Report of FFR120 HM3: Chapter 9-12

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

2024-11-23

1 Exercise 1 Particles with phoretic interaction.

1.1 Q1

The time step Δt must be chosen to balance simulation accuracy and computational efficiency while ensuring stability. The following factors influence this choice:

1. Particle Speed (v): To prevent particles from traveling too far (e.g., exceeding their radius R or interaction range $r_c = 10R$), the condition is:

$$\Delta t \ll \frac{r_c}{v} = \frac{10 \times 10^{-6}}{5 \times 10^{-6}} = 2\,\text{s}.$$

2. Phoretic Interaction Speed (v_0): To capture interactions accurately:

$$\Delta t \ll \frac{r_c}{v_0} = \frac{10 \times 10^{-6}}{20 \times 10^{-6}} = 0.5\,\text{s}.$$

3. Volume Exclusion and Particle Overlap: To prevent particle overlap, their displacement must remain smaller than the particle radius R. This gives:

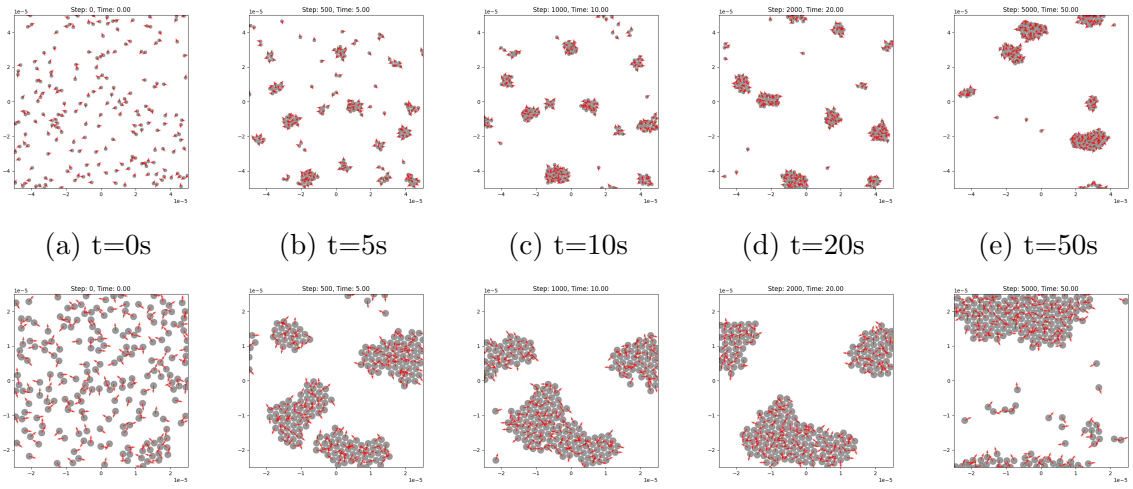
$$\Delta t \ll \frac{R}{v} = \frac{1 \times 10^{-6}}{5 \times 10^{-6}} = 0.2\,\text{s}.$$

So he most restrictive condition is $\Delta t \ll 0.2\,\text{s}$, one can choose a suitable time step is:

$$\Delta t = 0.01\,\text{s}.$$

1.2 P1 - P2

Particles of the same color represent those belonging to the same cluster. Overall, at both time points, the number of clusters under negative delay is smaller than that under positive delay.



(a) t=0s (b) t=5s (c) t=10s (d) t=20s (e) t=50s

(f) t=0s (g) t=5s (h) t=10s (i) t=20s (j) t=50s

Figure 1: (a)- (b) is Positive delay; (c) - (d) is Negative delay

Observations for P1 $L = 100R$: Weaker interactions result in slower clustering and a more random distribution, with less order emerging over time.

1. Initial stage ($t = 0$ s): Random particle distribution similar to P1.
2. Intermediate stage ($t = 5$ to $t = 20$ s):
   - Clustering is weaker and slower compared to P1 due to reduced interactions ($L = 50R$).
   - Clusters remain small and scattered.
3. Final stage ($t = 50$ s):
   - Particles remain more dispersed, with smaller and less dominant clusters compared to P1.
   - Particle orientations show less alignment.

Observations for P2 $L = 50R$: The system exhibits strong particle interactions, leading to significant clustering and increased order over time.

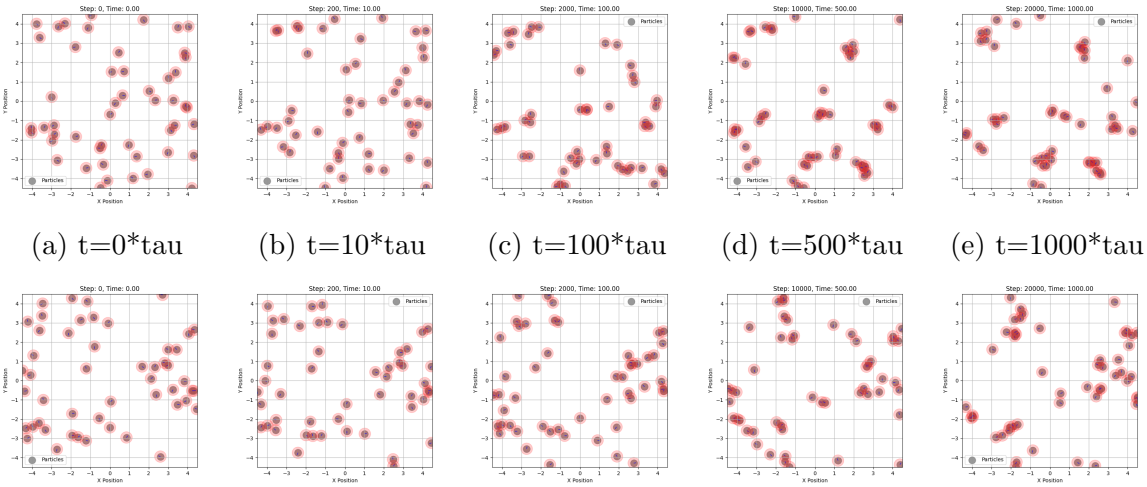
1. Initial stage ($t = 0$ s): Particles are randomly distributed with random orientations.
2. Intermediate stage ($t = 5$ to $t = 20$ s):
   - Small clusters begin to form, and particle orientations start aligning within clusters.
   - Clusters grow larger over time as interactions strengthen.

3. Final stage ($t = 50$ s): Large clusters dominate the system, and particles within each cluster show strong alignment.

Comparison of P1 and P2:

- P2 shows larger and fewer clusters, P1 has smaller and more dispersed clusters.
- P2 demonstrates stronger alignment of particle orientations, P1 retains more randomness.
- Spatial Distribution: P2 clusters are closer together, P1 particles remain more scattered.

2 Exercise 2 Sensory Delay

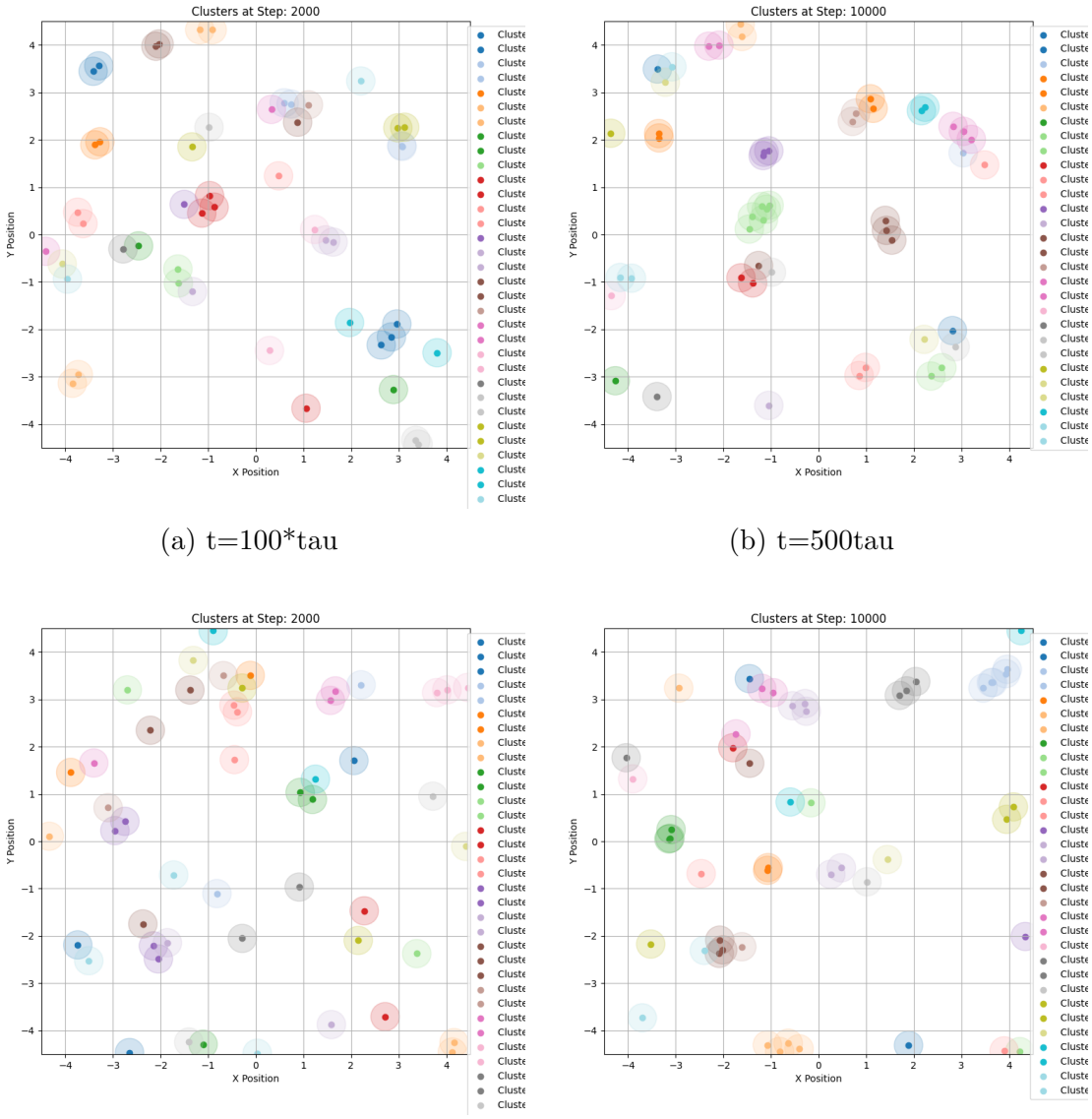


(a) t=0*tau (b) t=10*tau (c) t=100*tau (d) t=500*tau (e) t=1000*tau

(f) t=0*tau (g) t=10*tau (h) t=100*tau (i) t=500*tau (j) t=1000*tau

Figure 2: (a)- (e) is P1 Positive Delay; (f) - (j) is P2 Negative Delay

2.1 P3 Q1



(a) t=100*tau (b) t=500tau

(c) t=100tau (d) t=500tau

Figure 3: (a)- (e) is P1 L=100R; (f) - (j) is P2 L = 50R

3 Exercise 3 Disease Spreading(SRI model)

Introducing probability (alpha) to represent agents become susceptible again after recovered.

3.1 P1 and Q1

I0 = 30, 5 runs, steps = 50000, alpha = 0.05

the disease dose not die out, because the number of recover below 10.

