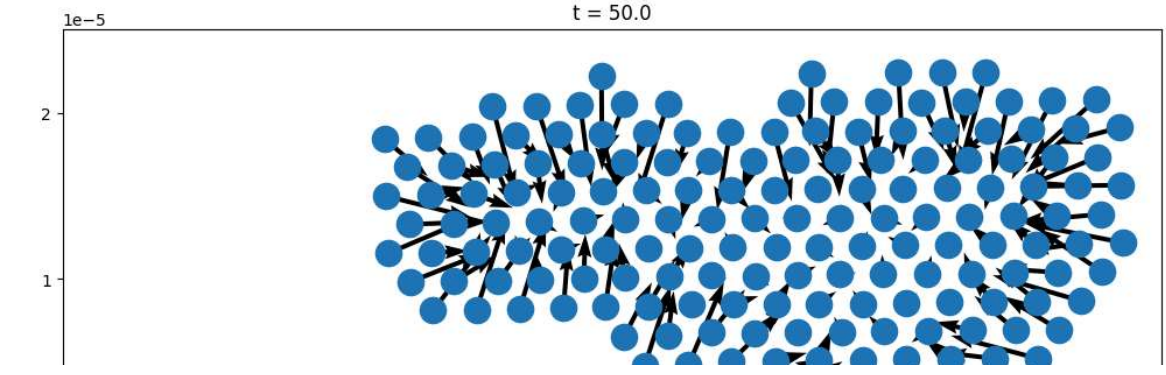
```
            tk.update_idletasks()
            tk.update()
            time.sleep(.001)  # Increase to slow down the simulation.









    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]
    if step == N_max_steps+1:
        running = False
if visuals_on:
    tk.update_idletasks()
    tk.update()
    tk.mainloop()  # Release animation handle (close window to finish).

fig.suptitle('P2: Configuration of the system at different t')
```

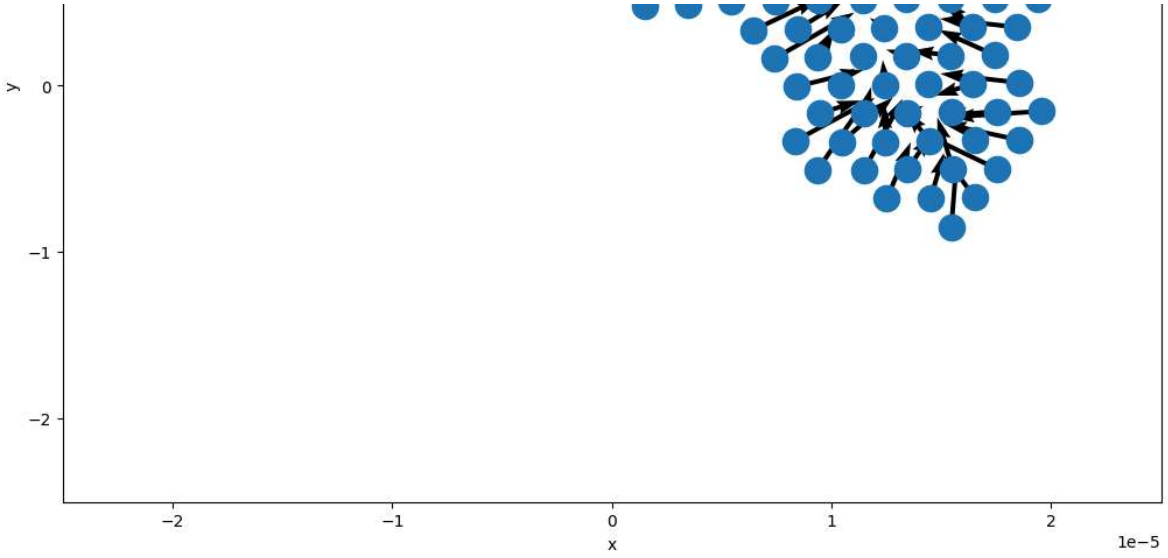
Out[6]: Text(0.5, 0.98, 'P2: Configuration of the system at different t')


P2: Configuration of the system at different t
t = 0.0
t = 5.0
t = 10.0
1e−5
x
y


1
0
−1
−2
y
−2
−1
0
1
2
x
1e−5


t = 20.0
1e−5
2
1
0
−1
−2
y
−2
−1
0
1
2
x
1e−5


t = 50.0
1e−5
2
1



Q3: At t = 0 the particles are scattred randomly with random directions as in P1 with the differance that the particles are more densly packed. As time increases several clusters (or crystals) are formed and fewer and fewer particles are alone until almost every particle is in a singe big cluster. The direction of travel for the clustering particles are towards the cluster. In this plot it is also easier to see that the drift from the phoretic force is small for particles in the center of clusters because the forces cancel from different directions. The main differance between P1 and P2 are that bigger clusters form earlier in P2.

Exercise 2

In [7]:
```
from functools import reduce

def calculate_intensity(x, y, I0, r0, L, r_c):
    """
    Function to calculate the intensity seen by each particle.

    Parameters
    ==========
    x, y : Positions.
    r0 : Standard deviation of the Gaussian light intensity zone.
    I0 : Maximum intensity of the Gaussian.
    L : Dimension of the squared arena.
    r_c : Cut-off radius. Pre-set it around 3 * r0.
    """

    N = np.size(x)

    I_particle = np.zeros(N)  # Intensity seen by each particle.

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

```
    for j in range(N - 1):

        # Check if replicas are needed to find the interacting neighbours.
        if np.size(np.where(replicas_needed == j)[0]):
            # Use replicas.
            xr, yr = replicas(x[j], y[j], L)
            for nr in range(9):
                dist2 = (x[j + 1:] - xr[nr]) ** 2 + (y[j + 1:] - yr[nr]) ** 2
                nn = np.where(dist2 <= r_c ** 2)[0] + j + 1

                # The list of nearest neighbours is set.
                # Contains only the particles with index > j

                if np.size(nn) > 0:
                    nn = nn.astype(int)

                    # Find total intensity
                    dx = x[nn] - xr[nr]
                    dy = y[nn] - yr[nr]
                    d2 = dx ** 2 + dy ** 2
                    I = I0 * np.exp(- d2 / r0 ** 2)

                    # Contribution for particle j.
                    I_particle[j] += np.sum(I)

                    # Contribution for nn of particle j nr replica.
                    I_particle[nn] += I

        else:
            dist2 = (x[j + 1:] - x[j]) ** 2 + (y[j + 1:] - y[j]) ** 2
            nn = np.where(dist2 <= r_c ** 2)[0] + j + 1

            # The list of nearest neighbours is set.
            # Contains only the particles with index > j

            if np.size(nn) > 0:
                nn = nn.astype(int)

                # Find interaction
                dx = x[nn] - x[j]
                dy = y[nn] - y[j]
                d2 = dx ** 2 + dy ** 2
                I = I0 * np.exp(- d2 / r0 ** 2)

                # Contribution for particle j.
                I_particle[j] += np.sum(I)

                # Contribution for nn of particle j.
                I_particle[nn] += I

    return I_particle
```

In [111…

```
N_part = 50  # Number of light-sensitive robots.
# Note: 5 is enough to demonstrate clustering - dispersal.

tau = 1  # Timescale of the orientation diffusion.
dt = 0.05  # Time step [s].

v0 = 0.1  # Self-propulsion speed at I=0 [m/s].
```

```
v_inf = 0.01  # Self-propulsion speed at I=+infty [m/s].
Ic = 0.1  # Intensity scale where the speed decays.
I0 = 1  # Maximum intensity.
r0 = 0.3  # Standard deviation of the Gaussian light intensity zone [m].

# delta = 0  # No delay. Tends to cluster.
delta = 5 * tau  # Positive delay. More stable clustering.
# delta = - 5 * tau  # Negative delay. Dispersal.

r_c = 4 * r0  # Cut-off radius [m].
L = 50 * r0  # Side of the arena[m].

# Initialization.

# Random position.
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]

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

# Coefficients for the finite difference solution.
c_noise_phi = np.sqrt(2 * dt / tau)

n_fit = 5
if delta < 0:
    # Negative delay.
    n_fit = 5
    I_fit = np.zeros([n_fit, N_part])
    t_fit = np.arange(n_fit) * dt
    dI_dt = np.zeros(N_part)
    # Initialize.
    I_ref = I0 * np.exp(- (x ** 2 + y ** 2) / r0 ** 2)
    for i in range(n_fit):
        I_fit[i, :] += I_ref

if delta > 0:
    # Positive delay.
    n_delay = int(delta / dt)  # Delay in units of time steps.
    I_memory = np.zeros([n_delay, N_part])
    # Initialize.
    I_ref = I0 * np.exp(- (x ** 2 + y ** 2) / r0 ** 2)
    for i in range(n_fit):
        I_memory[i, :] += I_ref



# For plots
t_to_plot = [0, 10, 100, 500, 1000]
N_max_steps = t_to_plot[-1]/dt
```

In [112…

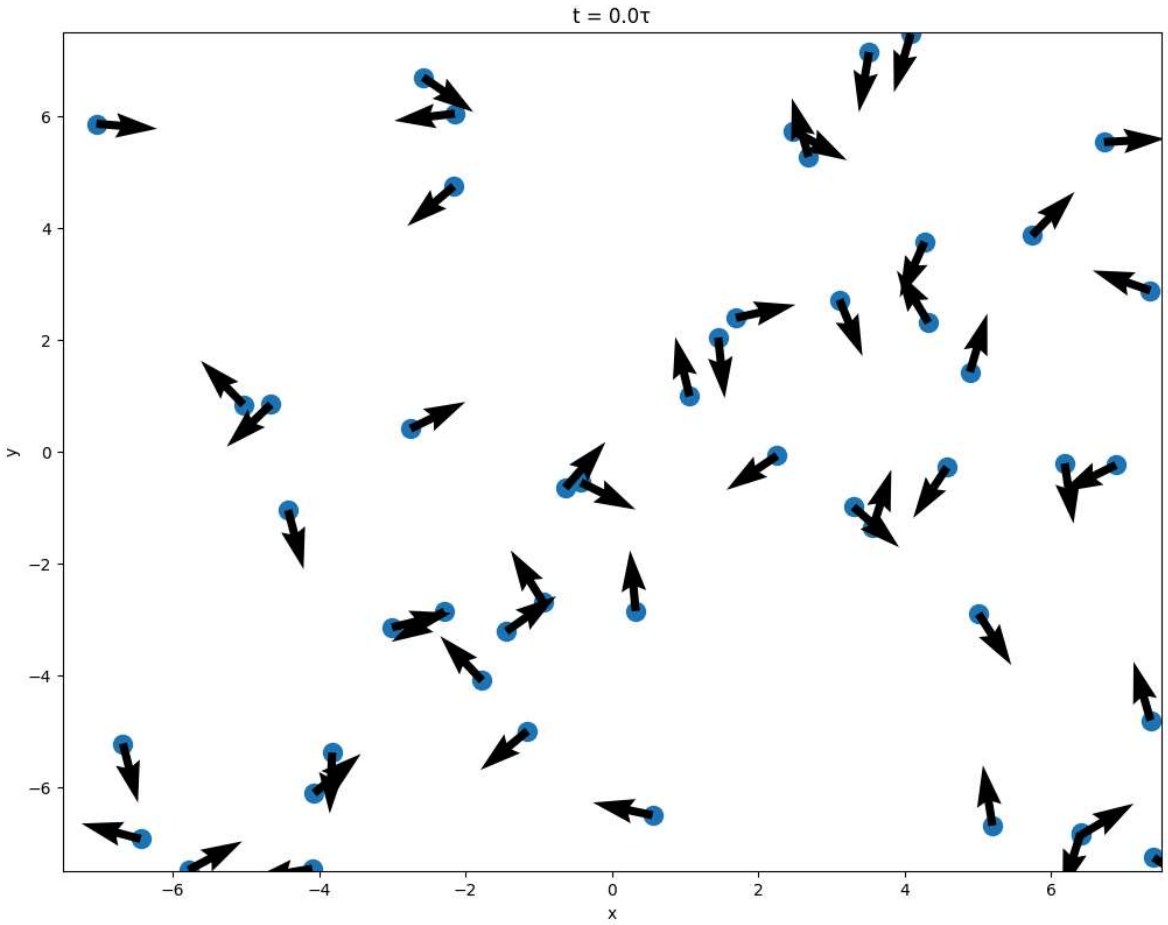
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *

visuals_on = False

fig, ax = plt.subplots(nrows=5, ncols=1, figsize=(10, 40), layout='constrained')
i_fig = 0
points_whole_ax = 5 * 0.8 * 72    # 1 point = dpi / 72 pixels
```

