Computations

Sohyun Park

Table of contents

1 Localised external potential

The localised external potential added to the system (Turci and Wilding 2021):

$$V_{\text{ext}}(x) = \varepsilon_w \left[\cos\left(\frac{\pi x}{d}\right) + 1\right] \times H(d-x)H(x+d), \tag{1}$$

where $H(x)$ is the Heaviside function, $d = \sigma$, and ε_w represents the repulsive barrier strength.

Turci, Francesco, and Nigel B. Wilding. 2021. "Wetting Transition of Active Brownian Particles on a Thin Membrane." *Physical Review Letters* 127 (23): 238002. https://doi.org/10.1103/PhysRevLett.127.238002.

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
sigma = 1.0  # Particle diameter
d = sigma    # Barrier width
ew_values = np.arange(20, 55, 5)  # Barrier strength values from 20 to 50 incrementing by 5

# Define the potential function
def V_ext(x, epsilon_w, d):
    H = lambda x: np.heaviside(x, 0.5)  # Heaviside step function
    return epsilon_w * (np.cos(np.pi * x / d) + 1) * H(d - x) * H(x + d)

# x range
x = np.linspace(-2 * d, 2 * d, 500)

# Plot
plt.figure(figsize=(10, 6))
for ew in ew_values:
    V = V_ext(x, ew, d)
    plt.plot(x, V, label=f"$\\epsilon_w = {ew}$")
plt.axvline(-d, color='gray', linestyle='--', label=r"$x = -d$", linewidth=0.8)
plt.axvline(d, color='gray', linestyle='--', label=r"$x = d$", linewidth=0.8)
plt.axhline(0, color='black', linewidth=0.5)
plt.xlabel("$x$", fontsize=12)
plt.ylabel("$V_{\\text{ext}}(x)$", fontsize=12)
plt.legend(fontsize=10, loc='upper right')
plt.grid(alpha=0.5)
plt.show()
```

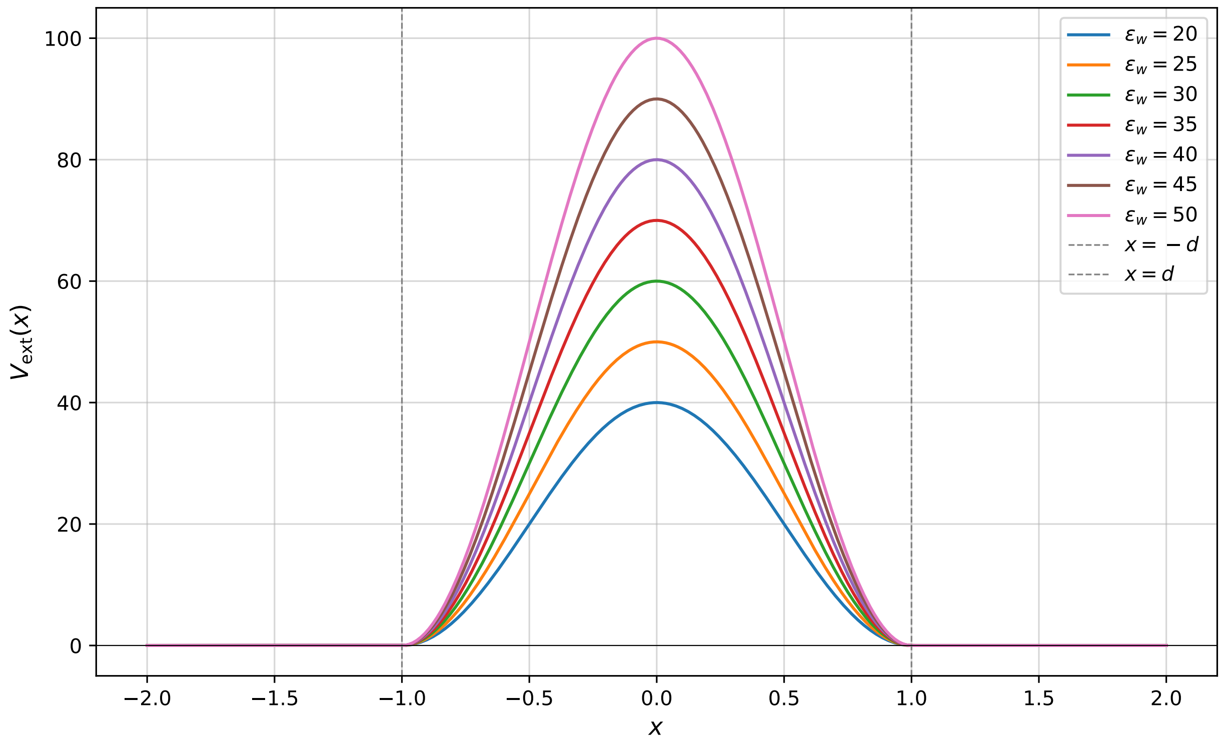

Figure 1: External potential $V_{\text{ext}}(x)$ for different ϵ_w

Note

Intermediate values can be thought of as representing a thin porous membrane with nonzero crossing probability.

Callout tip

Tip

Note that there are five types of callouts, including: `note`, `tip`, `warning`, `caution`, and `important`.

2 Asymmetry order parameter

The degree of asymmetry of the instantaneous density profile $\rho(x,t)$ with respect to the barrier location is quantified as:

$$\mathcal{A}(t) = \left| \frac{\int_0^{L_x/2} \rho(x,t)\,dx - \int_{-L_x/2}^{0} \rho(x,t)\,dx}{(\rho - \rho_{\mathrm{LD}})L_x} \right|.$$

In the steady state, the average $\overline{\mathcal{A}(t)}$ (over time and distinct initial conditions) provides a measure of the typical asymmetry of the liquid region with respect to $x = 0$.