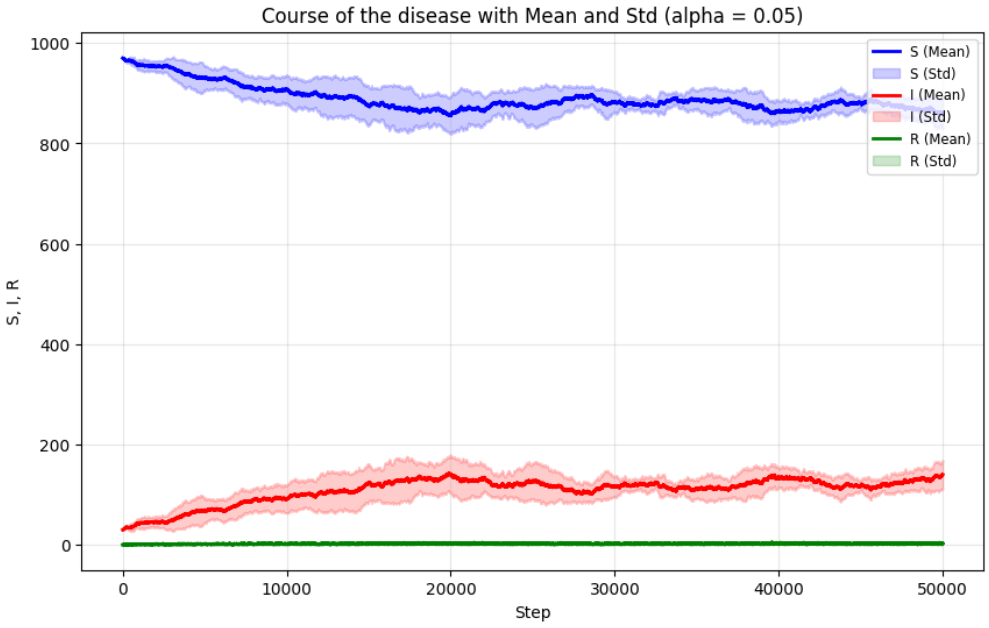
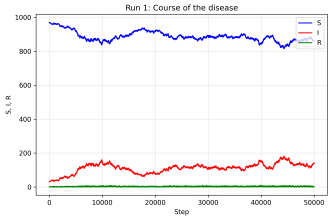
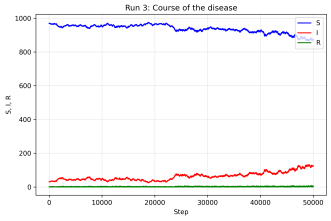


Figure 4: I0 =30, course of the disease with Mean and Std (alpha =0.05)

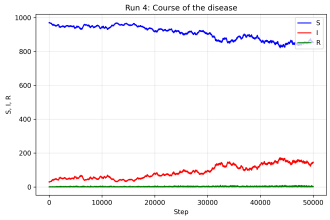


(a) t=0*tau

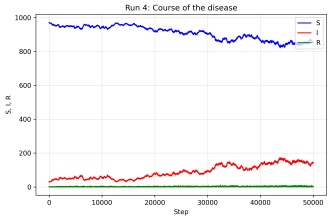
(b) t=10*tau



(c) t=100*tau



(d) t=500*tau



(e) t=1000*tau

Figure 5

3.2 P2 and Q2

I0 = 10, 5 runs, steps = 50000, alpha = 0.005

the disease sometimes die out, because the number of infected agent is zero in some runs.

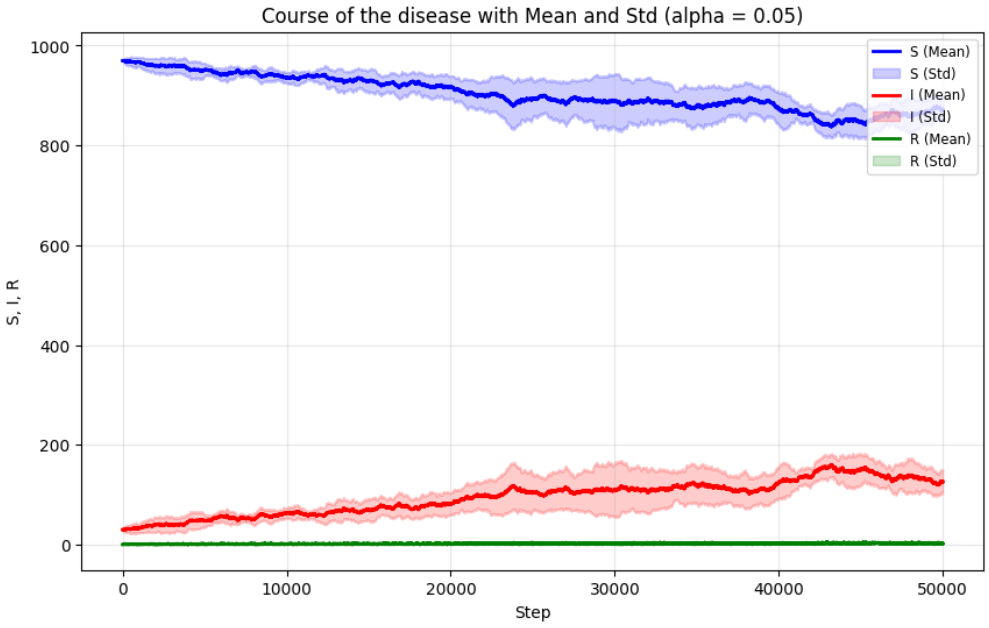
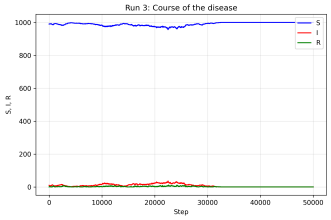


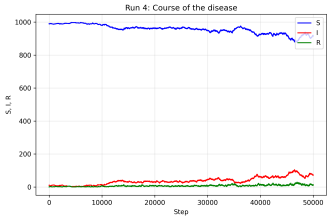
Figure 6: I0 =10, course of the disease with Mean and Std (alpha =0.05)

(a) t=0*tau

(b) t=10*tau



(c) t=100*tau



(d) t=500*tau

(e) t=1000*tau

Figure 7

4 Exercise 4 Network models

The clustering coefficient C increases linearly with p. For small p, C is close to 0 because very few triangles form. When p→1, the graph approaches a complete graph, so C approaches 1. The experimental results closely follow the theoretical linear trend C=p.

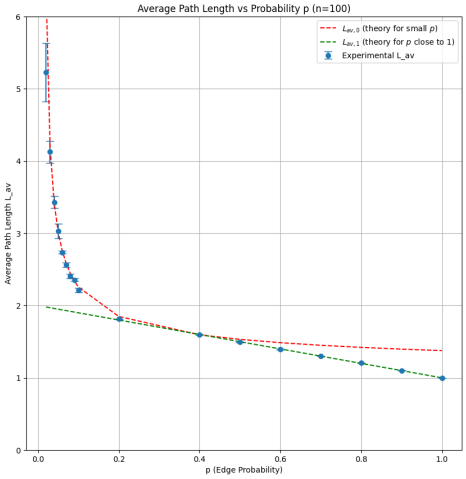
The average path length L_{av} is very high for small ($p \ll 1$) and decreases rapidly as p increases. For $p \to 1, L_{av} \to 1$. The experimental results match the theoretical predictions and the trends shown in Fig. 12.5a.

Q1: Why is $C = p$ in Erdős-Rényi graphs?

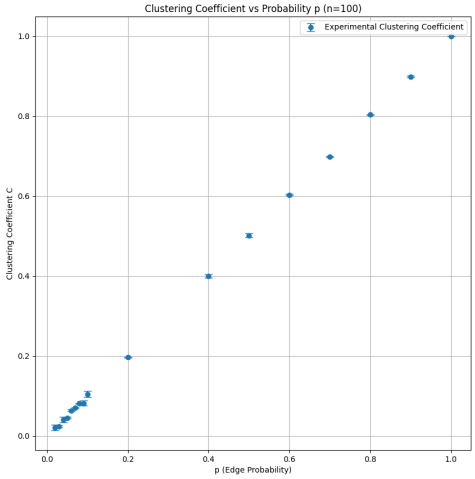
- 1. Independence of Edges: In Erdős-Rényi graphs, edges between nodes are generated independently with probability p.
- 2. Triangle Formation: For a node i, the probability that two neighbors j and k are connected is also p, leading to a direct proportionality between C and p.

n = 100 vs n = 200

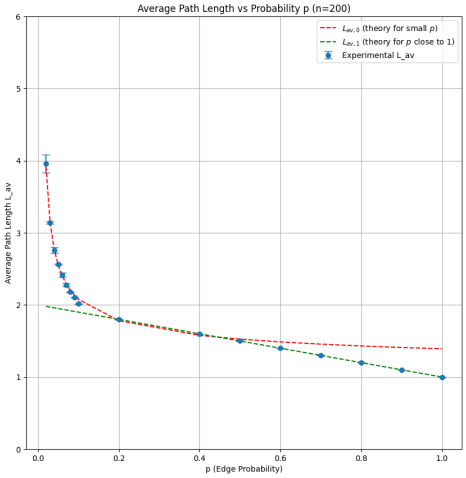
- Average Path Length: For $n = 200$, L_{av} is slightly shorter than for $n = 100$ because a larger graph allows more potential connections.
- Clustering Coefficient: C is independent of n, as it depends only on p. The trend is the same for both cases.



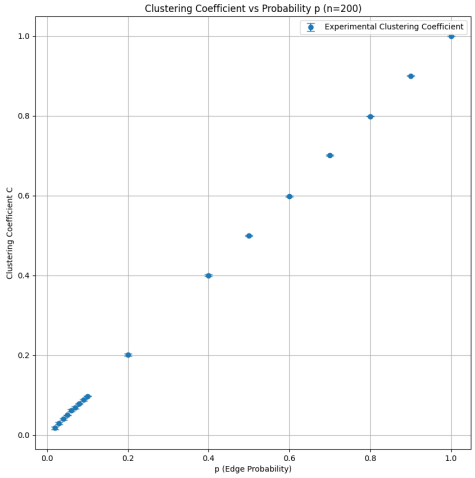
(a) Average path length n=100



(b) Clustering Coefficient n =100



(c) Average path length n=200



(d) Clustering Coefficient n =200

Figure 8: Erdos-R´enyi random graph

3.1_50R

November 26, 2024

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

[ ]:
```
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 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 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
```

[ ]:
```
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
```

[ ]:
```
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 = 1e-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.


# 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)
```

[ ]:
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *
import os
import matplotlib.pyplot as plt
```

```
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

# Plot
diameter_pixel = 2 * R / L * window_size
s_mpl = (diameter_pixel / 2) ** 2

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
target_time = [0, 5, 10, 20, 50]
N_steps = [int(t / dt) for t in target_time]

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:

    # 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,␣
↪N_part)
    ny = y + (v * np.sin(phi) + vp_y) * dt + c_noise_y * np.random.normal(0, 1,␣
↪N_part)
    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)

    # 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_size /
↪ 2,
                (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_size /
↪ 2,
            )
```

```
        tk.title(f'Time {step * dt:.1f} - Iteration {step}')
        #print(f'TK: step={step}, nx[1]={nx[1]}')
        tk.update_idletasks()
        tk.update()
        time.sleep(.001)  # Increase to slow down the simulation.


        if step in N_steps:
            #print(f'Mt: step={step}, nx[1]={nx[1]}')
            print(f'step = {step}, image saved.')
            plt.figure(figsize=(window_size / 100, window_size / 100))
            plt.scatter(nx, ny, c="gray", s=s_mpl, alpha=0.8, label="Particles")
            plt.quiver(
                nx, ny,
                vp * np.cos(nphi), vp * np.sin(nphi),
                angles="xy", scale_units="xy", scale=0.9,
                color="red", alpha=0.8
            )
            plt.title(f"Step: {step}, Time: {step * dt:.2f}")
            plt.xlim(-L / 2, L / 2)
            plt.ylim(-L / 2, L / 2)
            plt.gca().set_aspect("equal", adjustable="box")
            plt.savefig(f"50R_frame_{step}.png")
            plt.close()
    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]


    if step >= (max(N_steps)+1) :
        running = False

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

3.1_100R

November 26, 2024

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

[8]:
```
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
```

[9]:
```
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
```

[3]:
```
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
```

[4]:
```
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
```

0.1 L = 100 R

[ ]:
```
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 = 1e-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.


# 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)
```

```
The Kernel crashed while executing code in the current cell or a previous cell.


Please review the code in the cell(s) to identify a possible cause of the␣
↪failure.
```

Click here for more info.