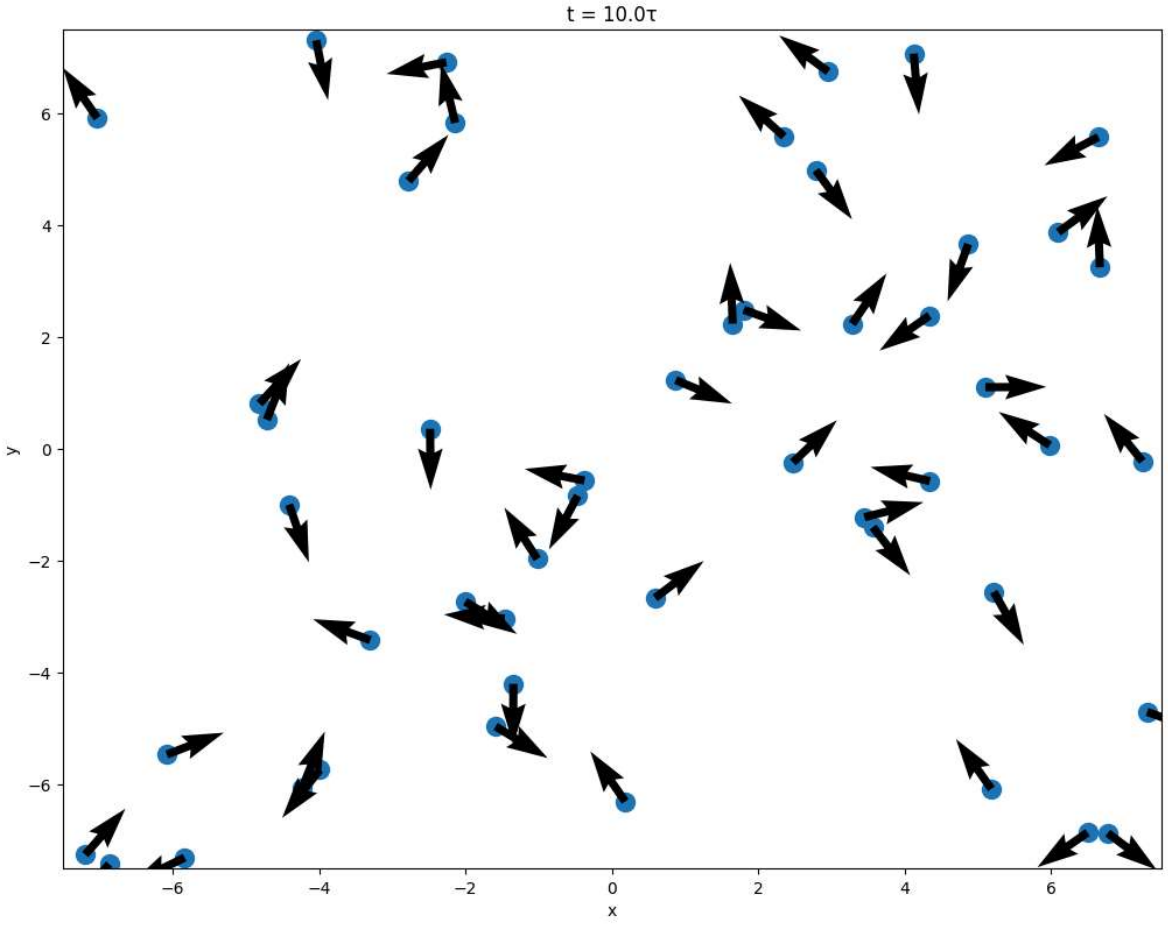
```
points_radius = 2 * r0 / L * points_whole_ax

window_size = 600

rp = r0 / 3
vp = rp  # Length of the arrow indicating the velocity direction.
line_width = 1  # Width of the arrow line.

N_skip = 2

if visuals_on:
    tk = Tk()
    tk.geometry(f'{window_size + 20}x{window_size + 20}')
    tk.configure(background='#000000')

    canvas = Canvas(tk, background='#ECECEC')  # Generate animation window
    tk.attributes('-topmost', 0)
    canvas.place(x=10, y=10, height=window_size, width=window_size)

    light_spots = []
    for j in range(N_part):
        light_spots.append(
            canvas.create_oval(
                (x[j] - r0) / L * window_size + window_size / 2,
                (y[j] - r0) / L * window_size + window_size / 2,
                (x[j] + r0) / L * window_size + window_size / 2,
                (y[j] + r0) / L * window_size + window_size / 2,
                outline='#FF8080',
            )
        )

    particles = []
    for j in range(N_part):
        particles.append(
            canvas.create_oval(
                (x[j] - rp) / L * window_size + window_size / 2,
                (y[j] - rp) / L * window_size + window_size / 2,
                (x[j] + rp) / L * window_size + window_size / 2,
                (y[j] + rp) / L * window_size + window_size / 2,
                outline='#000000',
                fill='#A0A0A0',
            )
        )

    velocities = []
    for j in range(N_part):
        velocities.append(
            canvas.create_line(
                x[j] / L * window_size + window_size / 2,
                y[j] / L * window_size + window_size / 2,
                (x[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                (y[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                width=line_width,
            )
        )

step = 0

def stop_loop(event):
    global running
```

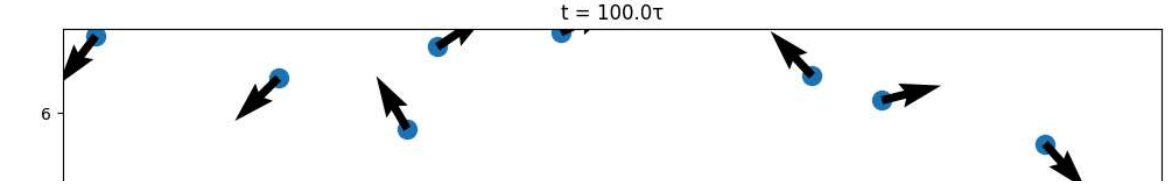
```
    running = False

if visuals_on:
    tk.bind("<Escape>", stop_loop)  # Bind the Escape key to stop the loop.
running = True  # Flag to control the loop.
while running:

    # Check whether plot configuration.
    if step*dt in t_to_plot:
        print(step*dt)
        if step*dt == 100:
            x1_100 = x
            y1_100 = y
        if step*dt == 500:
            x1_500 = x
            y1_500 = y
        #ax.clear()  # Clear previous plot.
        ax[i_fig].scatter(x, y, s=points_radius**2)
        ax[i_fig].quiver(x, y, np.cos(phi), np.sin(phi))
        '''ax[i_fig].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[i_fig].set_xlim([-L / 2, L / 2])
        ax[i_fig].set_ylim([-L / 2, L / 2])
        ax[i_fig].set_title(f't = {step*dt}τ')
        ax[i_fig].set_xlabel('x')
        ax[i_fig].set_ylabel('y')
        #display(fig)  # Display updated plot.
        #clear_output(wait=True)  # Clear previous output.
        i_fig += 1

    # Calculate current I.
    I_particles = calculate_intensity(x, y, I0, r0, L, r_c)

    if delta < 0:
        # Estimate the derivative of I linear using the last n_fit values.
        for i in range(N_part - 1):
            # Update I_fit.
            I_fit = np.roll(I_fit, -1, axis=0)
            I_fit[-1, :] = I_particles
            # Fit to determine the slope.
            for j in range(N_part):
                p = np.polyfit(t_fit, I_fit[:, j], 1)
                dI_dt[j] = p[0]
            # Determine forecast. Remember that here delta is negative.
            I = I_particles - delta * dI_dt
            I[np.where(I < 0)[0]] = 0
    elif delta > 0:
        # Update I_memory.
        I_memory = np.roll(I_memory, -1, axis=0)
        I_memory[-1, :] = I_particles
        I = I_memory[0, :]
    else:
        I = I_particles

    # Calculate new positions and orientations.
    v = v_inf + (v0 - v_inf) * np.exp(- I / Ic)
    nx = x + v * dt * np.cos(phi)
    ny = y + v * dt * np.sin(phi)
    nphi = phi + c_noise_phi * np.random.normal(0, 1, N_part)
```

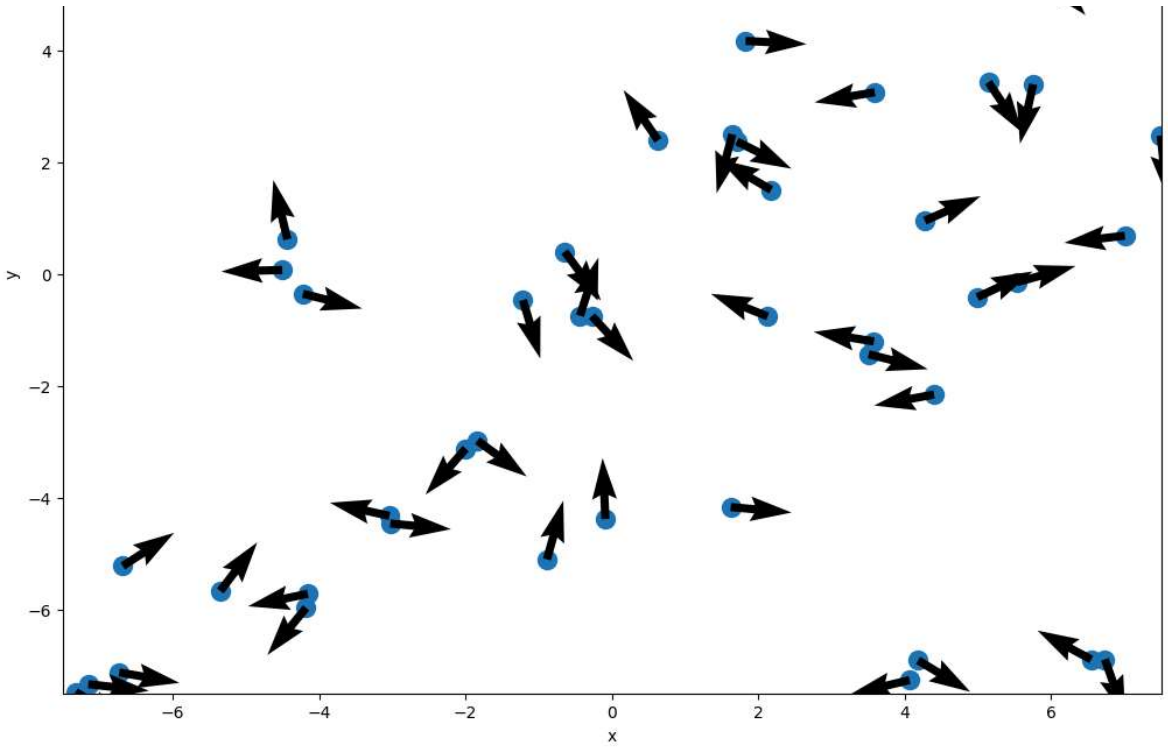
```
    # Apply pbc.
    nx, ny = pbc(nx, ny, L)

    if visuals_on:
        # Update animation frame.
        if step % N_skip == 0:

            for j, light_spot in enumerate(light_spots):
                canvas.coords(
                    light_spot,
                    (nx[j] - r0) / L * window_size + window_size / 2,
                    (ny[j] - r0) / L * window_size + window_size / 2,
                    (nx[j] + r0) / L * window_size + window_size / 2,
                    (ny[j] + r0) / L * window_size + window_size / 2,
                )

            for j, particle in enumerate(particles):
                canvas.coords(
                    particle,
                    (nx[j] - rp) / L * window_size + window_size / 2,
                    (ny[j] - rp) / L * window_size + window_size / 2,
                    (nx[j] + rp) / L * window_size + window_size / 2,
                    (ny[j] + rp) / L * window_size + window_size / 2,
                )

            for j, velocity in enumerate(velocities):
                canvas.coords(
                    velocity,
                    nx[j] / L * window_size + window_size / 2,
                    ny[j] / L * window_size + window_size / 2,
                    (nx[j] + vp * np.cos(nphi[j])) / L * window_size + window_si
                    (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_si
                )

            tk.title(f'Time {step * dt:.1f} - Iteration {step}')
            tk.update_idletasks()
            tk.update()
            time.sleep(.001)  # Increase to slow down the simulation.

    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]

    if step == N_max_steps+1:
        running = False

if visuals_on:
    tk.update_idletasks()
    tk.update()
    tk.mainloop()  # Release animation handle (close window to finish).
```

