3.2_positive_delay

November 26, 2024

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

[2]:
```python
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]:
```python
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]
```

```python
    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]:
```python
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]:

```python
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)
```

```python
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(
```

```python
                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
```