```python
import pandas as pd
import numpy as np
import re
import os
import matplotlib.pyplot as plt

def load_profile(filename):
    """
    Load LAMMPS profile dump file into a pandas DataFrame.
    Reads multiple timesteps and returns a DataFrame containing data for all timesteps.
    """
    data = []
    with open(filename, 'r') as f:
        lines = f.readlines()

    i = 0
    while i < len(lines):
        line = lines[i].strip()
        # Skip comment lines and empty lines
        if line.startswith('#') or line == '':
            i += 1
            continue
        tokens = line.split()
        if len(tokens) == 3:
            # Try to parse the line as Timestep and Number-of-chunks
            try:
                timestep = int(tokens[0])
                chunk_count = int(tokens[1])
                # Total-count is ignored
            except ValueError:
                # If parsing fails, skip this line
                i += 1
                continue
            # Read the data for this timestep
            data_start = i + 1
            data_end = data_start + chunk_count
            for j in range(data_start, data_end):
                if j >= len(lines):
                    break  # Prevent index out of range
                dl = lines[j].strip()
                if dl == '':
                    continue
                dl_tokens = dl.split()
                if len(dl_tokens) != 4:
                    continue
                try:
                    chunk = int(dl_tokens[0])
                    coord1 = float(dl_tokens[1])
                    ncount = int(float(dl_tokens[2]))  # Convert to float first to handle possibl
                    density = float(dl_tokens[3])
                    data.append({
                        'Timestep': timestep,
                        'Chunk': chunk,
                        'Coord1': coord1,
                        'Ncount': ncount,
                        'density': density
                    })
                except ValueError:
                    continue  # Skip lines with invalid data
            i = data_end
        else:
            i += 1

    df = pd.DataFrame(data)
    return df  # Return the DataFrame with all timesteps

def calculate_asymmetry(data, lx, rho_ld, rho):
    """
    Compute the asymmetry order parameter A(t) with proper normalization.
    """
    x_bins = data['Coord1']  # Bin center positions along x-axis
    densities = data['density']  # Density per bin

    # Ensure that x_bins and densities are sorted according to x_bins
    sorted_indices = np.argsort(x_bins)
    x_bins = x_bins.iloc[sorted_indices].reset_index(drop=True)
    densities = densities.iloc[sorted_indices].reset_index(drop=True)
    bin_width = abs(x_bins.iloc[1] - x_bins.iloc[0])  # Calculate bin width