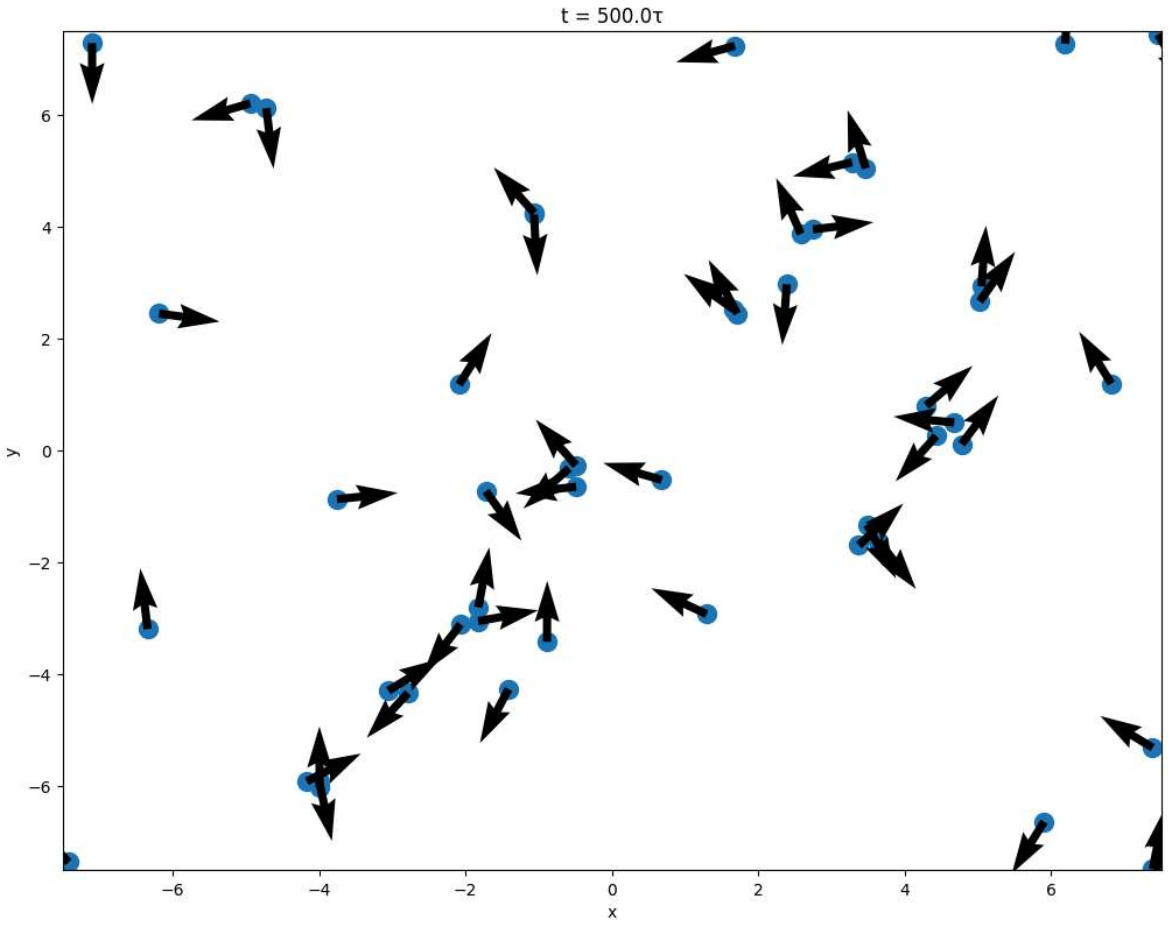
```
0.0
10.0
100.0
500.0
1000.0
```

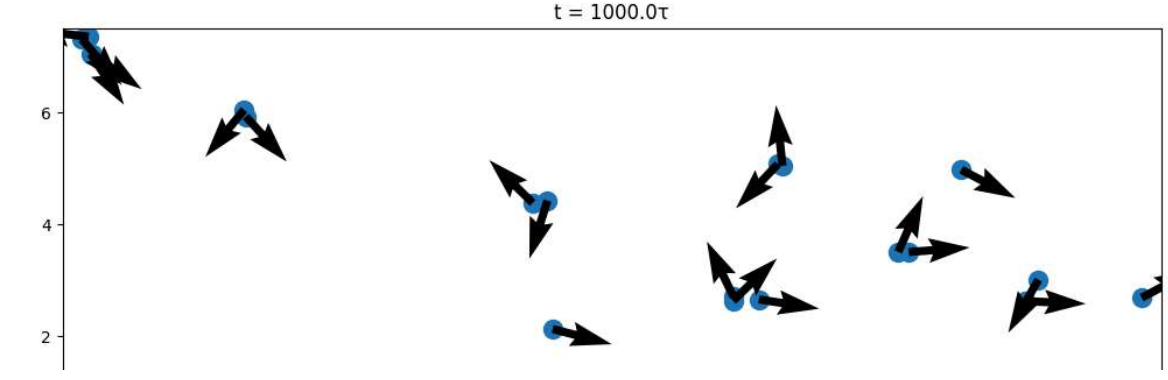

t = 0.0τ
x
y

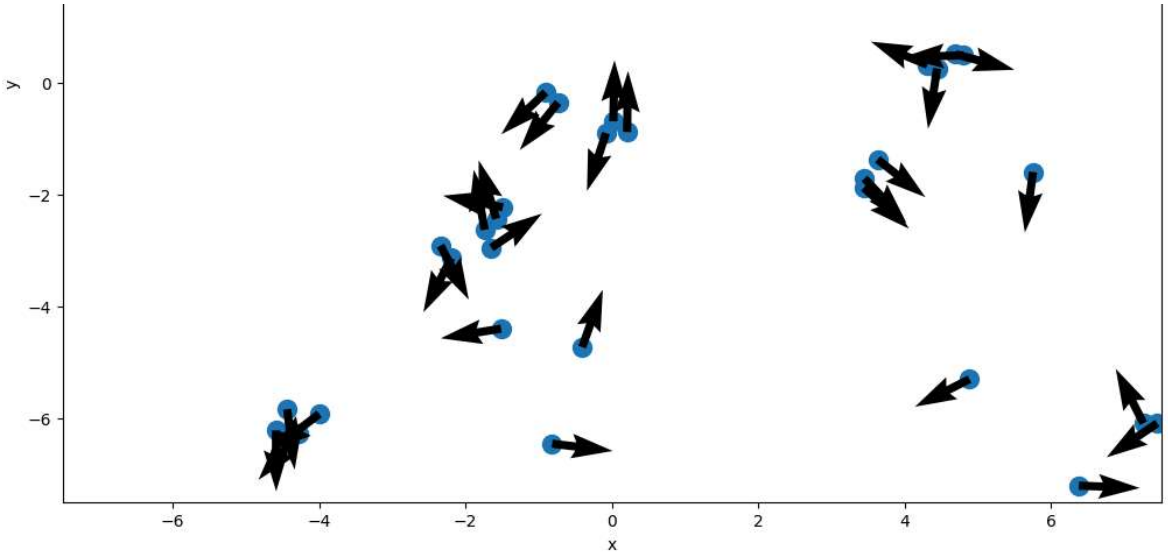

t = 10.0τ
x
y


t = 100.0τ


y
x


t = 500.0τ
y
x


t = 1000.0τ



In [261…

```
N_part = 50  # Number of light-sensitive robots.
# Note: 5 is enough to demonstrate clustering - dispersal.

tau = 1  # Timescale of the orientation diffusion.
dt = 0.05  # Time step [s].

v0 = 0.1  # Self-propulsion speed at I=0 [m/s].
v_inf = 0.01  # Self-propulsion speed at I=+infty [m/s].
Ic = 0.1  # Intensity scale where the speed decays.
I0 = 1  # Maximum intensity.
r0 = 0.3  # Standard deviation of the Gaussian light intensity zone [m].

# delta = 0  # No delay. Tends to cluster.
# delta = 5 * tau  # Positive delay. More stable clustering.
delta = - 5 * tau  # Negative delay. Dispersal.

r_c = 4 * r0  # Cut-off radius [m].
L = 50 * r0  # Side of the arena[m].

# Initialization.

# Random position.
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]

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

# Coefficients for the finite difference solution.
c_noise_phi = np.sqrt(2 * dt / tau)


if delta < 0:
    # Negative delay.
    n_fit = 5
    I_fit = np.zeros([n_fit, N_part])
    t_fit = np.arange(n_fit) * dt
    dI_dt = np.zeros(N_part)
    # Initialize.
    I_ref = I0 * np.exp(- (x ** 2 + y ** 2) / r0 ** 2)
    for i in range(n_fit):
        I_fit[i, :] += I_ref
```

```
if delta > 0:
    # Positive delay.
    n_delay = int(delta / dt)  # Delay in units of time steps.
    I_memory = np.zeros([n_delay, N_part])
    # Initialize.
    I_ref = I0 * np.exp(- (x ** 2 + y ** 2) / r0 ** 2)
    for i in range(n_fit):
        I_memory[i, :] += I_ref



# For plots
t_to_plot = [0, 10, 100, 500]
N_max_steps = t_to_plot[-1]/dt
```

In [262…

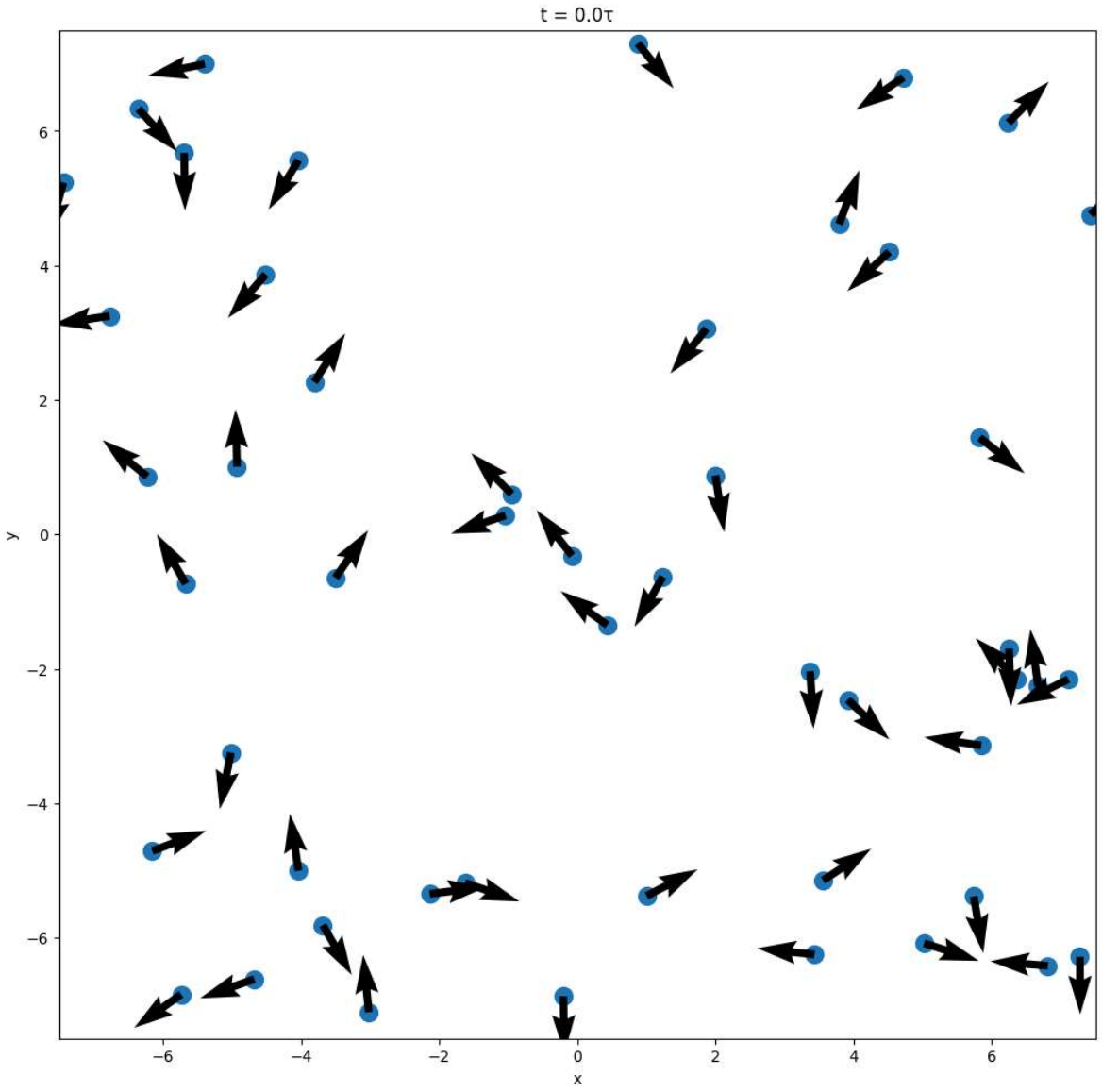
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *

visuals_on = False

fig, ax = plt.subplots(nrows=4, ncols=1, figsize=(10, 40), layout='constrained')
i_fig = 0
points_whole_ax = 5 * 0.8 * 72    # 1 point = dpi / 72 pixels
points_radius = 2 * r0 / L * points_whole_ax


window_size = 600

rp = r0 / 3
vp = rp  # Length of the arrow indicating the velocity direction.
line_width = 1  # Width of the arrow line.

N_skip = 2

if visuals_on:
    tk = Tk()
    tk.geometry(f'{window_size + 20}x{window_size + 20}')
    tk.configure(background='#000000')

    canvas = Canvas(tk, background='#ECECEC')  # Generate animation window
    tk.attributes('-topmost', 0)
    canvas.place(x=10, y=10, height=window_size, width=window_size)

    light_spots = []
    for j in range(N_part):
        light_spots.append(
            canvas.create_oval(
                (x[j] - r0) / L * window_size + window_size / 2,
                (y[j] - r0) / L * window_size + window_size / 2,
                (x[j] + r0) / L * window_size + window_size / 2,
                (y[j] + r0) / L * window_size + window_size / 2,
                outline='#FF8080',
            )
        )

    particles = []
    for j in range(N_part):
        particles.append(
```

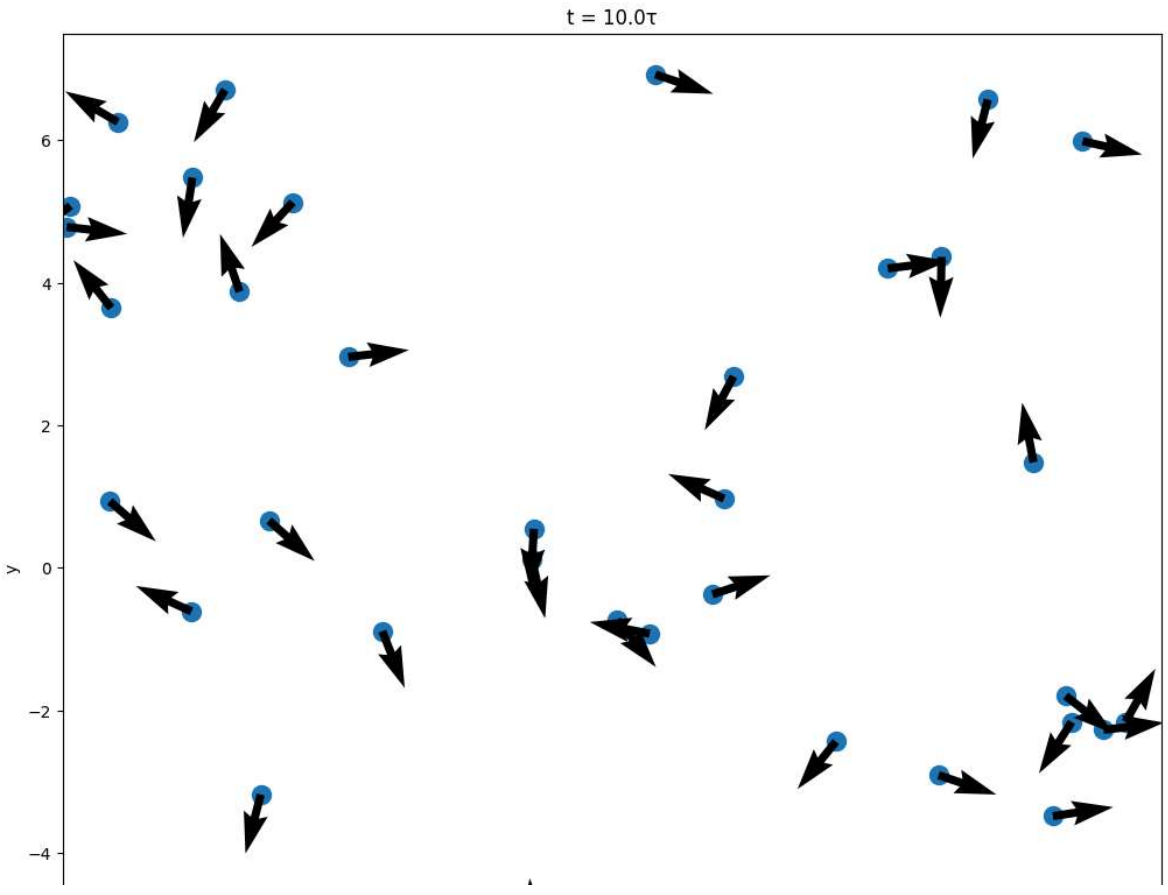
```
            canvas.create_oval(
                (x[j] - rp) / L * window_size + window_size / 2,
                (y[j] - rp) / L * window_size + window_size / 2,
                (x[j] + rp) / L * window_size + window_size / 2,
                (y[j] + rp) / L * window_size + window_size / 2,
                outline='#000000',
                fill='#A0A0A0',
            )
        )

    velocities = []
    for j in range(N_part):
        velocities.append(
            canvas.create_line(
                x[j] / L * window_size + window_size / 2,
                y[j] / L * window_size + window_size / 2,
                (x[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                (y[j] + vp * np.cos(phi[j])) / L * window_size + window_size / 2
                width=line_width,
            )
        )

step = 0

def stop_loop(event):
    global running
    running = False

if visuals_on:
    tk.bind("<Escape>", stop_loop)  # Bind the Escape key to stop the loop.
running = True  # Flag to control the loop.
while running:

    # Check whether plot configuration.
    if step*dt in t_to_plot:
        print(step*dt)
        if step*dt == 100:
            x2_100 = x
            y2_100 = y
        if step*dt == 500:
            x2_500 = x
            y2_500 = y
        #ax.clear()  # Clear previous plot.
        ax[i_fig].scatter(x, y, s=points_radius**2)
        ax[i_fig].quiver(x, y, np.cos(phi), np.sin(phi))
        '''ax[i_fig].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[i_fig].set_xlim([-L / 2, L / 2])
        ax[i_fig].set_ylim([-L / 2, L / 2])
        ax[i_fig].set_title(f't = {step*dt}τ')
        ax[i_fig].set_xlabel('x')
        ax[i_fig].set_ylabel('y')
        #display(fig)  # Display updated plot.
        #clear_output(wait=True)  # Clear previous output.
        i_fig += 1

    # Calculate current I.
    I_particles = calculate_intensity(x, y, I0, r0, L, r_c)
```

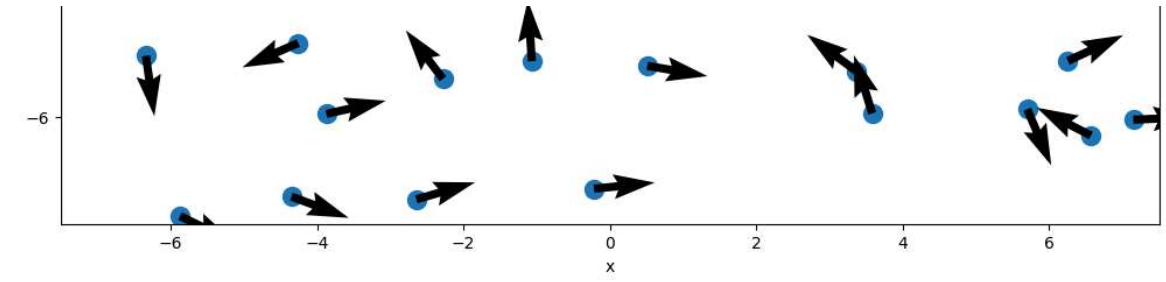
```
    if delta < 0:
        # Estimate the derivative of I linear using the last n_fit values.
        for i in range(N_part - 1):
            # Update I_fit.
            I_fit = np.roll(I_fit, -1, axis=0)
            I_fit[-1, :] = I_particles
            # Fit to determine the slope.
            for j in range(N_part):
                p = np.polyfit(t_fit, I_fit[:, j], 1)
                dI_dt[j] = p[0]
            # Determine forecast. Remember that here delta is negative.
            I = I_particles - delta * dI_dt
            I[np.where(I < 0)[0]] = 0
    elif delta > 0:
        # Update I_memory.
        I_memory = np.roll(I_memory, -1, axis=0)
        I_memory[-1, :] = I_particles
        I = I_memory[0, :]
    else:
        I = I_particles

    # Calculate new positions and orientations.
    v = v_inf + (v0 - v_inf) * np.exp(- I / Ic)
    nx = x + v * dt * np.cos(phi)
    ny = y + v * dt * np.sin(phi)
    nphi = phi + c_noise_phi * np.random.normal(0, 1, N_part)


    # Apply pbc.
    nx, ny = pbc(nx, ny, L)

    if visuals_on:
        # Update animation frame.
        if step % N_skip == 0:

            for j, light_spot in enumerate(light_spots):
                canvas.coords(
                    light_spot,
                    (nx[j] - r0) / L * window_size + window_size / 2,
                    (ny[j] - r0) / L * window_size + window_size / 2,
                    (nx[j] + r0) / L * window_size + window_size / 2,
                    (ny[j] + r0) / L * window_size + window_size / 2,
                )

            for j, particle in enumerate(particles):
                canvas.coords(
                    particle,
                    (nx[j] - rp) / L * window_size + window_size / 2,
                    (ny[j] - rp) / L * window_size + window_size / 2,
                    (nx[j] + rp) / L * window_size + window_size / 2,
                    (ny[j] + rp) / L * window_size + window_size / 2,
                )

            for j, velocity in enumerate(velocities):
                canvas.coords(
                    velocity,
                    nx[j] / L * window_size + window_size / 2,
                    ny[j] / L * window_size + window_size / 2,
                    (nx[j] + vp * np.cos(nphi[j])) / L * window_size + window_si
                    (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_si
```

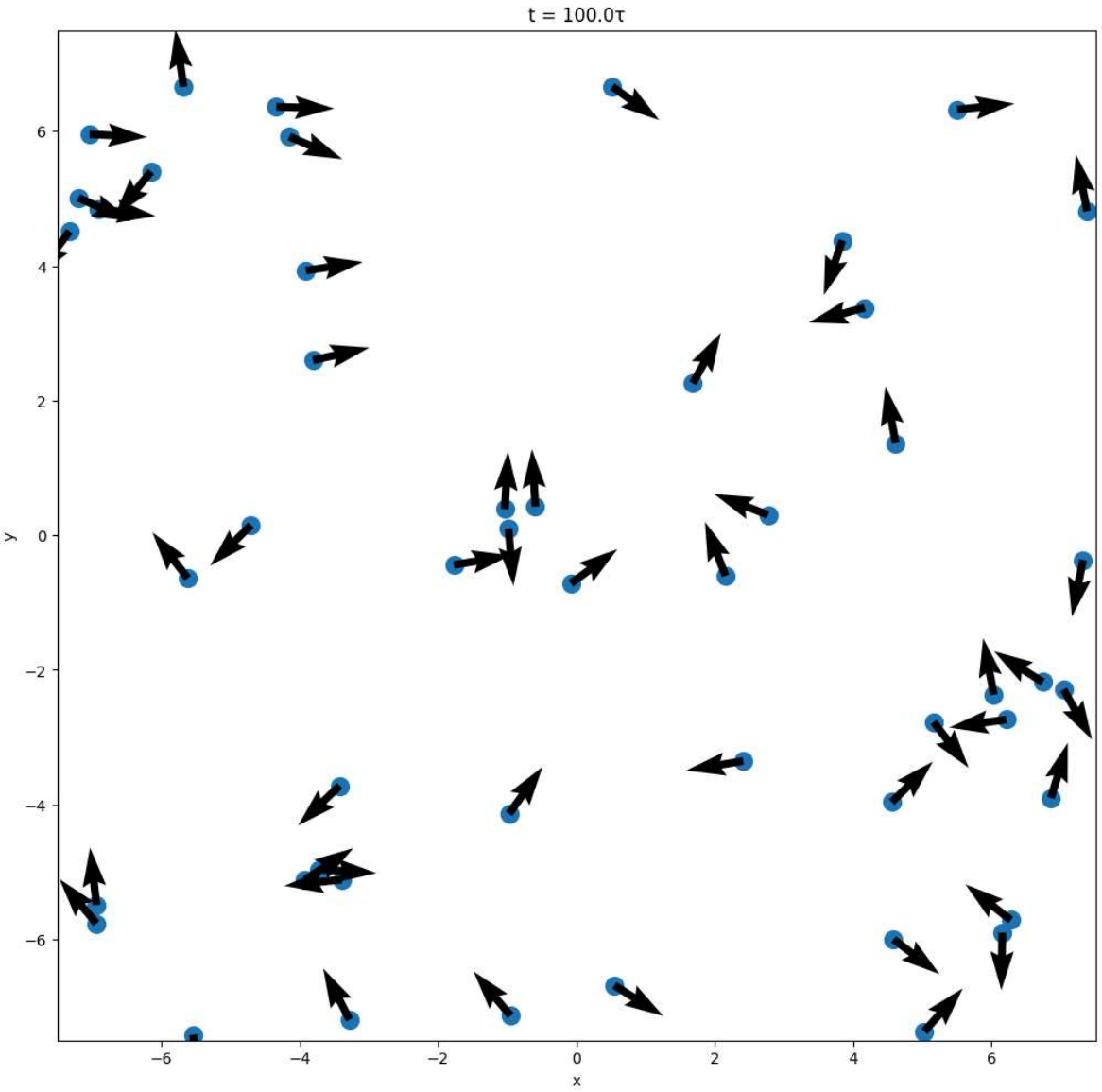
```
                )

            tk.title(f'Time {step * dt:.1f} - Iteration {step}')
            tk.update_idletasks()
            tk.update()
            time.sleep(.001)  # Increase to slow down the simulation.

    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]

    if step == N_max_steps+1:
        running = False

if visuals_on:
    tk.update_idletasks()
    tk.update()
    tk.mainloop()  # Release animation handle (close window to finish).
```

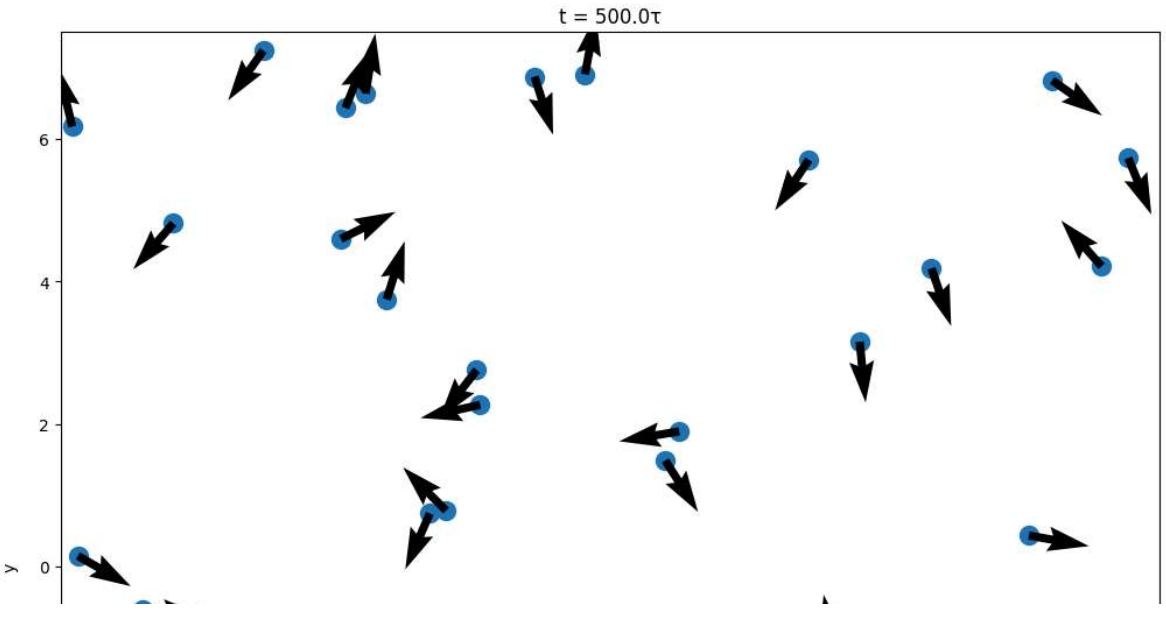
```
0.0
10.0
100.0
500.0
```

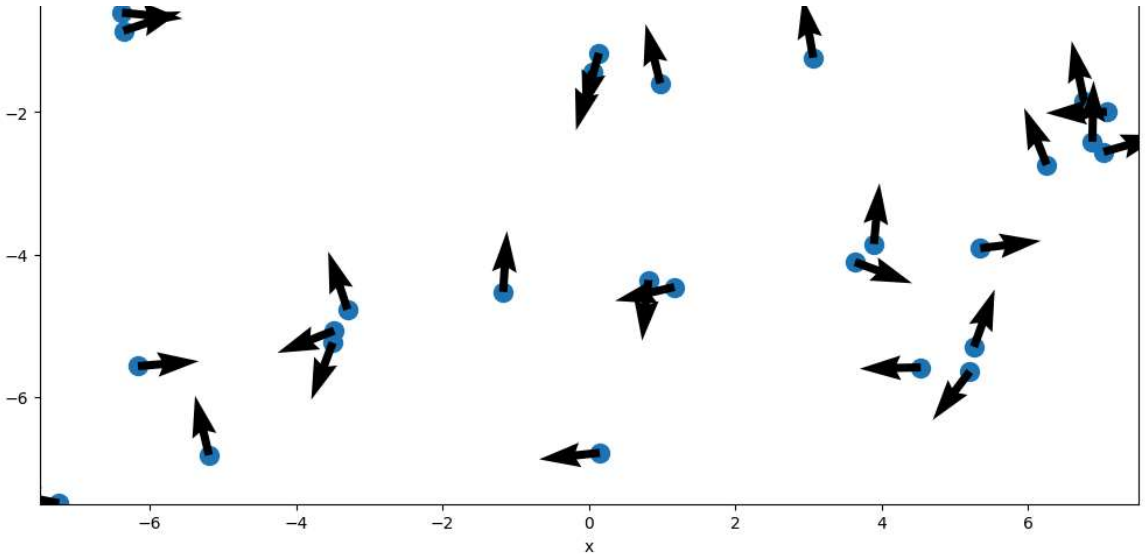

t = 0.0τ
x
y


t = 10.0τ
y


−6
−6
−4
−2
0
2
4
6
x


t = 100.0τ
6
4
2
0
−2
−4
−6
y
−6
−4
−2
0
2
4
6
x


t = 500.0τ
6
4
2
0
y



In [242…