View Jupyter log for further details.

[ ]:
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *
import os
import matplotlib.pyplot as plt



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

# Plot
diameter_pixel = 2 * R / L * window_size
s_mpl = (diameter_pixel / 2) ** 2

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
target_time = [0, 5, 10, 20, 50]
N_steps = [int(t / dt) for t in target_time]

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:

    # 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,␣
 ↪N_part)
    ny = y + (v * np.sin(phi) + vp_y) * dt + c_noise_y * np.random.normal(0, 1,␣
 ↪N_part)
    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)

    # 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_size /
↪ 2,
                (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_size /
↪ 2,
            )

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


        if step in N_steps:
            #print(f'Mt: step={step}, nx[1]={nx[1]}')
            print(f'step = {step}, image saved.')
            plt.figure(figsize=(window_size / 100, window_size / 100))
            plt.scatter(nx, ny, c="gray", s=s_mpl, alpha=0.8, label="Particles")
            plt.quiver(
                nx, ny,  #
                vp * np.cos(nphi), vp * np.sin(nphi),  #
                angles="xy", scale_units="xy", scale=0.9,  #
                color="red", alpha=0.8
            )
            plt.title(f"Step: {step}, Time: {step * dt:.2f}")
            plt.xlim(-L / 2, L / 2)
            plt.ylim(-L / 2, L / 2)
            plt.gca().set_aspect("equal", adjustable="box")
            plt.savefig(f"frame_{step}.png")
            plt.close()
    step += 1
    x[:] = nx[:]
    y[:] = ny[:]
    phi[:] = nphi[:]

    #
    if step >= (max(N_steps)+1) :
        running = False

tk.update_idletasks()
```

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

```
step = 0, image saved.
step = 500, image saved.
step = 1000, image saved.
step = 2000, image saved.
```

```
2024-11-24 21:05:53.913 Python[6531:400168] +[IMKClient subclass]: chose
IMKClient_Legacy
2024-11-24 21:05:53.913 Python[6531:400168] +[IMKInputSession subclass]: chose
IMKInputSession_Legacy
```

```
step = 5000, image saved.
```

3.2_negtive_delay

November 26, 2024

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

[2]:
```
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
```

[3]:
```
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
```

[4]:
```
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

def Plot_Simulation(x, y, phi, L, step, r_0, rp, vp, delay_type,␣
↪window_size,dt):
    """
    Function to plot particles with their light spots and velocity directions.

    Parameters:
    - x, y: Particle positions
    - phi: Orientation angles
    - L: Side length of the simulation box
    - step: Current simulation step
    - r_0: Radius of light spot
    - rp: Radius of particles
    - vp: Velocity arrow length
    - delay_type: 'positive' or 'negative' (used in saved filenames)
    """
    # screen DPI
    plt.figure()
    dpi = plt.gcf().dpi
    plt.close()

    fig_size_inch = window_size / dpi

    plt.figure(figsize=(fig_size_inch, fig_size_inch), dpi=dpi)
    ax = plt.gca()

    s_mpl = (2 * rp / L * window_size) ** 2
    plt.scatter(x, y, c="gray", s=s_mpl, alpha=0.8, label="Particles")

    for i in range(len(x)):
        light_spot = Circle((x[i], y[i]), r_0, color='red', alpha=0.2)
        ax.add_patch(light_spot)
        dx = vp * np.cos(phi[i])
        dy = vp * np.sin(phi[i])
        ax.arrow(x[i], y[i], dx, dy, head_width=0.1 * vp, head_length=0.1 * vp,␣
↪color='blue', alpha=0.6)
```

```
    plt.xlim(-L / 2, L / 2)
    plt.ylim(-L / 2, L / 2)
    plt.title(f"Step: {step}, Time: {step * dt:.2f}")
    plt.xlabel("X Position")
    plt.ylabel("Y Position")
    plt.grid(True)
    plt.legend()

    plt.savefig(f"{delay_type}_delay_frame_{step}.png", dpi=dpi)
    plt.close()
```

[5]:
```
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
```

```
[ ]: 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 = 30 * r0  # Side of the arena[m].



     target_time_index = [0, 10*tau, 100*tau, 500*tau, 1000*tau]

     N_steps = [int(t / dt) for t in target_time_index]

     # 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_delay):
        I_memory[i, :] += I_ref
```

The Kernel crashed while executing code in the current cell or a previous cell.

Please review the code in the cell(s) to identify a possible cause of the␣ ↪failure.

Click here for more info.

View Jupyter log for further details.

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


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
s_mpl = (rp / L * window_size) ** 2 # particles in matplotlib
vp = rp

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

    # 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)

# 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_size /
↪ 2,
                (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_size /
↪ 2,
            )

        if step in N_steps:
            delay_type = "positive" if delta > 0 else "negative"
            Plot_Simulation(nx, ny, nphi, L, step, r0, rp, vp, delay_type,␣
↪window_size, dt)
            print(f'dt ={dt}, step = {step}, image saved')

        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 >= (max(N_steps) +1):
        running = False

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

```
dt =0.05, step = 0, image saved
dt =0.05, step = 200, image saved
dt =0.05, step = 2000, image saved
dt =0.05, step = 10000, image saved
dt =0.05, step = 20000, image saved
```

3.2_positive_delay

November 26, 2024

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

[2]:
```
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
```

[3]:
```
def evolution_GI_posdelay(x0, y0, phi0, v_inf, v0, Ic, I0, r0, tau, dt,␣
↪duration, delta):
    """
    Function to generate the trajectory of a light-sensitive robot in a Gaussian
    light intensity zone with positive delay.

    Parameters
    ==========
    x0, y0 : Initial position [m].
    phi0 : Initial orientation [rad].
    v_inf : Self-propulsion speed at I=0 [m/s]
```

```
    v0 : Self-propulsion speed at I=I0 [m/s]
    Ic : Intensity scale over which the speed decays.
    I0 : Maximum intensity.
    r0 : Standard deviation of the Gaussian intensity.
    tau : Time scale of the rotational diffusion coefficient [s]
    dt : Time step for the numerical solution [s].
    duration : Total time for which the solution is computed [s].
    delta : Positive delay [s].
    """

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

    N = math.ceil(duration / dt)  # Number of time steps.

    x = np.zeros(N)
    y = np.zeros(N)
    phi = np.zeros(N)

    n_delay = int(delta / dt)  # Delay in units of time steps.

    rn = np.random.normal(0, 1, N - 1)

    x[0] = x0
    y[0] = y0
    phi[0] = phi0
    I_ref = I0 * np.exp(- (x0 ** 2 + y0 ** 2) / r0 ** 2)

    for i in range(N - 1):
        if i < n_delay:
            I = I_ref
        else:
            I = I0 * np.exp(- (x[i - n_delay] ** 2 + y[i - n_delay] ** 2) / r0␣
↪** 2)

        v = v_inf + (v0 - v_inf) * np.exp(- I / Ic)
        x[i + 1] = x[i] + v * dt * np.cos(phi[i])
        y[i + 1] = y[i] + v * dt * np.sin(phi[i])
        phi[i + 1] = phi[i] + c_noise_phi * rn[i]

    return x, y, phi
```

[4]:
```
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
```

[5]:
```
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
```

[6]:
```
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

def Plot_Simulation(x, y, phi, L, step, r_0, rp, vp, delay_type, window_size,␣
↪dt):
    """
    Function to plot particles with their light spots and velocity directions.

    Parameters:
    - x, y: Particle positions
    - phi: Orientation angles
    - L: Side length of the simulation box
    - step: Current simulation step
    - r_0: Radius of light spot
    - rp: Radius of particles
    - vp: Velocity arrow length
    - delay_type: 'positive' or 'negative' (used in saved filenames)
    """
    # screen DPI
    plt.figure()
    dpi = plt.gcf().dpi
    plt.close()

    fig_size_inch = window_size / dpi

    plt.figure(figsize=(fig_size_inch, fig_size_inch), dpi=dpi)
    ax = plt.gca()

    s_mpl = (2 * rp / L * window_size) ** 2
    plt.scatter(x, y, c="gray", s=s_mpl, alpha=0.8, label="Particles")

    for i in range(len(x)):
        light_spot = Circle((x[i], y[i]), r_0, color='red', alpha=0.2)
        ax.add_patch(light_spot)
        dx = vp * np.cos(phi[i])
        dy = vp * np.sin(phi[i])
        ax.arrow(x[i], y[i], dx, dy, head_width=0.1 * vp, head_length=0.1 * vp,␣
↪color='blue', alpha=0.6)


    plt.xlim(-L / 2, L / 2)
    plt.ylim(-L / 2, L / 2)
    plt.title(f"Step: {step}, Time: {step * dt:.2f}")
    plt.xlabel("X Position")
    plt.ylabel("Y Position")
    plt.grid(True)
    plt.legend()
```

```
    plt.savefig(f"{delay_type}_delay_frame_{step}.png", dpi=dpi)
    plt.close()
```

[ ]:
```
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 = 30 * r0  # Side of the arena[m].

target_time_index = [0, 10*tau, 100*tau, 500*tau, 1000*tau]
N_steps = [int(t/dt) for t in target_time_index]

# 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_delay):
        I_memory[i, :] += I_ref
```

```
The Kernel crashed while executing code in the current cell or a previous cell.


Please review the code in the cell(s) to identify a possible cause of the␣
 ↪failure.

Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info.


View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further details.
```

[ ]:
```
import time
from scipy.constants import Boltzmann as kB
from tkinter import *
import matplotlib.pyplot as plt

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
s_mpl = (rp / L * window_size) ** 2 # particles in matplotlib
vp = rp

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

    # 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)

    # 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_size /
↪ 2,
                (ny[j] + vp * np.sin(nphi[j])) / L * window_size + window_size /
↪ 2,
            )

        if step in N_steps:
            delay_type = "positive" if delta > 0 else "negative"
            Plot_Simulation(nx, ny, nphi, L, step, r0, rp, vp, delay_type,␣
↪window_size, dt)
            print(f'dt ={dt}, step = {step}, image saved')

        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 >= (max(N_steps) +1):
        running = False

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

```
dt =0.05, step = 0, image saved
dt =0.05, step = 200, image saved
dt =0.05, step = 2000, image saved
dt =0.05, step = 10000, image saved
dt =0.05, step = 20000, image saved
```

3.3_Disease_Spreading

November 26, 2024

[55]:
```
import numpy as np
from matplotlib import pyplot as plt
```

[56]:
```
def diffuse_spread_recover(x, y, status, d, beta, gamma, alpha, L):
    """
    Function performing the diffusion step, the infection step, the recovery␣
 ↪step,
    and the temporary immunity 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.
    alpha : Probability of recovered agents becoming susceptible again.
    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 periodic boundary conditions (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

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

    # Recovered agents becoming susceptible again
    recovered = np.where(status == 2)[0]
    for i in recovered:
        if np.random.rand() < alpha:
            status[i] = 0

    return x, y, status
```

0.1 P1 & Q1

I0 = 30, 5 runs, steps = 50000, alpha = 0.05

the disease dose not die out, because the number of recover below 10.

