# 3.1_100R

November 26, 2024

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

[8]:
```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
```

[9]:
```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)
```

```python
    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]:

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

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

[ ]:

```python
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()
```

```python
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.
```