```
from scipy.spatial import distance
import sys
import numpy
numpy.set_printoptions(threshold=sys.maxsize)

# DOESEN'T WORK

# def clusters(x, y):
#     cluster = []
#     coords = np.vstack([x,y]).T
#     to_delete = []
#     print(x.shape)
#     print(coords.shape)
#     distances = distance.cdist(coords, coords, 'euclidean')
#     less_than_r0 = distances<r0
#     n, m = less_than_r0.shape
#     for i in range(n):
#         inds = np.where(less_than_r0[i,:])[0]
#         #print(i)
#         if len(inds) != 1:
#             print(inds, len(inds))
#             cluster.append(inds.tolist())




#             '''for index in inds:
#                 print(f'index {index}')
#                 print(cluster)
#                 print(np.where(cluster==index)[0])
#                 already_ins = np.where(cluster==index)[0]
#                 print(already_ins)
#                 if len(already_ins) > 1:
#                     for already_in in already_ins:
#                         cluster[already_in].append(inds)
#                 else:
#                     cluster.append(inds)'''
#     for i in range(len(cluster)):
#         for j in range(i+1, len(cluster)):
#             if cluster[i] == cluster[j]:
#                 to_delete.append(j)
```

```
#     to_delete.sort(reverse=True)
#     for i in to_delete:
#         del cluster[i]
#     print(cluster)



#     return cluster









# cluster = clusters(x1_500, y1_500)
# plt.scatter(x1_500, y1_500)

# for inds in cluster:
#     for index in inds:
#         plt.scatter(x1_500[index], y1_500[index], c='r')
```

In [263…

```
def find_clusters(x, y, r0):
    """
    Identifies clusters of particles based on a distance threshold r0.

    Parameters:
        x (np.ndarray): An N x 1 array representing N particles x position.
        y (np.ndarray): An N x 1 array representing N particles y position.
        r0 (float): Distance threshold for clustering.

    Returns:
        list: A list of clusters, where each cluster is a list of particle indic
    """
    particles = np.vstack([x, y]).T

    # Number of particles
    n_particles = len(particles)

    # Initialize union-find (disjoint-set) data structure
    parent = list(range(n_particles))

    def find(x):
        """Find with path compression."""
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        """Union two sets."""
        root_x = find(x)
        root_y = find(y)
        if root_x != root_y:
            parent[root_y] = root_x
```

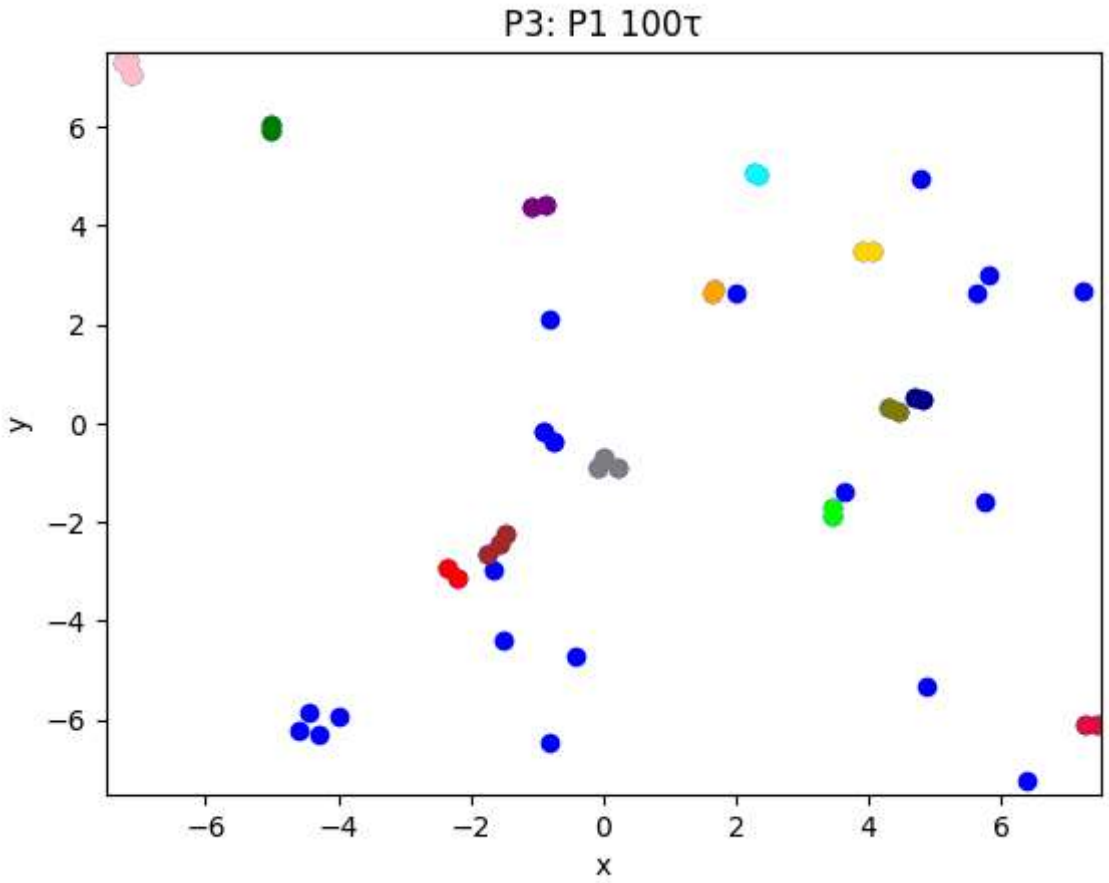
```
    # Check distances between all pairs of particles
    for i in range(n_particles):
        for j in range(i + 1, n_particles):
            if np.linalg.norm(particles[i] - particles[j]) < r0:
                union(i, j)

    # Group particles into clusters based on their root parent
    clusters = {}
    for i in range(n_particles):
        root = find(i)
        if root not in clusters:
            clusters[root] = []
        clusters[root].append(i)

    clusters = list(clusters.values())

    to_delete = []
    for i in range(len(clusters)):

        if len(clusters[i]) < 2:
            to_delete.append(i)

    to_delete.sort(reverse=True)
    for i in to_delete:
        del clusters[i]


    return clusters

#color_list = ['g', 'r', 'c', 'm', 'y', 'k', 'tab:blue', 'tab:orange', 'tab:purp
colors = [
    "blue", "orange", "green", "red", "purple", "brown", "pink", "gray", "olive"
    "darkblue", "gold", "lime", "crimson", "violet", "teal", "darkred", "silver"
]

r0 = 0.3
L = 50 * r0  # Side of the arena[m].
```

In [257…

```
cluster = find_clusters(x1_100, y1_100, r0)
print("Clusters:", len(cluster))

plt.scatter(x1_100, y1_100, c='b')
i = 0
for inds in cluster:

    color = colors[i]
    i += 1
    for index in inds:
        plt.scatter(x1_100[index], y1_100[index], c=color)

plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-L / 2, L / 2])
plt.ylim([-L / 2, L / 2])
plt.title('P3: P1 100τ')
```

```
Clusters: 14
```

Out[257… Text(0.5, 1.0, 'P3: P1 100τ')

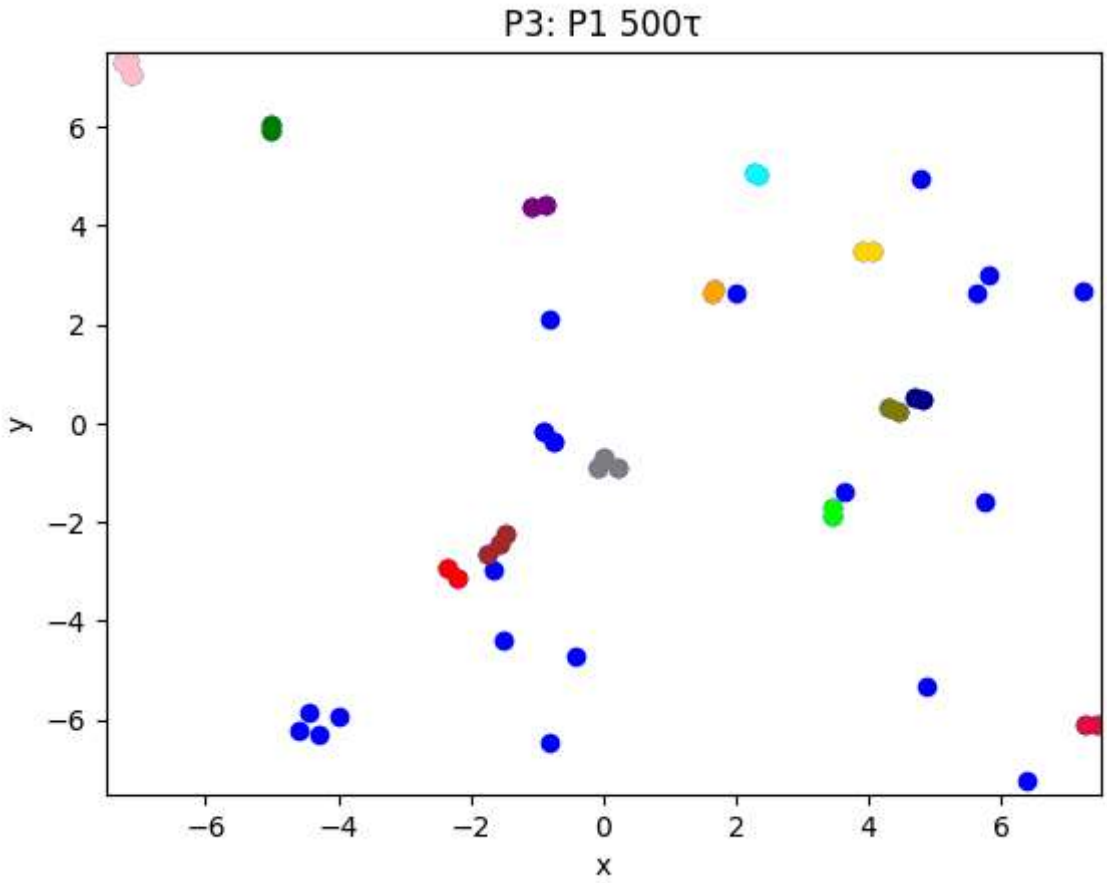


In [258…

```
cluster = find_clusters(x1_500, y1_500, r0)
print("Clusters:", len(cluster))

plt.scatter(x1_500, y1_500, c='b')
i = 0
for inds in cluster:

    color = colors[i]
    i += 1
    for index in inds:
        plt.scatter(x1_500[index], y1_500[index], c=color)

plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-L / 2, L / 2])
plt.ylim([-L / 2, L / 2])
plt.title('P3: P1 500τ')
```

```
Clusters: 14
```

Out[258… Text(0.5, 1.0, 'P3: P1 500τ')

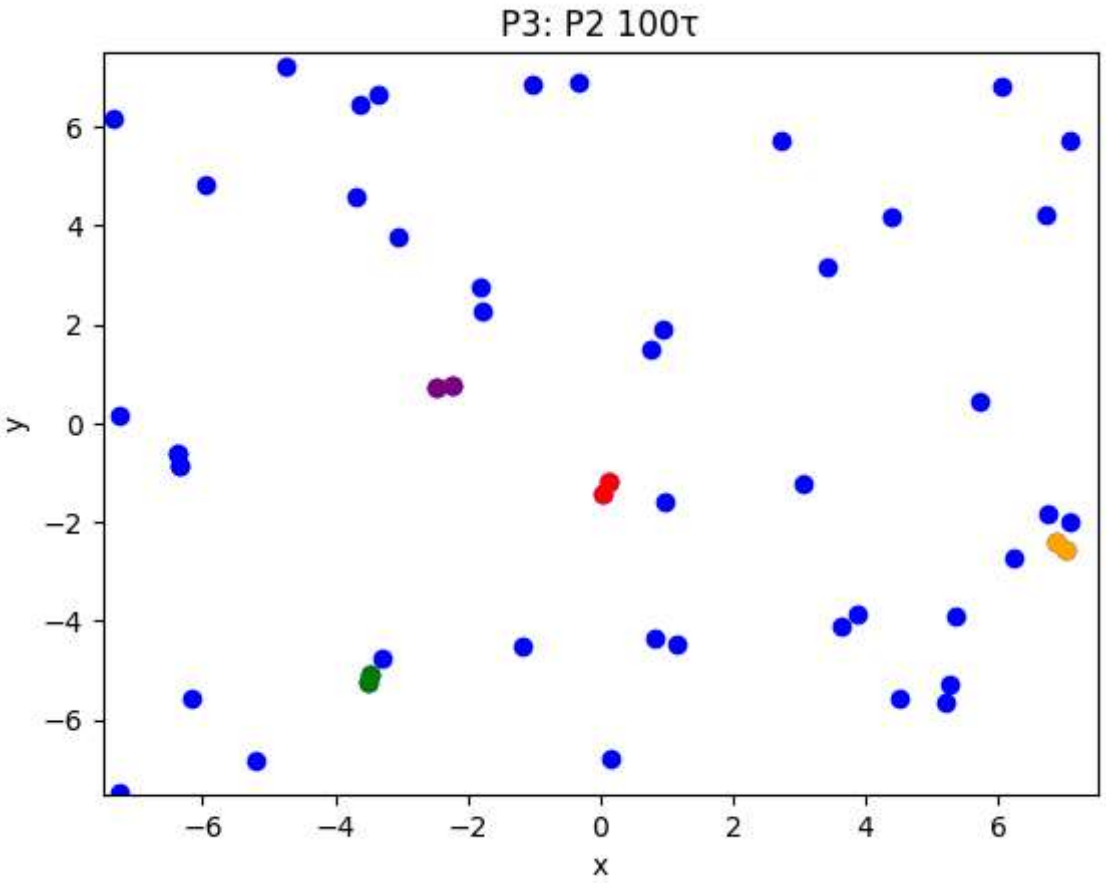


In [264…