[57]:
```
# Simulation parameters.
N_part = 1000  # Total agent population.
d = 0.95  # Diffusion probability.
beta = 0.05  # Infection spreading probability.
gamma = 0.001  # Recovery probability.
alpha = 0.05  # Probability of losing immunity.
L = 200  # Side of the lattice.

I0 = 30  # Initial number of infected agents.

N_steps = 50000
n_runs = 5
```

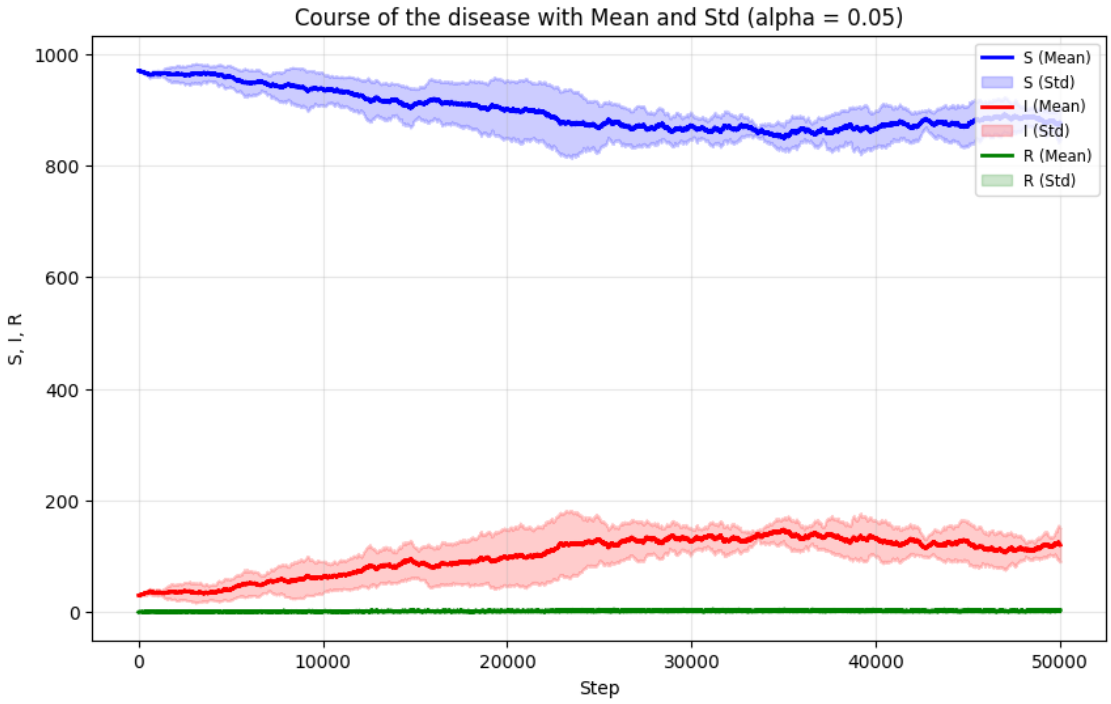
```
# Initialize agents position.
x = np.random.randint(L, size=N_part)
y = np.random.randint(L, size=N_part)

# Initialize agents status.
status = np.zeros(N_part)  # All agents are susceptible initially.
status[0:I0] = 1  # Set the first I0 agents as infected.

S_runs = []
I_runs = []
R_runs = []

for run in range(n_runs):
    print(f"Starting run {run + 1}...")

    # Initialize agents position and status.
    x = np.random.randint(L, size=N_part)
    y = np.random.randint(L, size=N_part)
    status = np.zeros(N_part)
    status[0:I0] = 1  # Set the first I0 agents as infected.


    step = 0
    S = [N_part - I0]
    I = [I0]
    R = [0]
    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,␣
 ↪alpha, L)

        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 step % 10000 ==0:
            print(f'step ={step}, susceptible = {S[-1]}, infectious ={I[-1]},␣
 ↪recovered = {R[-1]}, ')
```

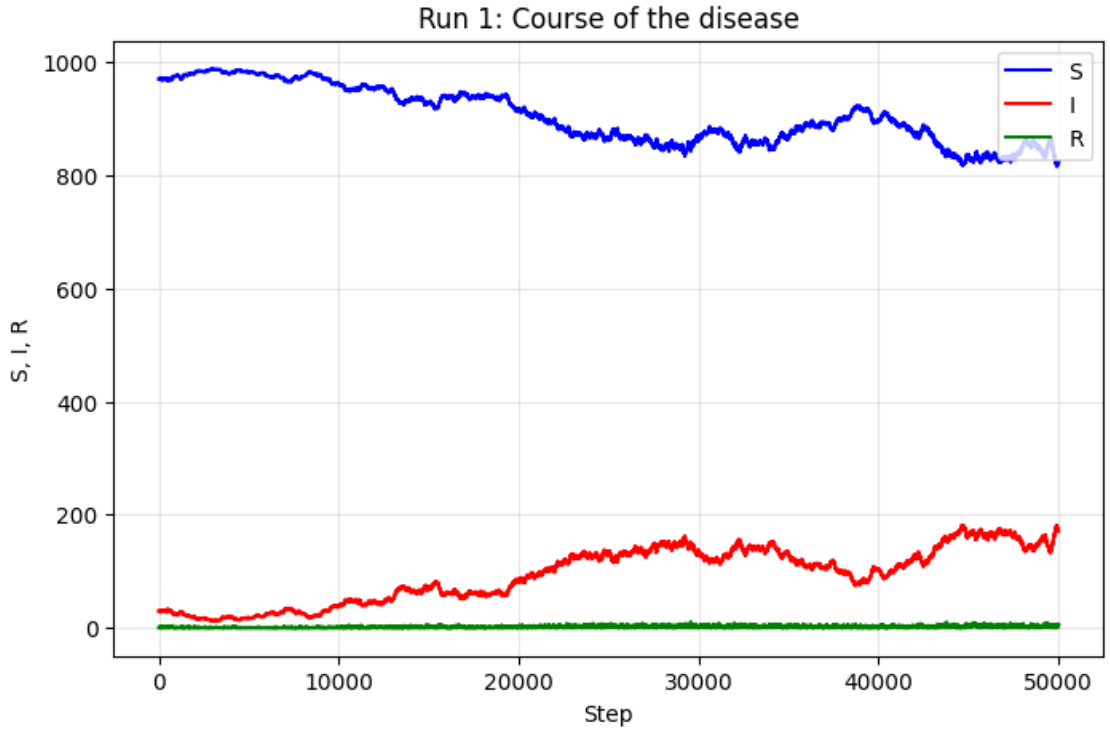
```
        if step > N_steps:
            running = False

    S_runs.append(S)
    I_runs.append(I)
    R_runs.append(R)


print('Done.')
```

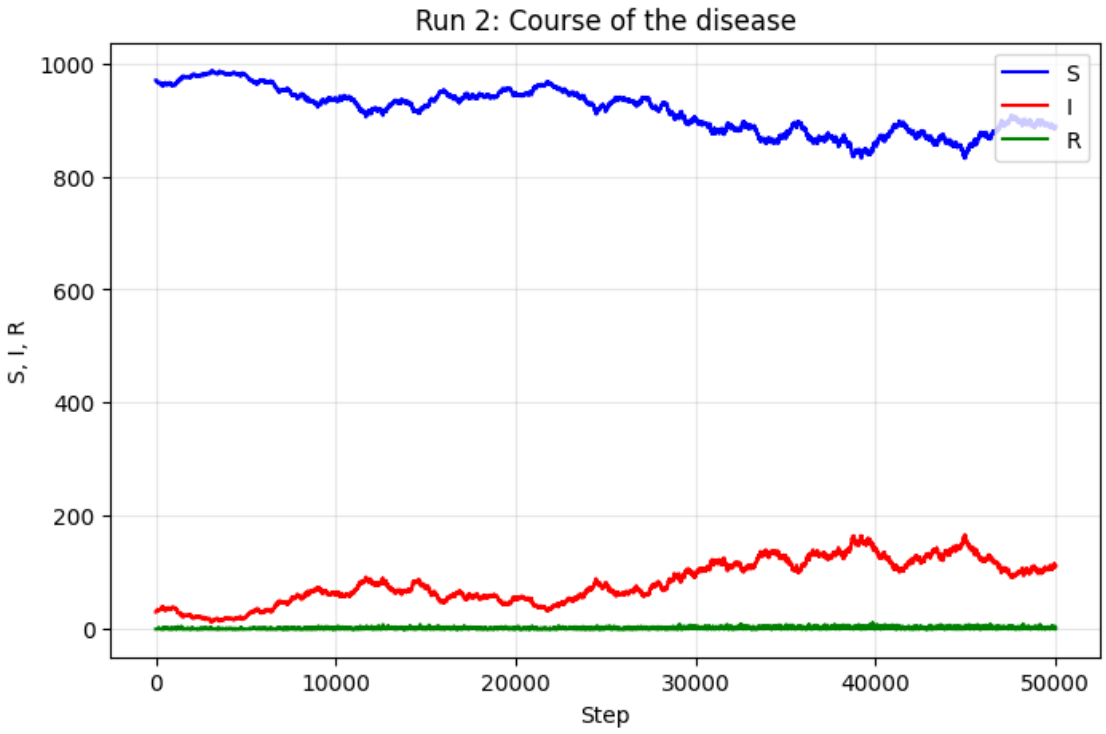
```
Starting run 1…
step =10000, susceptible = 960, infectious =39, recovered = 1,
step =20000, susceptible = 913, infectious =84, recovered = 3,
step =30000, susceptible = 864, infectious =136, recovered = 0,
step =40000, susceptible = 897, infectious =103, recovered = 0,
step =50000, susceptible = 824, infectious =171, recovered = 5,
Starting run 2…
step =10000, susceptible = 938, infectious =62, recovered = 0,
step =20000, susceptible = 945, infectious =55, recovered = 0,
step =30000, susceptible = 901, infectious =95, recovered = 4,
step =40000, susceptible = 859, infectious =140, recovered = 1,
step =50000, susceptible = 887, infectious =113, recovered = 0,
Starting run 3…
step =10000, susceptible = 898, infectious =100, recovered = 2,
step =20000, susceptible = 814, infectious =184, recovered = 2,
step =30000, susceptible = 835, infectious =159, recovered = 6,
step =40000, susceptible = 859, infectious =138, recovered = 3,
step =50000, susceptible = 916, infectious =80, recovered = 4,
Starting run 4…
step =10000, susceptible = 914, infectious =85, recovered = 1,
step =20000, susceptible = 873, infectious =124, recovered = 3,
step =30000, susceptible = 889, infectious =109, recovered = 2,
step =40000, susceptible = 906, infectious =91, recovered = 3,
step =50000, susceptible = 870, infectious =127, recovered = 3,
Starting run 5…
step =10000, susceptible = 975, infectious =25, recovered = 0,
step =20000, susceptible = 953, infectious =45, recovered = 2,
step =30000, susceptible = 861, infectious =135, recovered = 4,
step =40000, susceptible = 812, infectious =185, recovered = 3,
step =50000, susceptible = 885, infectious =112, recovered = 3,
Done.
```

[58]:
```
S_array = np.array(S_runs)
I_array = np.array(I_runs)
R_array = np.array(R_runs)

S_mean = np.mean(S_array, axis=0)
```