    # Split the bins into left and right of the barrier (x = 0)
    left_mask = x_bins < 0
    right_mask = x_bins > 0

    left_bins = densities[left_mask]
    right_bins = densities[right_mask]

    # Integrate density over the left and right regions
    left_integral = (left_bins * bin_width).sum()
    right_integral = (right_bins * bin_width).sum()

    # Numerator for A(t)
    numerator = abs(right_integral - left_integral)

    # Denominator normalisation
    denominator = (rho - rho_ld) * lx

    # Asymmetry order parameter
    asymmetry = numerator / denominator if denominator != 0 else 0

    return asymmetry

def plot_asymmetry_vs_time(filename, lx, rho_ld, rho, taur):
    """
    Compute and plot the asymmetry order parameter A(t) as a function of t/taur for a given dump
    Estimate the steady-state relaxation time tau_c based on the rate of change of A(t).
    """
    data = load_profile(filename)
    # Compute asymmetry at each timestep
    asymmetry_list = []
    timesteps = []
    grouped = data.groupby('Timestep')
    for timestep, df_timestep in grouped:
        asymmetry = calculate_asymmetry(df_timestep, lx=lx, rho_ld=rho_ld, rho=rho)
        asymmetry_list.append(asymmetry)
        timesteps.append(timestep)

    # Convert to numpy arrays
    asymmetry_array = np.array(asymmetry_list)
    timestep_array = np.array(timesteps)

    # Sort by timestep
    sorted_indices = np.argsort(timestep_array)
    timestep_array = timestep_array[sorted_indices]
    asymmetry_array = asymmetry_array[sorted_indices]

    # Calculate t / taur
    dt = 0.00004 * taur
    t_over_taur = timestep_array * dt / taur  # Simplifies to timestep_array * 0.00004

    # Extract epsilon_w from filename
    # filename format: 'wet.<timestamp>.eps.<epsilon_w>.ly.<ly>.dump'
    pattern = r'wet\..*\.eps\.(\d+)\.ly\.(\d+)\.dump'
    match = re.match(pattern, os.path.basename(filename))
    if match:
        epsilon_w = match.group(1)
        ly = match.group(2)
    else:
        epsilon_w = 'Unknown'
        ly = 'Unknown'

    # Plot A(t) vs t / taur
    plt.figure(figsize=(10, 6))
    plt.plot(t_over_taur, asymmetry_array, linestyle='-', label=r'$\mathcal{A}(t)$')
    plt.xlabel(r'$t / \tau_r$', fontsize=14)
    plt.ylabel(r'$\mathcal{A}(t)$', fontsize=14)
    plt.text(0.05, 0.95, f'$\\epsilon_w$ = {epsilon_w}, $L_y$ = {ly}',
            transform=plt.gca().transAxes,
            fontsize=12,
            verticalalignment='top',
            bbox=dict(facecolor='white', edgecolor='black', boxstyle='round', pad=0.5))
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    return t_over_taur, asymmetry_array

def compute_taur():
    """
    Compute the persistence time tau_r based on the LAMMPS input script variables.
    """
    sigma = 1.0
    wcaepsilon = 1.0
    friction = 50.0
    Pe = 50.0
    activity = 24 * wcaepsilon / (sigma * friction)  # activity = 24*epsilon/(sigma*friction)
    T = activity * friction * sigma / (3 * Pe)  # temperature
    Dr = 3 * T / (friction * sigma**2)  # rotational diffusion
    taur = 1 / Dr  # persistence time
    return taur

# Main script
if __name__ == "__main__":
    folder = "dumps"  # Folder containing dump files
    lx = 240.0  # Total system size in x-direction
    rho_ld = 0.15  # Low-density MIPS value
    rho = 0.5  # Total system density
    # Compute taur
    taur = compute_taur()
    print(f"Computed tau_r (persistence time): {taur}")

    # Plot asymmetry vs time for multiple dump files
    for eps in range(20, 51, 5):  # Iterate over eps values: 20, 25, 30, ..., 50
        filename = os.path.join(folder, f'wet.241206.0213.eps.{eps}.ly.120.dump')
        if os.path.exists(filename):
            t_over_taur, asymmetry_array = plot_asymmetry_vs_time(filename, lx=lx, rho_ld=rho_ld,
        # else:
            # print(f"File {filename} does not exist.")
```