```
cluster = find_clusters(x2_100, y2_100, r0)
print("Clusters:", len(cluster))

plt.scatter(x2_100, y2_100, c='b')
i = 0
for inds in cluster:

    color = colors[i]
    i += 1
    for index in inds:
        plt.scatter(x2_100[index], y2_100[index], c=color)

plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-L / 2, L / 2])
plt.ylim([-L / 2, L / 2])
plt.title('P3: P2 100τ')
```

```
Clusters: 5
```

Out[264… Text(0.5, 1.0, 'P3: P2 100τ')

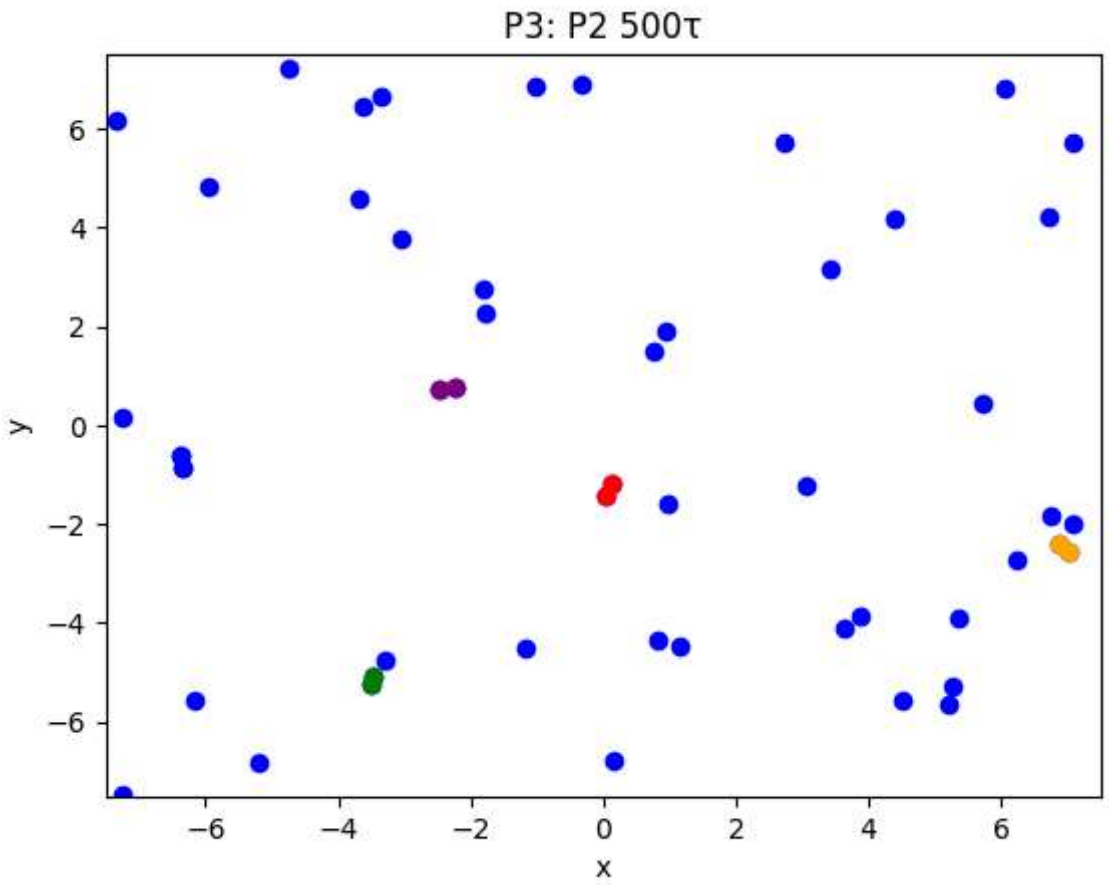


In [265…

```
cluster = find_clusters(x2_500, y2_500, r0)
print("Clusters:", len(cluster))

plt.scatter(x2_500, y2_500, c='b')
i = 0
for inds in cluster:

    color = colors[i]
    i += 1
    for index in inds:
        plt.scatter(x2_500[index], y2_500[index], c=color)

plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-L / 2, L / 2])
plt.ylim([-L / 2, L / 2])
plt.title('P3: P2 500τ')
```

```
Clusters: 5
```

Out[265… Text(0.5, 1.0, 'P3: P2 500τ')



Q1: The robots cluster far less in P2 than in P1.

Exercise 3

In [266…

```
def diffuse_spread_recover(x, y, status, d, beta, gamma, L, alpha):
    """
    Function performing the diffusion step, the infection step, and the
    recovery step happening in one turn for a population of agents.

    Parameters
    ==========
    x, y : Agents' positions.
    status : Agents' status.
    d : Diffusion probability.
    beta : Infection probability.
    gamma : Recovery probability.
    L : Side of the square lattice.
    """

    N = np.size(x)

    # Diffusion step.
    diffuse = np.random.rand(N)
    move = np.random.randint(4, size=N)
    for i in range(N):
        if diffuse[i] < d:
            if move[i] == 0:
                x[i] = x[i] - 1
            elif move[i] == 1:
                y[i] = y[i] - 1
            elif move[i] == 2:
```

```
                x[i] = x[i] + 1
            else:
                # move[i] == 3
                y[i] = y[i] + 1

    # Enforce pbc.
    x = x % L
    y = y % L

    # Spreading disease step.
    infected = np.where(status == 1)[0]

    for i in infected:
        # Check whether other particles share the same position.
        same_x = np.where(x == x[i])
        same_y = np.where(y == y[i])
        same_cell = np.intersect1d(same_x, same_y)
        for j in same_cell:
            if status[j] == 0:
                if np.random.rand() < beta:
                    status[j] = 1


    # Temporary immunity
    for i in np.where(status == 2):
        if np.random.rand() < alpha:
            status[i] = 0

    # Recover step.
    for i in infected:
        # Check whether the infected recovers.
        if np.random.rand() < gamma:
            status[i] = 2

    return x, y, status
```

In [267…

```
# For P1
N_part = 1000  # Total agent population.
d = 0.95  # Diffusion probability.
beta = 0.05  # Infection spreading probability.
gamma = 0.001  # Recovery probability.
L = 200  # Side of the lattice.

I0 = 30  # Initial number of infected agents.

# Initialize agents position.
x = np.random.randint(L, size=N_part)
y = np.random.randint(L, size=N_part)

# Initialize agents status.
status = np.zeros(N_part)
status[0:I0] = 1

alpha = 0.05
max_steps = 100000
```

In [268…

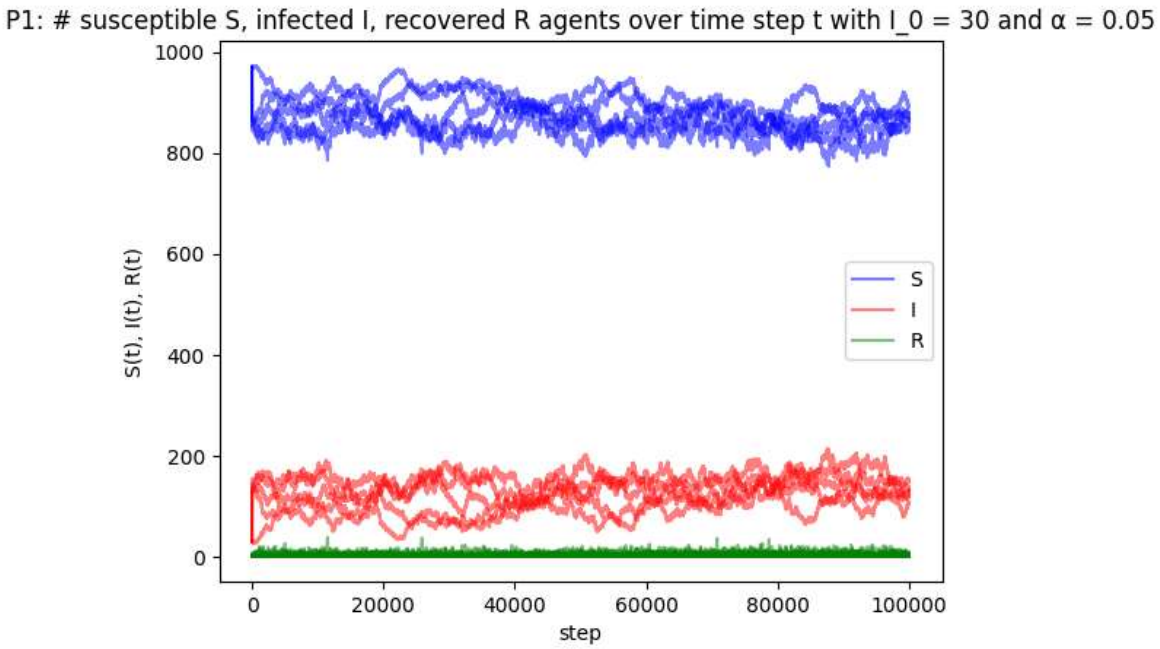
```
many_S1 = []
many_I1 = []
many_R1 = []
```

```
for i in range(5):
    step = 0

    S = []  # Keeps track of the susceptible agents.
    I = []  # Keeps track of the infectious agents.
    R = []  # Keeps track of the recovered agents.
    S.append(N_part - I0)
    I.append(I0)
    R.append(0)

    running = True  # Flag to control the loop.
    while running:


        x, y, status = diffuse_spread_recover(x, y, status, d, beta, gamma, L, a

        S.append(np.size(np.where(status == 0)[0]))
        I.append(np.size(np.where(status == 1)[0]))
        R.append(np.size(np.where(status == 2)[0]))

        step += 1
        if I[-1] == 0 or step > max_steps:
            running = False

    print('Done.')

    many_S1.append(S)
    many_I1.append(I)
    many_R1.append(R)
```

```
Done.
Done.
Done.
Done.
Done.
```

In [269…

```
for i in range(len(many_S1)):

    t = np.array(np.arange(len(S)))
    S_agents = np.array(many_S1[i])
    I_agents = np.array(many_I1[i])
    R_agents = np.array(many_R1[i])


    if i == 0:
        plt.plot(t, S_agents, 'b-', label='S', alpha=0.5)
        plt.plot(t, I_agents, 'r-', label='I', alpha=0.5)
        plt.plot(t, R_agents, 'g-', label='R', alpha=0.5)
    else:
        plt.plot(t, S_agents, 'b-', alpha=0.5)
        plt.plot(t, I_agents, 'r-', alpha=0.5)
        plt.plot(t, R_agents, 'g-', alpha=0.5)
plt.legend()
plt.title('P1: # susceptible S, infected I, recovered R agents over time step t
plt.xlabel('step')
plt.ylabel('S(t), I(t), R(t)')
plt.show()
```



In [240…

```
# For Q1

visuals_on = False

if visuals_on:

    N_part = 1000  # Total agent population.
    d = 0.95  # Diffusion probability.
    beta = 0.05  # Infection spreading probability.
    gamma = 0.001  # Recovery probability.
    L = 200  # Side of the lattice.

    I0 = 30  # Initial number of infected agents.

    # Initialize agents position.
    x = np.random.randint(L, size=N_part)
    y = np.random.randint(L, size=N_part)

    # Initialize agents status.
    status = np.zeros(N_part)
    status[0:I0] = 1

    alpha = 0.05



    import time
    from tkinter import *

    N_skip = 100  # Visualize status every N_skip steps.
    ra = 0.5  # Radius of the circle representing the agents.

    window_size = 600

    tk = Tk()
    tk.geometry(f'{window_size + 20}x{window_size + 20}')
    tk.configure(background='#000000')

    canvas = Canvas(tk, background='#ECECEC')  # Generate animation window.
```

```
    tk.attributes('-topmost', 0)
    canvas.place(x=10, y=10, height=window_size, width=window_size)


    step = 0

    S = []  # Keeps track of the susceptible agents.
    I = []  # Keeps track of the infectious agents.
    R = []  # Keeps track of the recovered agents.
    S.append(N_part - I0)
    I.append(I0)
    R.append(0)


    def stop_loop(event):
        global running
        running = False
    tk.bind("<Escape>", stop_loop)  # Bind the Escape key to stop the loop.
    running = True  # Flag to control the loop.
    while running:

        x, y, status = diffuse_spread_recover(x, y, status, d, beta, gamma, L, a

        S.append(np.size(np.where(status == 0)[0]))
        I.append(np.size(np.where(status == 1)[0]))
        R.append(np.size(np.where(status == 2)[0]))

        # Update animation frame.
        if step % N_skip == 0:
            canvas.delete('all')
            agents = []
            for j in range(N_part):
                agent_color = '#1f77b4' if status[j] == 0 \
                    else '#d62728' if status[j] == 1 else '#2ca02c'
                agents.append(
                    canvas.create_oval(
                        (x[j] - ra) / L * window_size,
                        (y[j] - ra) / L * window_size,
                        (x[j] + ra) / L * window_size,
                        (y[j] + ra) / L * window_size,
                        outline='',
                        fill=agent_color,
                    )
                )

            tk.title(f'Iteration {step}')
            tk.update_idletasks()
            tk.update()
            time.sleep(0.1)  # Increase to slow down the simulation.

        step += 1
        if I[-1] == 0:
            running = False

    tk.update_idletasks()
    tk.update()
    tk.mainloop()  # Release animation handle (close window to finish).
```

Q1: The system seems to have reached a steady state. This disease would continue existing and affecting the population forever. If the visualisation above is run, one can see the infected agents moving around the whole square meaning the disease is not endemic.

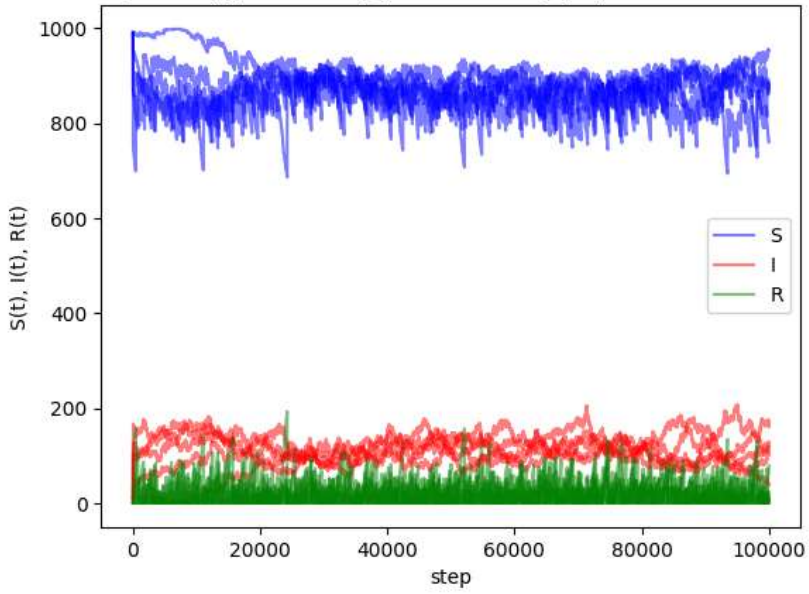
In [165…
```
# For P2
N_part = 1000  # Total agent population.
d = 0.95  # Diffusion probability.
beta = 0.05  # Infection spreading probability.
gamma = 0.001  # Recovery probability.
L = 200  # Side of the lattice.