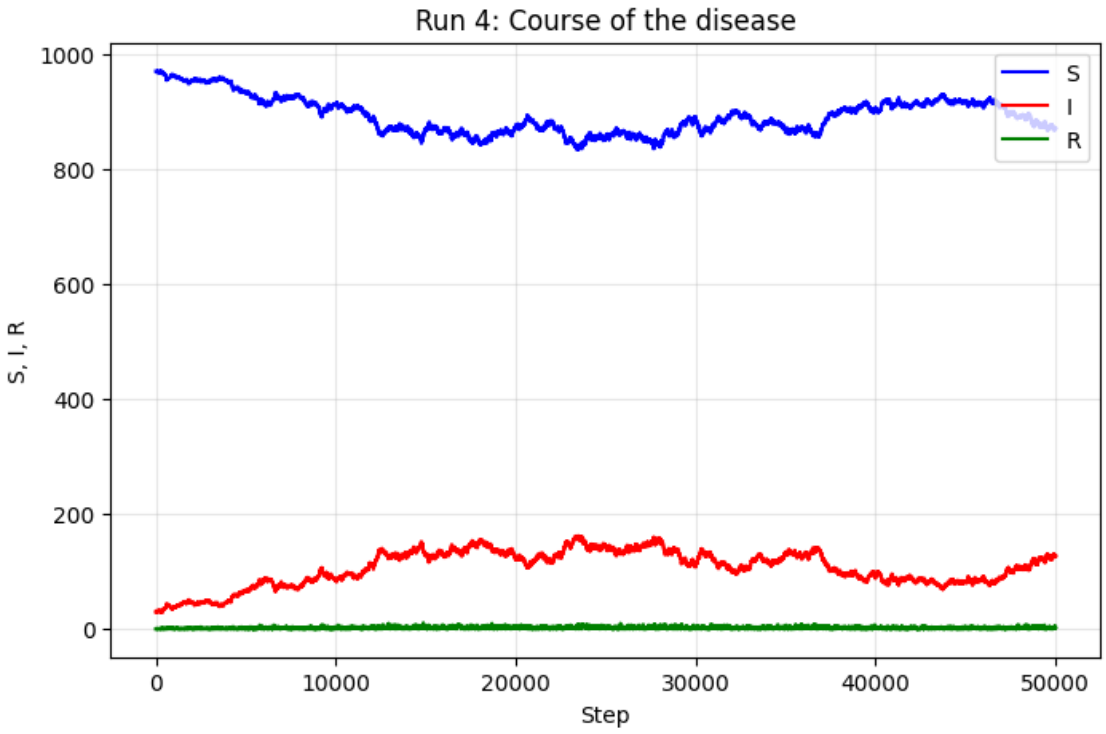
```
I_mean = np.mean(I_array, axis=0)
R_mean = np.mean(R_array, axis=0)

S_std = np.std(S_array, axis=0)
I_std = np.std(I_array, axis=0)
R_std = np.std(R_array, axis=0)

t = np.arange(S_mean.shape[0])

plt.figure(figsize=(10, 6))

# Plot mean with error bars
plt.plot(t, S_mean, 'b-', label='S (Mean)', linewidth=2)  #
plt.fill_between(t, S_mean - S_std, S_mean + S_std, color='blue', alpha=0.2,␣
 ↪label='S (Std)')

plt.plot(t, I_mean, 'r-', label='I (Mean)', linewidth=2)  #
plt.fill_between(t, I_mean - I_std, I_mean + I_std, color='red', alpha=0.2,␣
 ↪label='I (Std)')

plt.plot(t, R_mean, 'g-', label='R (Mean)', linewidth=2)  #
plt.fill_between(t, R_mean - R_std, R_mean + R_std, color='green', alpha=0.2,␣
 ↪label='R (Std)')
plt.legend(loc='upper right', fontsize='small')
plt.title(f'Course of the disease with Mean and Std (alpha = {alpha})')
plt.xlabel('Step')
plt.ylabel('S, I, R')
plt.grid(alpha=0.3)
plt.savefig('P1_mean_and_std.png', dpi=300)
plt.show()


for run in range(len(S_runs)):
    t = np.arange(len(S_runs[run]))

    plt.figure(figsize=(8, 5))
    plt.plot(t, S_runs[run], 'b-', label='S')
    plt.plot(t, I_runs[run], 'r-', label='I')
    plt.plot(t, R_runs[run], 'g-', label='R')
    plt.legend(loc='upper right')
    plt.title(f'Run {run + 1}: Course of the disease')
    plt.xlabel('Step')
    plt.ylabel('S, I, R')
    plt.grid(alpha=0.3)
    plt.savefig(f'Run_{run + 1}.png', dpi=300)  #
    plt.show()
```


Course of the disease with Mean and Std (alpha = 0.05)
S (Mean)
S (Std)
I (Mean)
I (Std)
R (Mean)
R (Std)
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 1: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 2: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 3: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 4: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 5: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000

0.2 P2 & Q2

I0 = 10, 5 runs, steps = 50000, alpha = 0.005

[52]:
```
# Simulation parameters.
N_part = 1000  # Total agent population.
d = 0.95  # Diffusion probability.
beta = 0.05  # Infection spreading probability.
gamma = 0.001  # Recovery probability.
alpha = 0.005  # Probability of losing immunity.
L = 200  # Side of the lattice.

I0 = 10  # Initial number of infected agents.

N_steps = 50000
n_runs = 5

# 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)  # All agents are susceptible initially.
status[0:I0] = 1  # Set the first I0 agents as infected.

S_runs = []
I_runs = []
R_runs = []

for run in range(n_runs):
    print(f"Starting run {run + 1}...")

    # Initialize agents position and status.
    x = np.random.randint(L, size=N_part)
    y = np.random.randint(L, size=N_part)
    status = np.zeros(N_part)
    status[0:I0] = 1  # Set the first I0 agents as infected.


    step = 0
    S = [N_part - I0]
    I = [I0]
    R = [0]
    S.append(N_part - I0)
    I.append(I0)
    R.append(0)

    running = True  # Flag to control the loop.
```

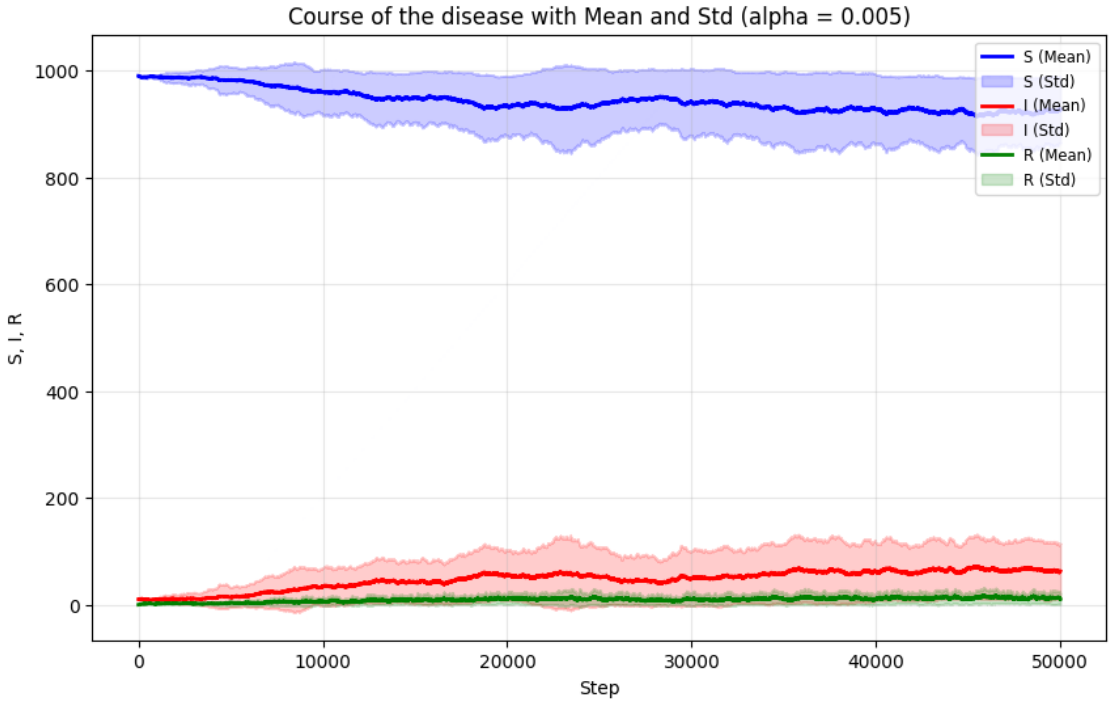
```
    while running:

        x, y, status = diffuse_spread_recover(x, y, status, d, beta, gamma,␣
 ↪alpha, L)

        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 step % 10000 ==0:
            print(f'step ={step}, susceptible = {S[-1]}, infectious ={I[-1]},␣
 ↪recovered = {R[-1]}, ')

        if step > N_steps:
            running = False

    S_runs.append(S)
    I_runs.append(I)
    R_runs.append(R)


print('Done.')

max_length = max(len(S) for S in S_runs)


for i in range(len(S_runs)):
    S_runs[i] += [S_runs[i][-1]] * (max_length - len(S_runs[i]))
    I_runs[i] += [I_runs[i][-1]] * (max_length - len(I_runs[i]))
    R_runs[i] += [R_runs[i][-1]] * (max_length - len(R_runs[i]))

S_array = np.array(S_runs)
I_array = np.array(I_runs)
R_array = np.array(R_runs)
```

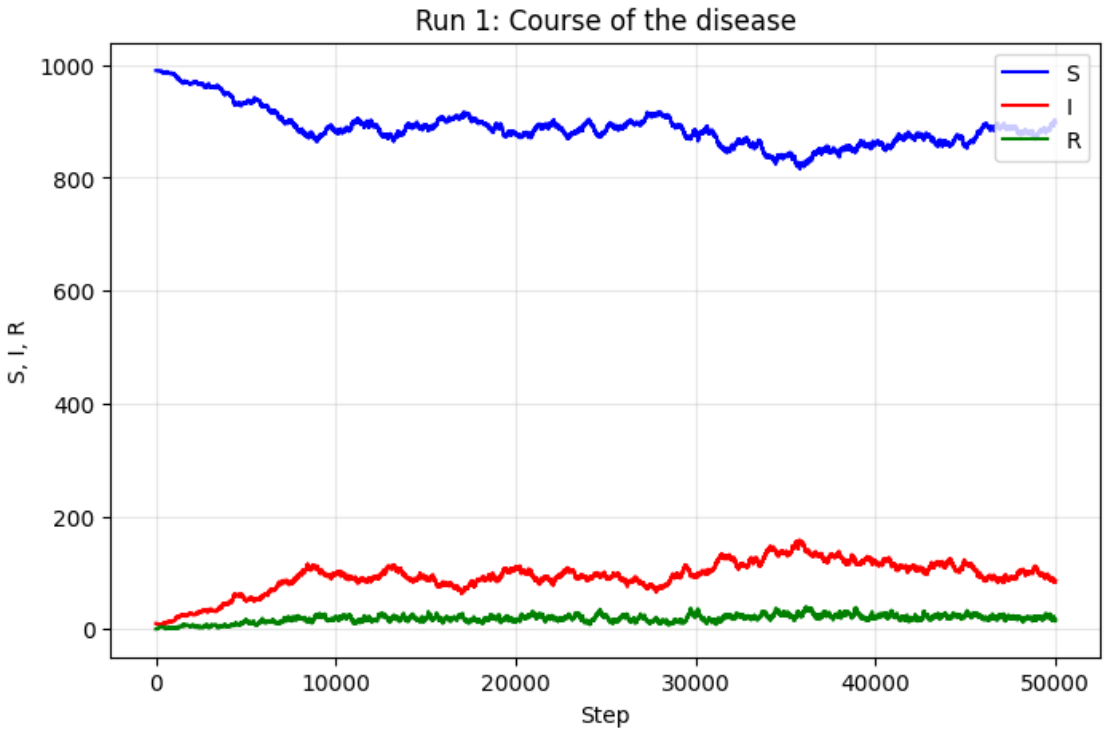
```
Starting run 1…
step =10000, susceptible = 881, infectious =99, recovered = 20,
step =20000, susceptible = 876, infectious =111, recovered = 13,
step =30000, susceptible = 880, infectious =94, recovered = 26,
step =40000, susceptible = 860, infectious =120, recovered = 20,
step =50000, susceptible = 898, infectious =86, recovered = 16,
Starting run 2…
step =10000, susceptible = 949, infectious =44, recovered = 7,
step =20000, susceptible = 872, infectious =100, recovered = 28,
step =30000, susceptible = 852, infectious =127, recovered = 21,
```