```
Computed tau_r (persistence time): 104.16666666666667
```

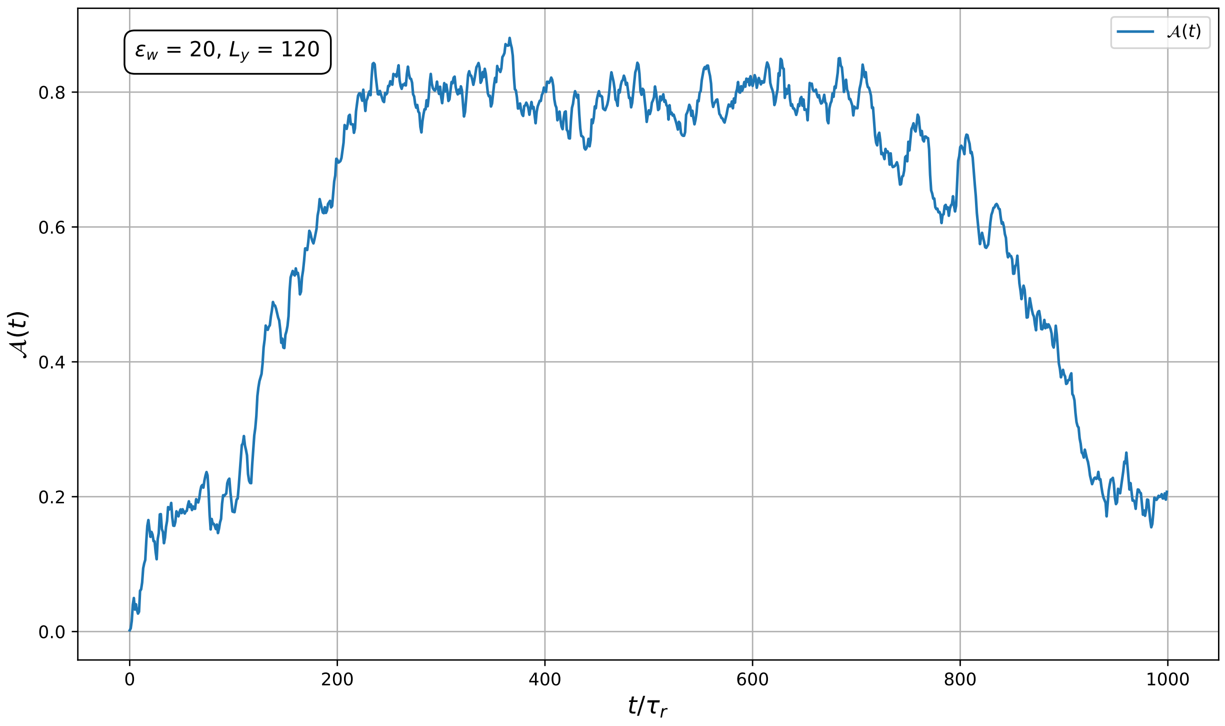

Figure 2: Asymmetry order parameter $\mathcal{A}(t)$ over time for $\epsilon_w = 20$

3 Slurm Jobs

- **slurm-10797760**: It took 20 minutes to compute 2,500,000 steps.
  $\rightarrow$ This means I need about 27 hours to compute 4 $\times$ 50,000,000 steps for four epsilons.
- **slurm-10802470**: Lost atoms in all epsilons.
- **slurm-10814916**: Cancelled, since I realised what was happening was:
  ABPs interacted to form clusters before the wall was even there.
  So, I commented out the following lines:

```
# variable relaxation equal ceil(10*${taur}/${dt})
# thermo_style custom step time v_tscaled pe density press
# thermo ${snapshot} # output thermo data every snapshot steps
# run ${relaxation}
```

- **slurm-10815371**: Lost atoms again in `eps=20` even when `dt = 0.00002*${taur}`.
- **slurm-10815451**: Lost atoms again for all epsilons even when `dt = 0.00001*${taur}`...
  I realised I didn't include `timestep ${dt}`! Now it works perfectly.

- **slurm-10824222**: All of them were extremely successful! It took about 8 hours to compute 4 different potentials for 4 × 25,000,000 steps.
  Removed OMPI error by adding `export OMPI_MCA_mca_base_component_show_load_errors=0`.

- **slurm-10846320**: As soon as I moved to 3D, I experienced, for the first time, an issue where not enough atoms were created:

  ```
  Created orthogonal box = (-50 -12 -12) to (50 12 12)
    16 by 2 by 2 MPI processor grid
  WARNING: Only inserted 37118 particles out of 43200 (../create_atoms.cpp:894)
  Created 37118 atoms
    using lattice units in orthogonal box = (-50 -12 -12) to (50 12 12)
    create_atoms CPU = 334.523 seconds
  ```

  Message to Francesco:

  > I fixed it! I realised the random insertion was basically the Monte Carlo placement of atoms in the box, so I relaxed the constraint for the overlap parameter to be 0.8 rather than ${sigma} (which is 1), then added a short soft-potential relaxation before switching to WCA parameters, allowing atoms to move apart. Then I ran a short FIRE minimisation before proceeding with ABP dynamics, and boom! It all works out perfectly! It took about 6 hours of reading LAMMPS documentation to solve this issue, but it worked, so I was so happy to share!

  Before:

  ```
  # Define simulation box and atoms
  region box block -${lx} ${lx} -${ly} ${ly} -${lz} ${lz}
  create_box 1 box # allocate one atom type within this region
  create_atoms 1 random ${npart} ${seed} box overlap ${sigma} # create 'npart' particles dispe

  # Pair potential
  pair_style lj/cut 3.0 # LJ potential truncated at rc
  pair_coeff * * ${wcaepsilon} ${sigma} ${rc} # set WCA parameters for ALL particles (indicate
  pair_modify shift yes # shift potential to zero at rc
  mass 1 1 # set particle mass to 1 in LJ units

  # Neighbor list settings of every = 1 and delay = 0 are required for the minimisation, read
  neighbor 0.3 bin
  neigh_modify every 1 delay 0 check yes

  # ABP dynamics
  fix 1 all abp2d ${T} ${friction} ${activity} ${seed} # implements fix_abp
  ```

  After:

```
# Define simulation box and atoms
region box block -${lx} ${lx} -${ly} ${ly} -${lz} ${lz}
create_box 1 box # allocate one atom type within this region

# ---- Modified Section for Atom Creation and Pre-Relaxation ----

# Attempt to create all atoms with a slightly relaxed overlap constraint
# Setting 'overlap 0.8' means atoms can be placed if their pairwise distance
# is at least 0.8*diameter (instead of full 1.0*diameter), making insertion easier.
create_atoms 1 random ${npart} ${seed} box overlap 0.8 # create 'npart' particles dispersed a

# Immediately after insertion, a soft-potential relaxation to push particles apart.
# Switch to a soft pair style:
pair_style soft 1.122462  # The cutoff isn't crucial here; just needs to be >= sigma
pair_coeff * * 10.0       # Initial softness amplitude; 10.0 is fairly large repulsion
mass 1 1 # set particle mass to 1 in LJ units

# A short run to resolve overlaps:
fix relax all nve
thermo_style custom step pe press density
thermo 1000

# Run for some steps to separate particles:
run 10000
unfix relax

# Pair potential
pair_style lj/cut 3.0 # LJ potential truncated at rc
pair_coeff * * ${wcaepsilon} ${sigma} ${rc} # set WCA parameters for ALL particles (indicate
pair_modify shift yes # shift potential to zero at rc

# Neighbor list settings of every = 1 and delay = 0 are required for the minimisation, read
neighbor 0.3 bin
neigh_modify every 1 delay 0 check yes

# Run a short minimisation before starting ABP:
min_style fire
minimize 1.0e-9 1.0e-9 1000 1000
reset_timestep 0

# ABP dynamics
fix 1 all abp ${T} ${friction} ${activity} ${seed} # implements fix_abp
```