I0 = 10  # Initial number of infected agents.

# Initialize agents position.
x = np.random.randint(L, size=N_part)
y = np.random.randint(L, size=N_part)

# Initialize agents status.
status = np.zeros(N_part)
status[0:I0] = 1

alpha = 0.005
max_steps = 100000
```

In [ ]:
```
many_S2 = []
many_I2 = []
many_R2 = []

for i in range(5):
    step = 0

    S = []  # Keeps track of the susceptible agents.
    I = []  # Keeps track of the infectious agents.
    R = []  # Keeps track of the recovered agents.
    S.append(N_part - I0)
    I.append(I0)
    R.append(0)

    running = True  # Flag to control the loop.
    while running:

        x, y, status = diffuse_spread_recover(x, y, status, d, beta, gamma, L, a

        S.append(np.size(np.where(status == 0)[0]))
        I.append(np.size(np.where(status == 1)[0]))
        R.append(np.size(np.where(status == 2)[0]))

        step += 1
        if I[-1] == 0 or step > max_steps:
            running = False

    print('Done.')

    many_S2.append(S)
    many_I2.append(I)
    many_R2.append(R)
```

```
Done.
Done.
Done.
Done.
Done.
```

In [ ]:
```
for i in range(len(many_S2)):

    t = np.array(np.arange(len(S)))
    S_agents = np.array(many_S2[i])
    I_agents = np.array(many_I2[i])
    R_agents = np.array(many_R2[i])


    if i == 0:
        plt.plot(t, S_agents, 'b-', label='S', alpha=0.5)
        plt.plot(t, I_agents, 'r-', label='I', alpha=0.5)
        plt.plot(t, R_agents, 'g-', label='R', alpha=0.5)
    else:
        plt.plot(t, S_agents, 'b-', alpha=0.5)
        plt.plot(t, I_agents, 'r-', alpha=0.5)
        plt.plot(t, R_agents, 'g-', alpha=0.5)
plt.legend()
plt.title('P2: # susceptible S, infected I, recovered R agents over time step t
plt.xlabel('step')
plt.ylabel('S(t), I(t), R(t)')
plt.show()
```

P2: Number of susceptible S(t), infected I(t), recovered R(t) agents as a function of the time step t



Q2: The plots don't look too different aside from that the disease sometimes dies when running the simulation. The number of susceptible and recovered agents also fluctuate more.

Exercise 4

In [172…
```
def erdos_renyi_rg(n, p):
    """
    Function generating an Erdős-Rényi random graph

    Parameters
```

```
    ==========
    n : Number of nodes.
    p : Probability that each possible edge is present.
    """

    A = np.zeros([n, n])
    rn = np.random.rand(n, n)
    A[np.where(rn < p)] = 1

    for i in range(n):
        A[i, i] = 0

    # This below is for plotting in a circular arrangement.
    x = np.cos(np.arange(n) / n * 2 * np.pi)
    y = np.sin(np.arange(n) / n * 2 * np.pi)

    return A, x, y

def nodes_degree(A):
    """
    Function returning the degree of a node.

    Parameters
    ==========
    A : Adjacency matrix (assumed symmetric).
    """

    degree = np.sum(A, axis=0)

    return degree

def path_length(A, i, j):
    """
    Function returning the minimum path length between thwo nodes.

    Parameters
    ==========
    A : Adjacency matrix (assumed symmetric).
    i, j : Nodes indices.
    """

    Lij = - 1

    if A[i, j] > 0:
        Lij = 1
    else:
        N = np.size(A[0, :])
        P = np.zeros([N, N]) + A
        n = 1
        running = True
        while running:
            P = np.matmul(P, A)
            n += 1
            running
            if P[i, j] > 0:
                Lij = n
            if (n > N) or (Lij > 0):
                running = False

    return Lij
```

```
def matrix_path_length(A):
    """
    Function returning a matrix L of minimum path length between nodes.

    Parameters
    ==========
    A : Adjacency matrix (assumed symmetric).
    """

    N = np.size(A[0, :])
    L = np.zeros([N, N]) - 1

    for i in range(N):
        for j in range(i + 1, N):
            L[i, j] = path_length(A, i, j)
            L[j, i] = L[i, j]

    return L


def clustering_coefficient(A):
    """
    Function returning the clustering coefficient of a graph.

    Parameters
    ==========
    A : Adjacency matrix (assumed symmetric).
    """

    K = nodes_degree(A)
    N = np.size(K)

    C_n = np.sum(np.diagonal(np.linalg.matrix_power(A, 3)))
    C_d = np.sum(K * (K - 1))

    C = C_n / C_d

    return C
```

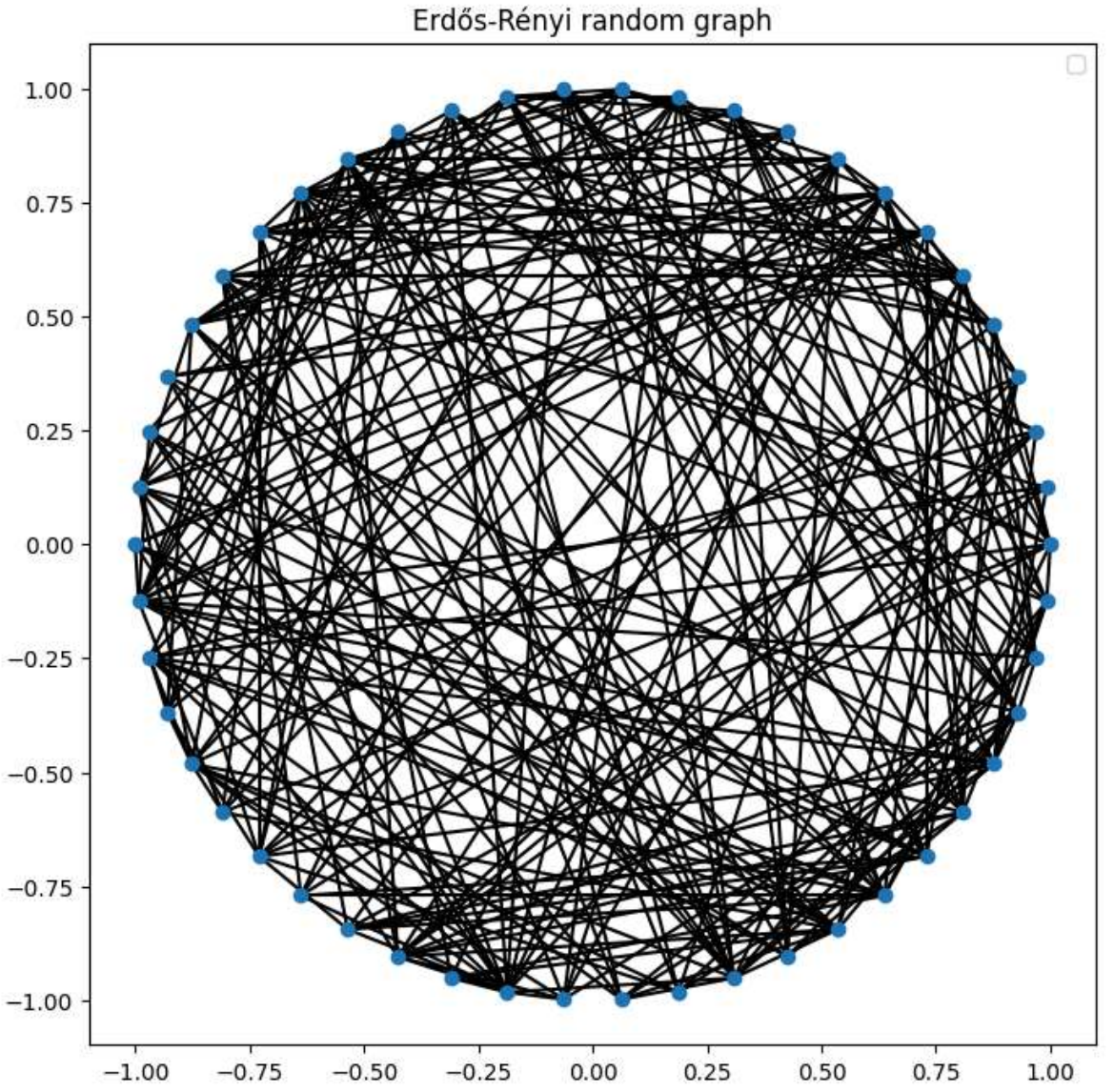
Task 1

In [216…

```
n = 50
p = 0.2

A_ER, x_ER, y_ER = erdos_renyi_rg(n, p)

plt.figure(figsize=(8, 8))

for i in range(n):
    for j in range(i + 1, n):
        if A_ER[i, j] > 0:
            plt.plot([x_ER[i], x_ER[j]], [y_ER[i], y_ER[j]], '-',
                     color='k')
plt.plot(x_ER, y_ER, '.', markersize=12)
plt.legend()
plt.title('Erdős-Rényi random graph')
```

```
plt.axis('equal')
plt.show()
```

```
C:\Users\richa\AppData\Local\Temp\ipykernel_12724\1698227184.py:14: UserWarning:
No artists with labels found to put in legend.  Note that artists whose label sta
rt with an underscore are ignored when legend() is called with no argument.
  plt.legend()
```



In [219…
```
n = 100
probs = [0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.4, 0.5, 0.6
```

In [220…
```
many_L_avg = np.zeros([len(probs), 10])
many_C = np.zeros([len(probs), 10])

k = 0
for p in probs:
    for i in range(10):
        A_ER, x_ER, y_ER = erdos_renyi_rg(n, p)
        L = matrix_path_length(A_ER)
        many_L_avg[k, i] = np.sum(L - np.diag(np.diag(L)))/(n*(n-1)) # For avera
        many_C[k, i] = clustering_coefficient(A_ER)
    k += 1
```

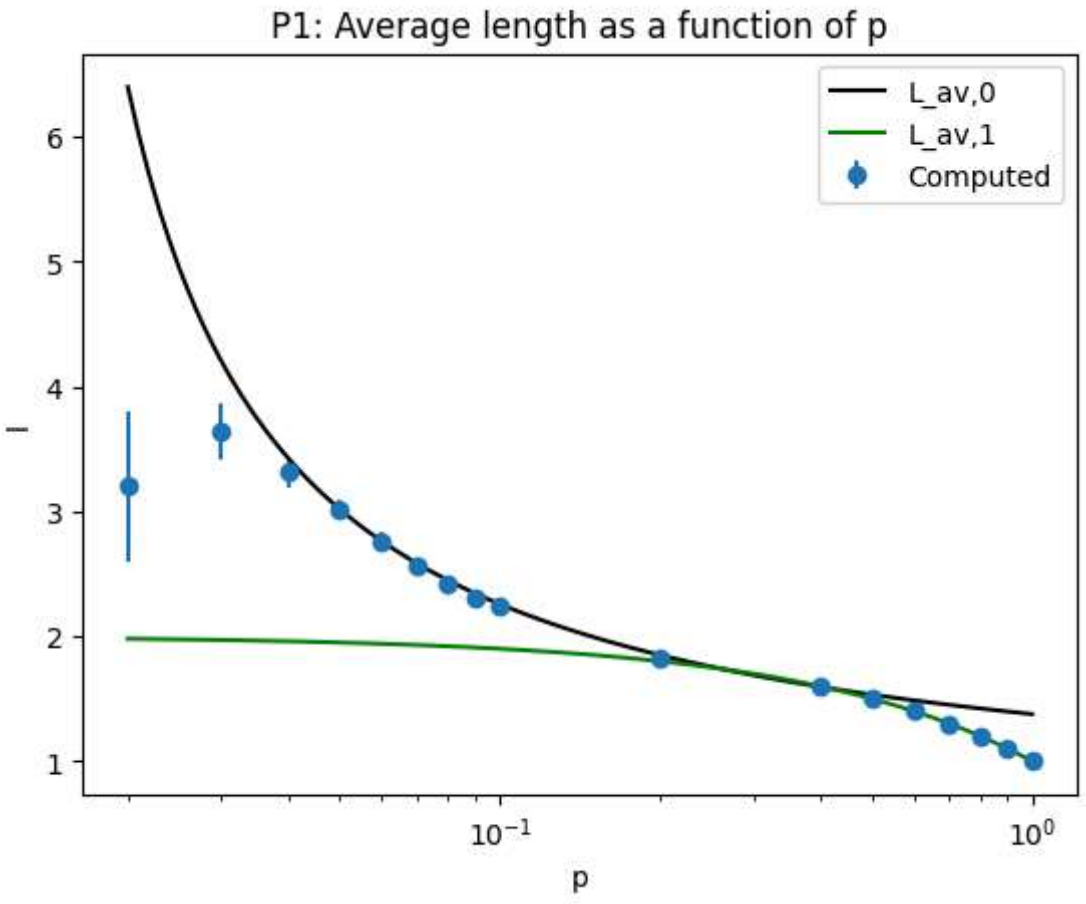
In [221…
```
# For error bars
many_L_avg_mean = np.mean(many_L_avg, axis=1)
```