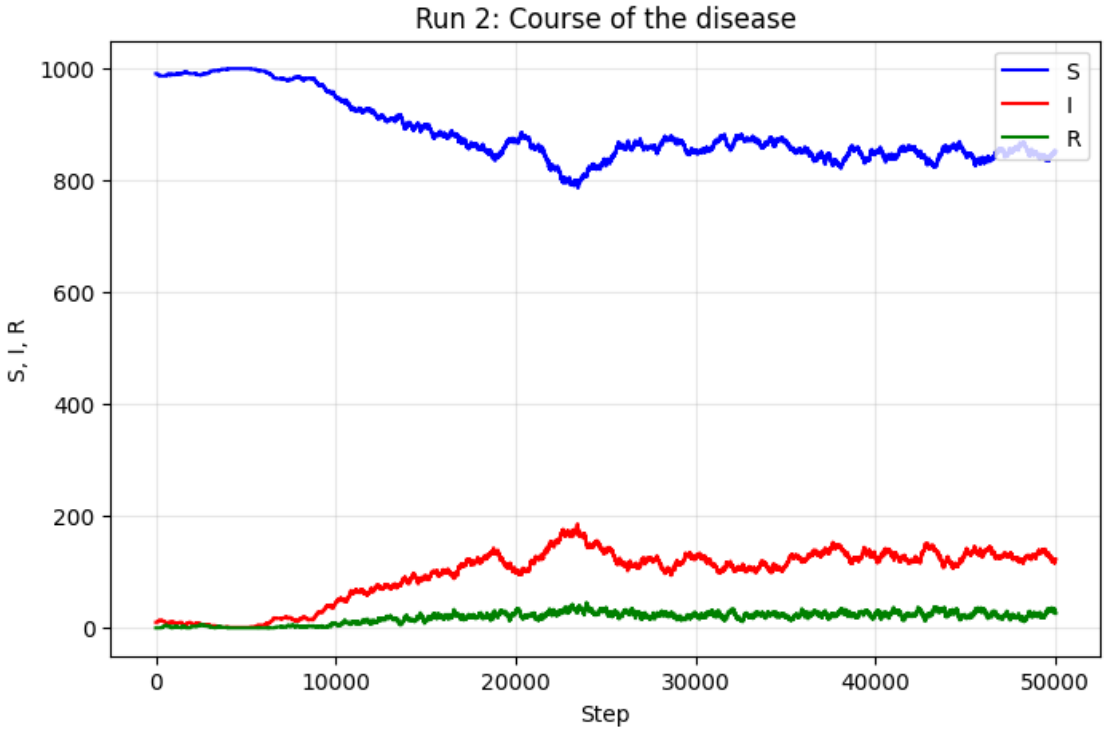
```
step =40000, susceptible = 848, infectious =125, recovered = 27,
step =50000, susceptible = 851, infectious =122, recovered = 27,
Starting run 3…
step =10000, susceptible = 978, infectious =19, recovered = 3,
step =20000, susceptible = 924, infectious =66, recovered = 10,
step =30000, susceptible = 973, infectious =26, recovered = 1,
step =40000, susceptible = 930, infectious =64, recovered = 6,
step =50000, susceptible = 883, infectious =105, recovered = 12,
Starting run 4…
step =10000, susceptible = 991, infectious =8, recovered = 1,
step =20000, susceptible = 998, infectious =1, recovered = 1,
step =30000, susceptible = 1000, infectious =0, recovered = 0,
step =40000, susceptible = 1000, infectious =0, recovered = 0,
step =50000, susceptible = 1000, infectious =0, recovered = 0,
Starting run 5…
step =10000, susceptible = 1000, infectious =0, recovered = 0,
step =20000, susceptible = 1000, infectious =0, recovered = 0,
step =30000, susceptible = 1000, infectious =0, recovered = 0,
step =40000, susceptible = 1000, infectious =0, recovered = 0,
step =50000, susceptible = 1000, infectious =0, recovered = 0,
Done.
```

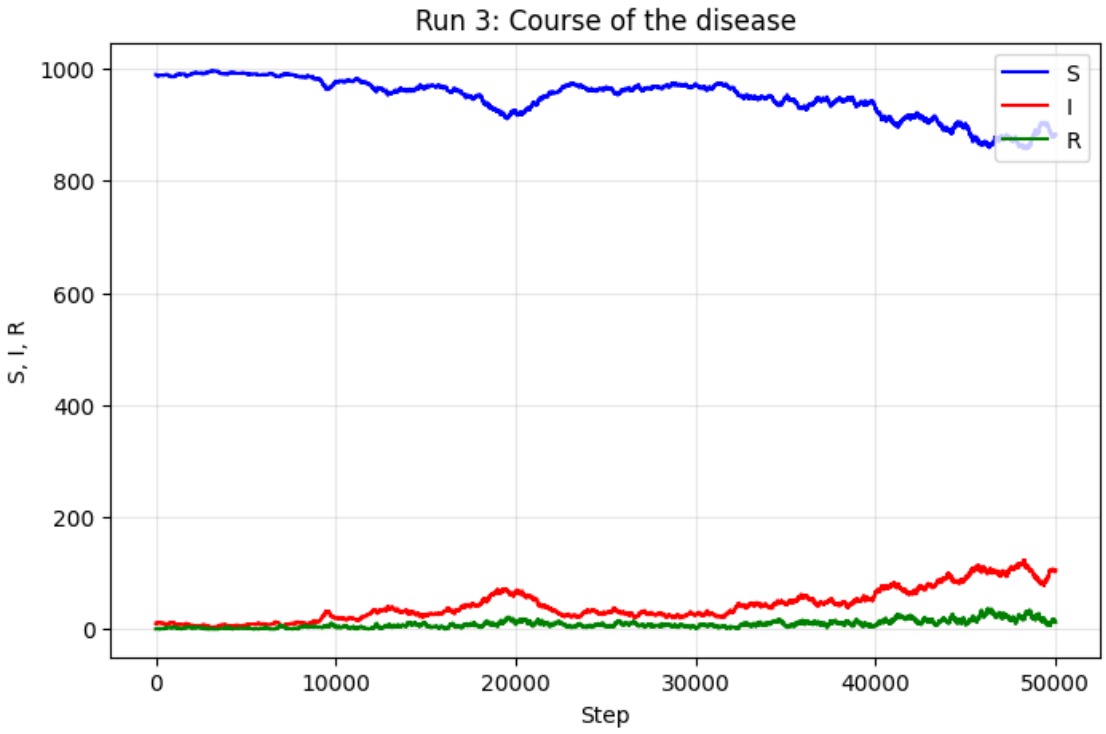
[54]:
```
S_mean = np.mean(S_array, axis=0)
I_mean = np.mean(I_array, axis=0)
R_mean = np.mean(R_array, axis=0)

S_std = np.std(S_array, axis=0)
I_std = np.std(I_array, axis=0)
R_std = np.std(R_array, axis=0)

t = np.arange(S_mean.shape[0])

plt.figure(figsize=(10, 6))

# Plot mean with error bars
plt.plot(t, S_mean, 'b-', label='S (Mean)', linewidth=2)
plt.fill_between(t, S_mean - S_std, S_mean + S_std, color='blue', alpha=0.2,␣
 ↪label='S (Std)')

plt.plot(t, I_mean, 'r-', label='I (Mean)', linewidth=2)
plt.fill_between(t, I_mean - I_std, I_mean + I_std, color='red', alpha=0.2,␣
 ↪label='I (Std)')

plt.plot(t, R_mean, 'g-', label='R (Mean)', linewidth=2)
plt.fill_between(t, R_mean - R_std, R_mean + R_std, color='green', alpha=0.2,␣
 ↪label='R (Std)')
plt.legend(loc='upper right', fontsize='small')
```

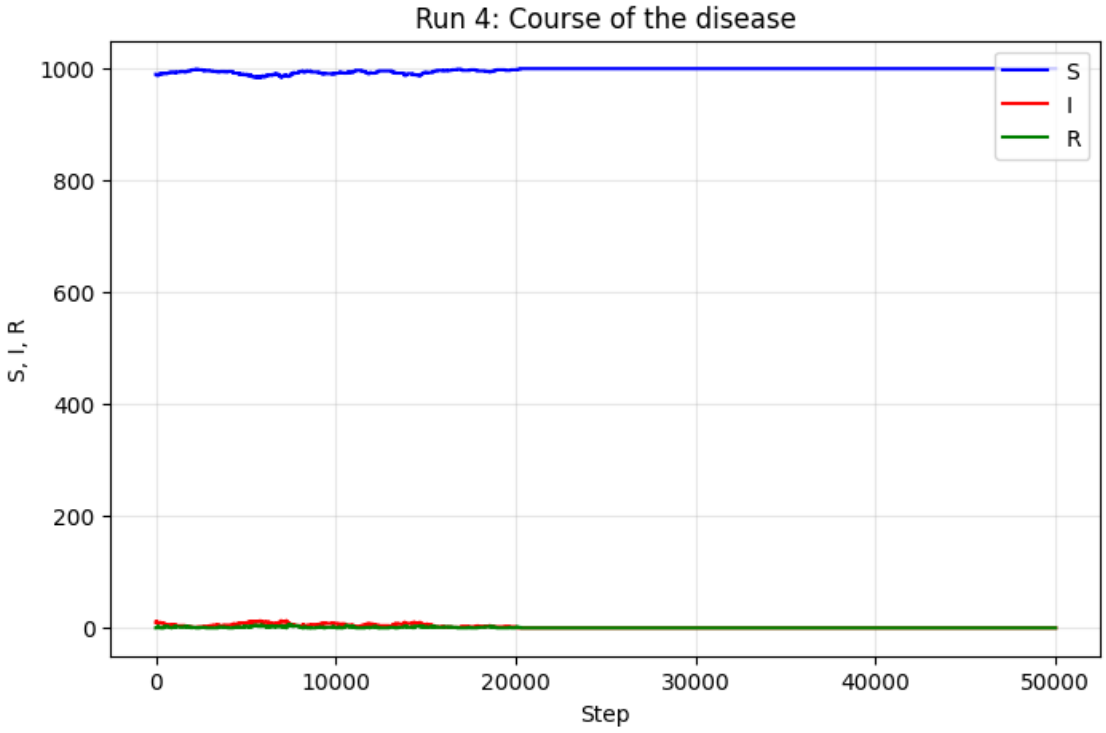
```
plt.title(f'Course of the disease with Mean and Std (alpha = {alpha})')
plt.xlabel('Step')
plt.ylabel('S, I, R')
plt.grid(alpha=0.3)
plt.savefig('P2_mean_and_std.png', dpi=300)
plt.show()


for run in range(len(S_runs)):
    t = np.arange(len(S_runs[run]))

    plt.figure(figsize=(8, 5))
    plt.plot(t, S_runs[run], 'b-', label='S')
    plt.plot(t, I_runs[run], 'r-', label='I')
    plt.plot(t, R_runs[run], 'g-', label='R')
    plt.legend(loc='upper right')
    plt.title(f'Run {run + 1}: Course of the disease')
    plt.xlabel('Step')
    plt.ylabel('S, I, R')
    plt.grid(alpha=0.3)
    plt.savefig(f'Run_{run + 1}.png', dpi=300)  #
    plt.show()
```

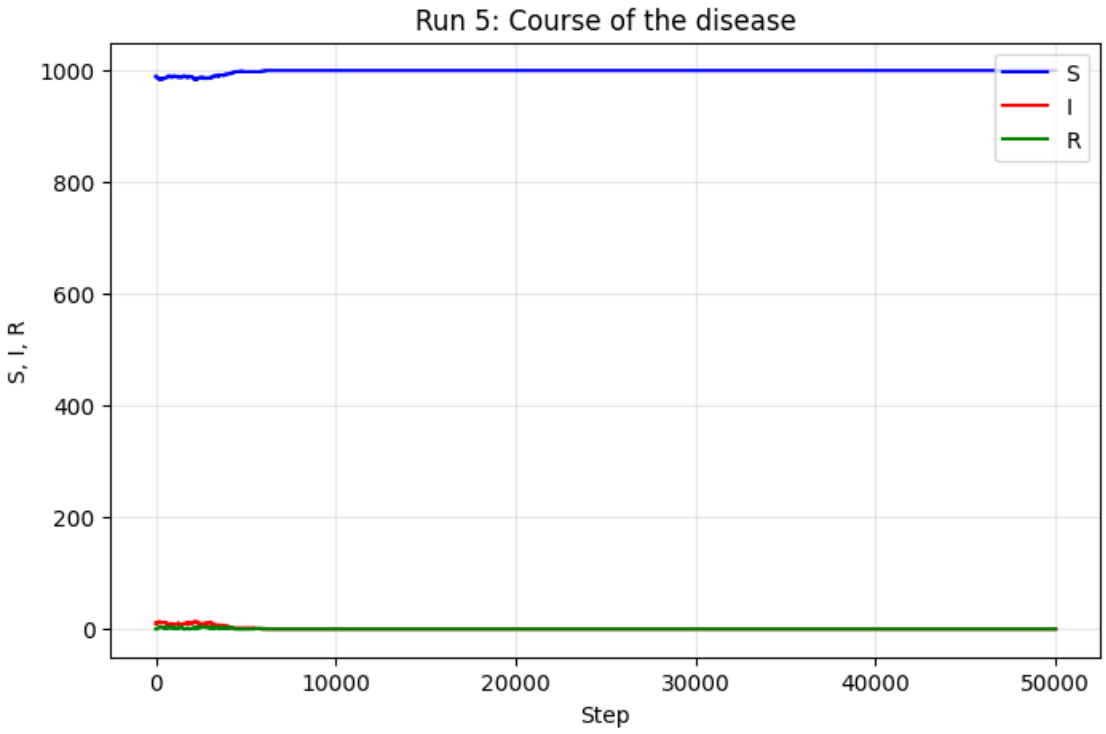



Run 1: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 2: Course of the disease
S
I
R
S, I, R
Step
0
200
400
600
800
1000
0
10000
20000
30000
40000
50000