```
many_L_avg_std = np.std(many_L_avg, axis=1)
```

In [222…

```
p_space = np.linspace(probs[0], probs[-1], 1000)

gamma = 0.57722
L_av0 = (np.log(n)-gamma) / (np.log(p_space*(n-1))) + 1/2
L_av1 = 2 - p_space

plt.errorbar(probs, many_L_avg_mean, many_L_avg_std, fmt='o', label = 'Computed'
plt.plot(p_space, L_av0, 'k-', label = 'L_av,0')
plt.plot(p_space, L_av1, 'g-', label = 'L_av,1')
plt.xscale('log')
plt.title('P1: Average length as a function of p')
plt.ylabel('l')
plt.xlabel('p')
plt.xscale('log')
plt.legend()
```

Out[222… <matplotlib.legend.Legend at 0x1ec9473be90>

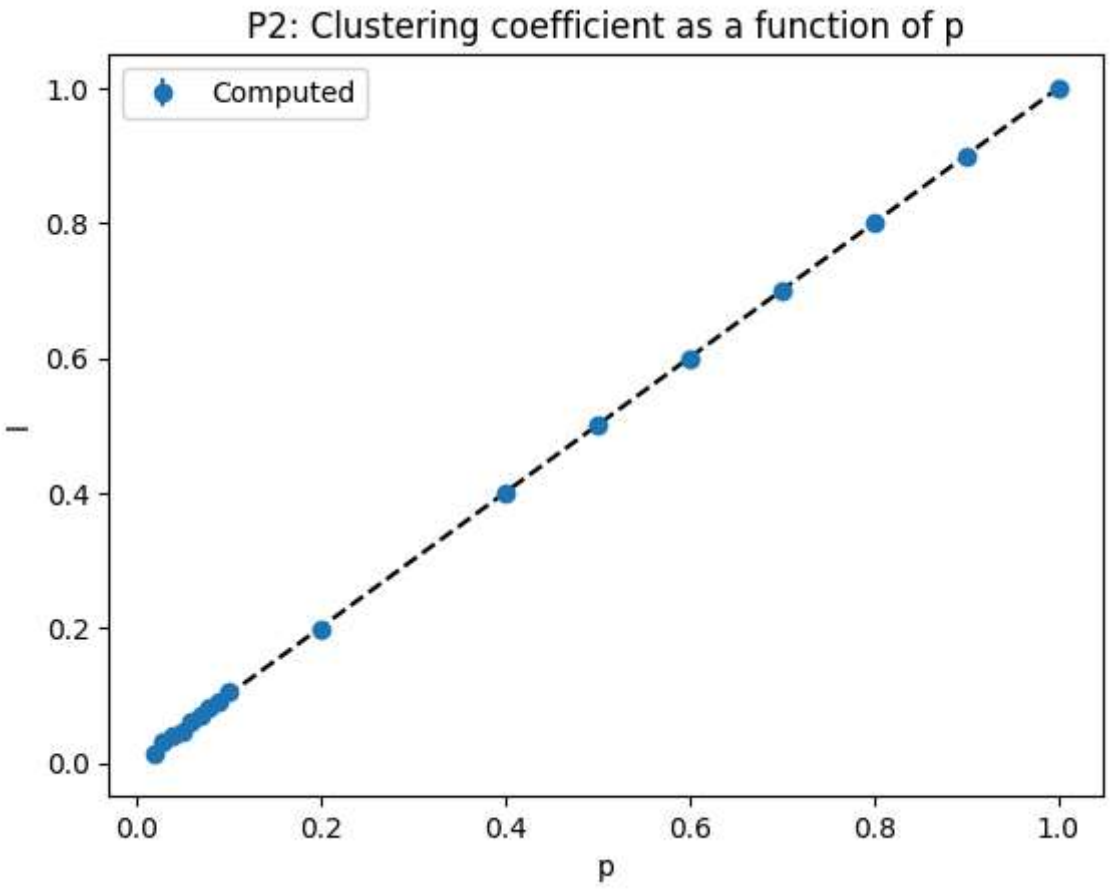


In [223…

```
# For error bars
many_C_mean = np.mean(many_C, axis=1)
many_C_std = np.std(many_C, axis=1)
```

In [224…
```
plt.errorbar(probs, many_C_mean, many_C_std, fmt='o', label = 'Computed')
plt.plot(p_space, p_space, '--k')


plt.title('P2: Clustering coefficient as a function of p')
plt.ylabel('l')
plt.xlabel('p')
plt.legend()
```

Out[224…
```
<matplotlib.legend.Legend at 0x1ecc67803b0>
```



Task 2

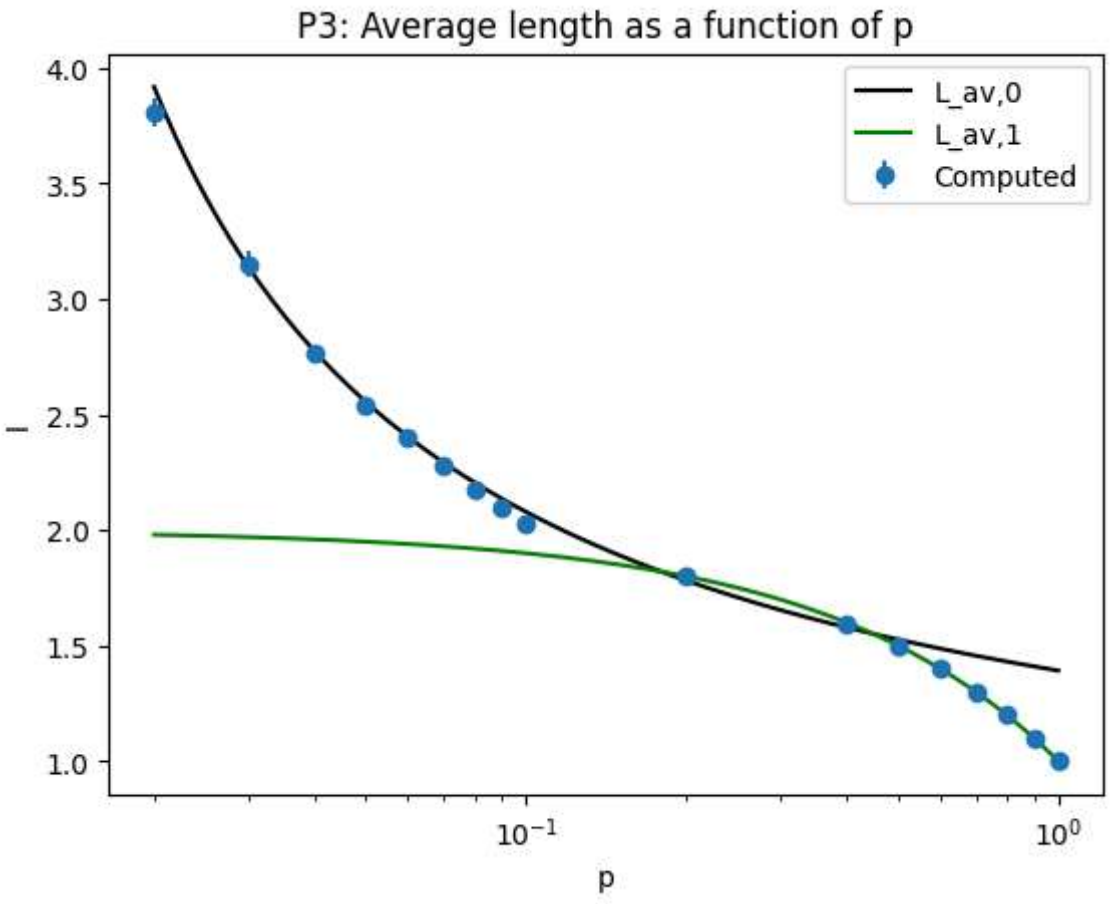
In [225…
```
n = 200
probs = [0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.4, 0.5, 0.6
```

In [226…
```
many_L_avg = np.zeros([len(probs), 10])
many_C = np.zeros([len(probs), 10])

k = 0
for p in probs:
    for i in range(10):
        A_ER, x_ER, y_ER = erdos_renyi_rg(n, p)
        L = matrix_path_length(A_ER)
        many_L_avg[k, i] = np.sum(L - np.diag(np.diag(L)))/(n*(n-1)) # For avera
        many_C[k, i] = clustering_coefficient(A_ER)
    k += 1
```

In [231…
```
# For error bars
many_L_avg_mean = np.mean(many_L_avg, axis=1)
many_L_avg_std = np.std(many_L_avg, axis=1)
```

In [232…
```
p_space = np.linspace(probs[0], probs[-1], 1000)

gamma = 0.57722
L_av0 = (np.log(n)-gamma) / (np.log(p_space*(n-1))) + 1/2
L_av1 = 2 - p_space

plt.errorbar(probs, many_L_avg_mean, many_L_avg_std, fmt='o', label = 'Computed'
plt.plot(p_space, L_av0, 'k-', label = 'L_av,0')
plt.plot(p_space, L_av1, 'g-', label = 'L_av,1')
plt.xscale('log')
plt.title('P3: Average length as a function of p')
plt.ylabel('l')
plt.xlabel('p')
plt.xscale('log')
plt.legend()
```

Out[232… <matplotlib.legend.Legend at 0x1eca6f267b0>

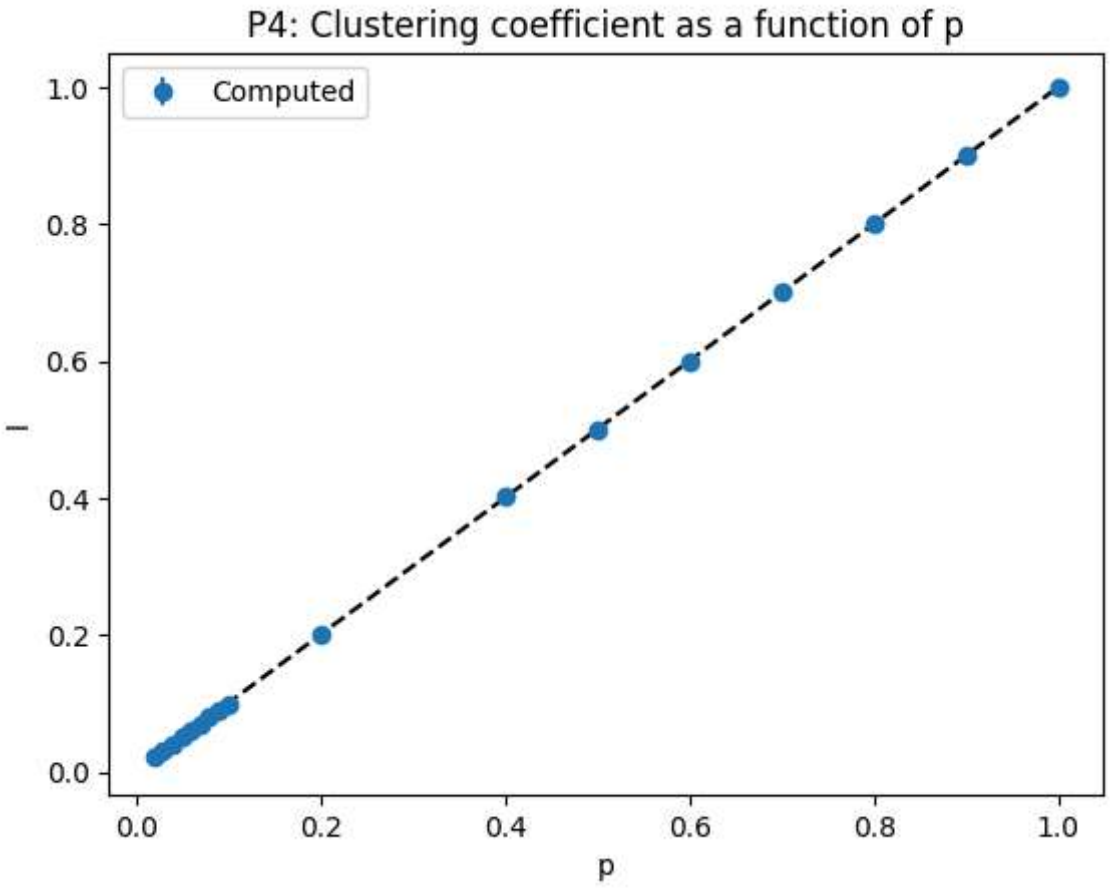


In [233…
```
# For error bars
many_C_mean = np.mean(many_C, axis=1)
many_C_std = np.std(many_C, axis=1)
```

In [234…
```
plt.errorbar(probs, many_C_mean, many_C_std, fmt='o', label = 'Computed')
plt.plot(p_space, p_space, '--k')


plt.title('P4: Clustering coefficient as a function of p')
```

```
plt.ylabel('l')
plt.xlabel('p')
plt.legend()
```

Out[234… <matplotlib.legend.Legend at 0x1ecc56dc380>



Q1: The clustering coefficient of a graph quantifies how the nodes are clustered together. Since in the Erdős–Rényi random graphs the way we choose how to connect the nodes is simply with the probability the clustering coefficient follows the probability.