Run 3: Course of the disease
S
I
R
S, I, R
1000
800
600
400
200
0
0
10000
20000
30000
40000
50000
Step


Run 4: Course of the disease
S
I
R
S, I, R
1000
800
600
400
200
0
0
10000
20000
30000
40000
50000
Step


Run 5: Course of the disease
S
I
R
S, I, R
Step
1000
800
600
400
200
0
0
10000
20000
30000
40000
50000

3.4

November 26, 2024

[59]:
```python
import numpy as np

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
```

[60]:
```python
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
```

[61]:
```
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
```

[62]:
```
def average_path_length(A):
    """
    Calculate the average path length for an adjacency matrix A.
    """
    L = matrix_path_length(A)
    L = L[np.where(L > 0)]  # delete length <= 0

    if len(L) == 0:
        return np.inf
    return np.mean(L)
```

[63]:
```
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
```

[64]:
```
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
```

0.1 Task 1

n = 100

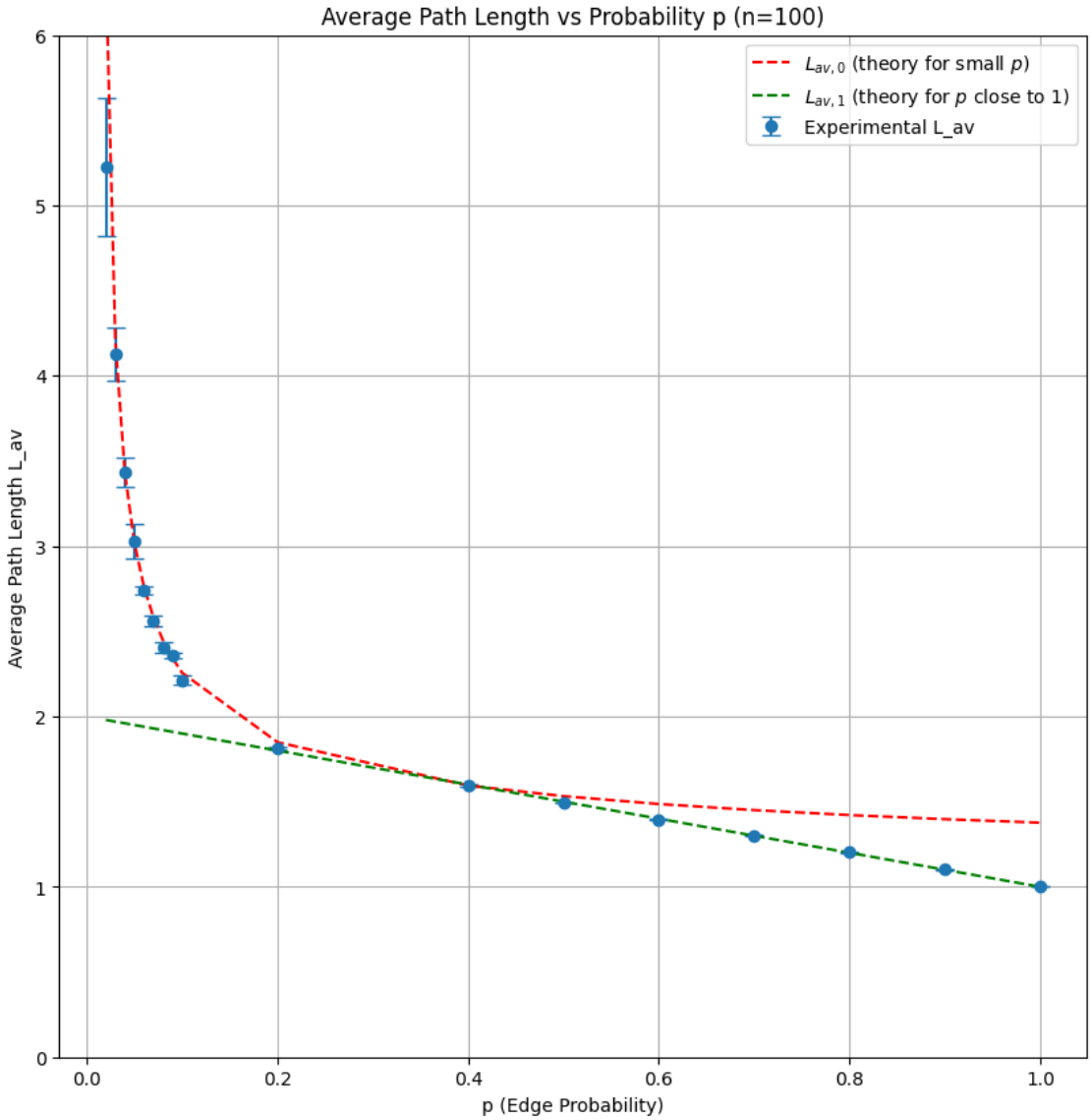
[ ]:
```
from matplotlib import pyplot as plt


n = 100  # nodes
ps = [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,␣
 ↪0.7, 0.8, 0.9, 1] # different p value
average_lengths = []
clustering_coeffs = []

length_std_errors = []
clustering_std_errors = []
```

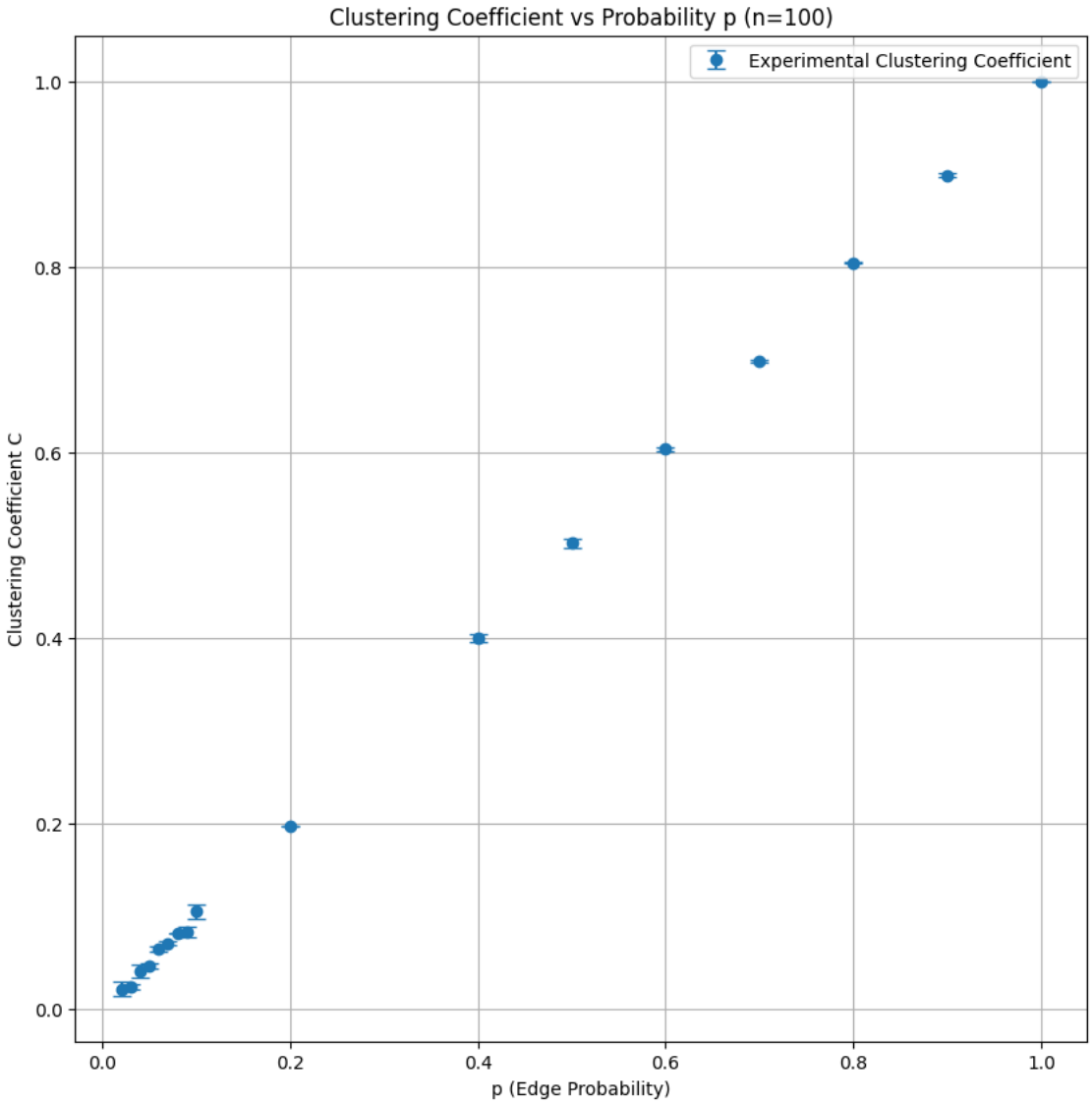
```
gamma = 0.57722 # Euler-Mascheroni constant

for p in ps:
    lengths = []
    coeffs = []
    for _ in range(3):
        A, _, _ = erdos_renyi_rg(n, p)
        avg_len = average_path_length(A)
        lengths.append(avg_len)
        clus_coeff = clustering_coefficient(A)
        coeffs.append(clus_coeff)

    average_lengths.append(np.mean(lengths))
    length_std_errors.append(np.std(lengths))
    clustering_coeffs.append(np.mean(coeffs))
    clustering_std_errors.append(np.std(coeffs))

    print(f'p = {p}, saved.')
```

```
p = 0.02, saved.
p = 0.03, saved.
p = 0.04, saved.
p = 0.05, saved.
p = 0.06, saved.
p = 0.07, saved.
p = 0.08, saved.
p = 0.09, saved.
p = 0.1, saved.
p = 0.2, saved.
p = 0.4, saved.
p = 0.5, saved.
p = 0.6, saved.
p = 0.7, saved.
p = 0.8, saved.
p = 0.9, saved.
p = 1, saved.
```

[68]:
```
ps = np.array(ps)
lav_0 = (np.log(n) - gamma) / np.log(ps * (n - 1)) + 0.5
lav_1 = 2 - ps

plt.figure(figsize=(10, 10))
plt.ylim(0, 6)
plt.errorbar(ps, average_lengths, yerr=length_std_errors, fmt='o',␣
 ↪label='Experimental L_av', capsize=5)
plt.plot(ps, lav_0, 'r--', label=r'$L_{av,0}$ (theory for small $p$)')
plt.plot(ps, lav_1, 'g--', label=r'$L_{av,1}$ (theory for $p$ close to 1)')
```

```
plt.xlabel('p (Edge Probability)')
plt.ylabel('Average Path Length L_av')
plt.title('Average Path Length vs Probability p (n=100)')
plt.legend()
plt.grid()
plt.savefig('3.4_Task1_P1.png')
plt.show()

plt.figure(figsize=(10, 10))

plt.errorbar(ps, clustering_coeffs, yerr=clustering_std_errors, fmt='o',␣
 ↪label='Experimental Clustering Coefficient', capsize=5)
plt.xlabel('p (Edge Probability)')
plt.ylabel('Clustering Coefficient C')
plt.title('Clustering Coefficient vs Probability p (n=100)')
plt.legend()
plt.grid()
plt.savefig('3.4_Task1_P2.png')
plt.show()
```


Average Path Length vs Probability p (n=100)
$L_{av,0}$ (theory for small p)
$L_{av,1}$ (theory for p close to 1)
Experimental L_av
Average Path Length L_av
p (Edge Probability)
0
1
2
3
4
5
6
0.0
0.2
0.4
0.6
0.8
1.0



0.2 Task 2

n = 200

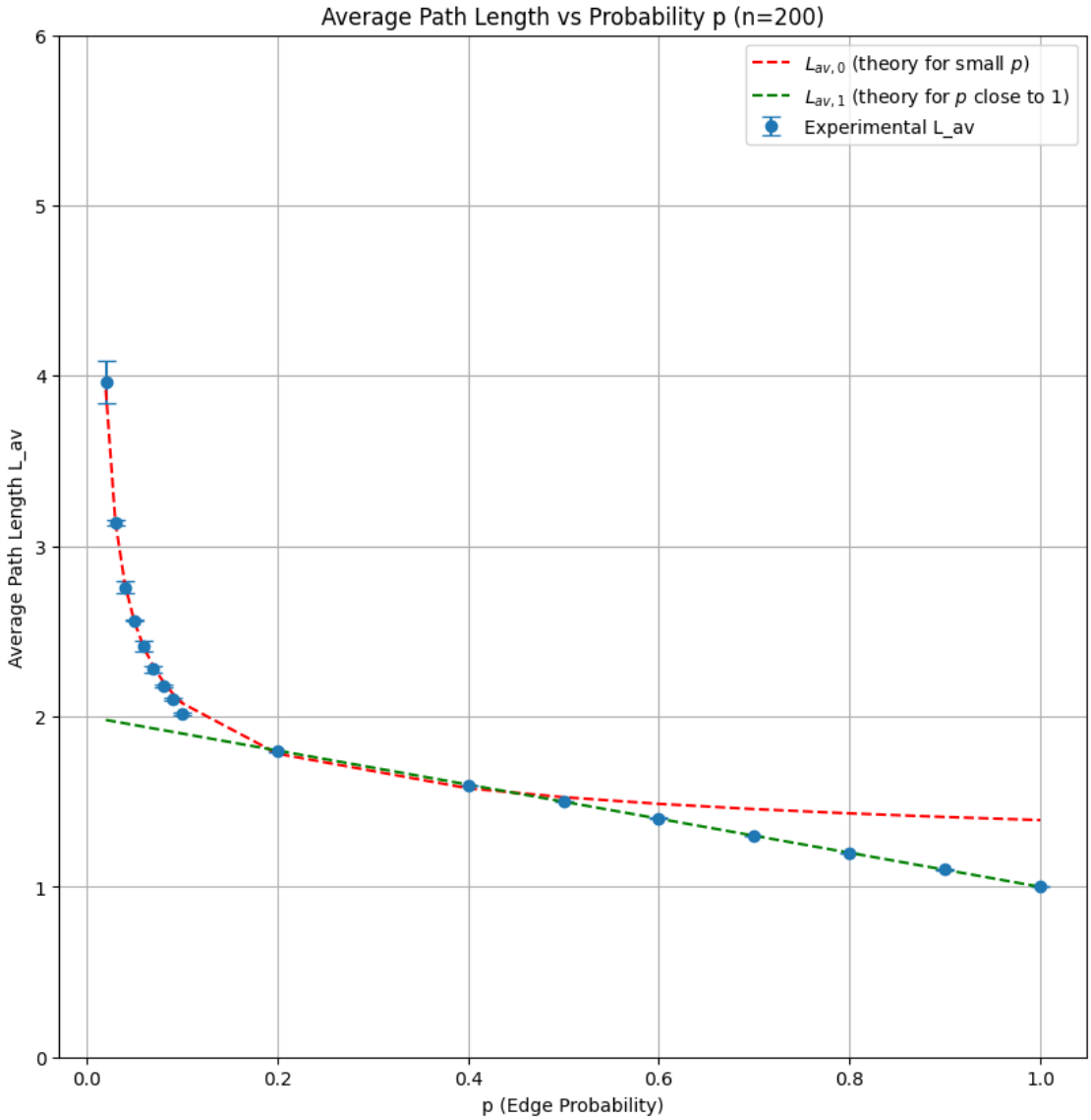
[69]:
```
from matplotlib import pyplot as plt


n = 200  # nodes
ps = [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,␣
 ↪0.7, 0.8, 0.9, 1] # different p value
average_lengths = []
clustering_coeffs = []
```

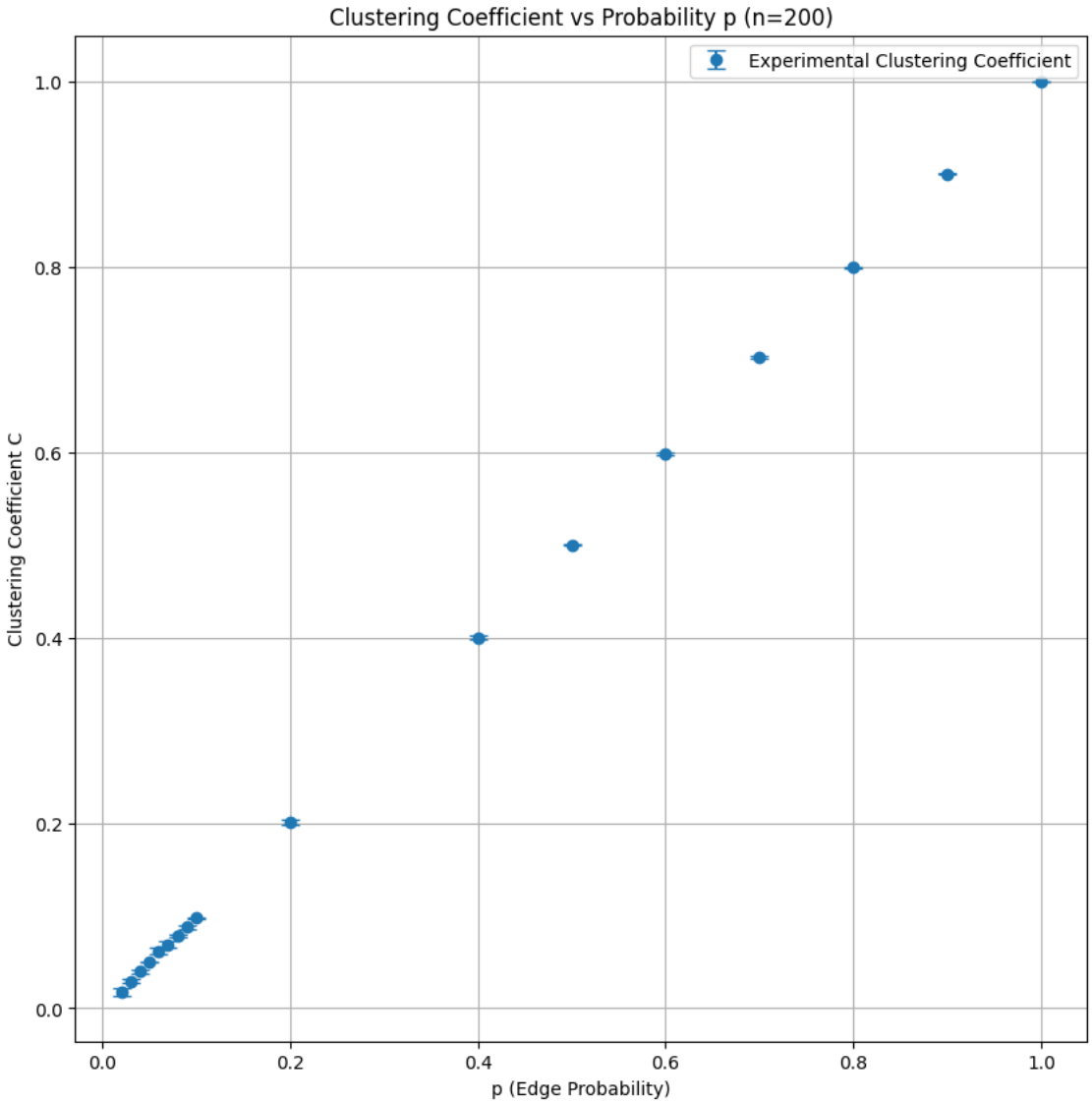
```
length_std_errors = []
clustering_std_errors = []
gamma = 0.57722 # Euler-Mascheroni constant

for p in ps:
    lengths = []
    coeffs = []
    for _ in range(3):
        A, _, _ = erdos_renyi_rg(n, p)
        avg_len = average_path_length(A)
        lengths.append(avg_len)
        clus_coeff = clustering_coefficient(A)
        coeffs.append(clus_coeff)

    average_lengths.append(np.mean(lengths))
    length_std_errors.append(np.std(lengths))
    clustering_coeffs.append(np.mean(coeffs))
    clustering_std_errors.append(np.std(coeffs))

    print(f'p = {p}, saved.')
```

```
p = 0.02, saved.
p = 0.03, saved.
p = 0.04, saved.
p = 0.05, saved.
p = 0.06, saved.
p = 0.07, saved.
p = 0.08, saved.
p = 0.09, saved.
p = 0.1, saved.
p = 0.2, saved.
p = 0.4, saved.
p = 0.5, saved.
p = 0.6, saved.
p = 0.7, saved.
p = 0.8, saved.
p = 0.9, saved.
p = 1, saved.
```

[70]:
```
ps = np.array(ps)
lav_0 = (np.log(n) - gamma) / np.log(ps * (n - 1)) + 0.5
lav_1 = 2 - ps

plt.figure(figsize=(10, 10))
plt.ylim(0, 6)
plt.errorbar(ps, average_lengths, yerr=length_std_errors, fmt='o',␣
 ↪label='Experimental L_av', capsize=5)
```

```
plt.plot(ps, lav_0, 'r--', label=r'$L_{av,0}$ (theory for small $p$)')
plt.plot(ps, lav_1, 'g--', label=r'$L_{av,1}$ (theory for $p$ close to 1)')
plt.xlabel('p (Edge Probability)')
plt.ylabel('Average Path Length L_av')
plt.title('Average Path Length vs Probability p (n=200)')
plt.legend()
plt.grid()
plt.savefig('3.4_Task2_P3.png')
plt.show()

plt.figure(figsize=(10, 10))
plt.errorbar(ps, clustering_coeffs, yerr=clustering_std_errors, fmt='o',␣
 ↪label='Experimental Clustering Coefficient', capsize=5)
plt.xlabel('p (Edge Probability)')
plt.ylabel('Clustering Coefficient C')
plt.title('Clustering Coefficient vs Probability p (n=200)')
plt.legend()
plt.grid()
plt.savefig('3.4_Task2_P4.png')
plt.show()
```


Average Path Length vs Probability p (n=200)
$L_{av,0}$ (theory for small p)
$L_{av,1}$ (theory for p close to 1)
Experimental L_av
Average Path Length L_av
p (Edge Probability)
0
1
2
3
4
5
6
0.0
0.2
0.4
0.6
0.8
1.0


Clustering Coefficient vs Probability p (n=200)
Experimental Clustering Coefficient
Clustering Coefficient C
p (Edge Probability)
0.0
0.2
0.4
0.6
0.8
1.0