Computations

Sohyun Park

Table of contents

# 1. Localised external potential

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

where is the Heaviside function, , and represents the repulsive barrier strength.

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 1.1: External potential for different |

|  |
| --- |
| 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 with respect to the barrier location is quantified as:

In the steady state, the average (over time and distinct initial conditions) provides a measure of the typical asymmetry of the liquid region with respect to .

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 possible decimals  
 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, start\_threshold, start\_window\_fraction):  
 """  
 Compute and plot the asymmetry order parameter A(t) as a function of t/taur for a given dump file.  
 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'  
  
 # Compute the absolute rate of change of A(t)  
 delta\_A = np.abs(np.diff(asymmetry\_array))  
 delta\_t = np.diff(t\_over\_taur)  
  
 # Avoid division by zero  
 delta\_t[delta\_t == 0] = np.nan  
  
 # Compute the rate of change per unit time  
 rate\_of\_change = delta\_A / delta\_t  
 i = 0  
 # while i < len(rate\_of\_change):  
 # print(f"rate\_of\_change {i}: {rate\_of\_change[i]}")  
 # i += 1  
  
 # Define a threshold for the rate of change  
 # For example, start\_threshold \* 100% of the maximum A(t) per unit time  
 threshold = start\_threshold \* np.max(asymmetry\_array)  
 print(f"threshold: {threshold}, np.max(asymmetry\_array): {np.max(asymmetry\_array)}")  
  
 # Identify when rate of change consistently remains below the threshold  
 window\_size = max(1, int(len(rate\_of\_change) \* start\_window\_fraction)) # start\_window\_fraction \* 100% of data length  
 print(f"window\_size: {window\_size}")  
 below\_threshold = rate\_of\_change < threshold  
 # Use a rolling window to check for consecutive points below threshold  
 below\_threshold\_series = pd.Series(below\_threshold)  
 rolling\_sum = below\_threshold\_series.rolling(window=window\_size, min\_periods=1).sum()  
 # Find the index where the rolling sum equals the window size (steady state)  
 steady\_indices = np.where(rolling\_sum == window\_size)[0]  
 if len(steady\_indices) > 0:  
 tau\_c\_index = steady\_indices[0] + 1 # +1 because rate\_of\_change is one element shorter  
 tau\_c\_timestep = timestep\_array[tau\_c\_index]  
 tau\_c = t\_over\_taur[tau\_c\_index]  
 # print(f"Estimated relaxation time tau\_c: {tau\_c:.2f} tau\_r (Timestep: {tau\_c\_timestep})")  
 else:  
 # print("Could not determine relaxation time tau\_c; rate of change did not consistently fall below threshold.")  
 tau\_c = None  
  
 # Plot A(t) vs t / taur  
 plt.figure(figsize=(10, 6))  
 plt.plot(t\_over\_taur, asymmetry\_array, linestyle='-', label=r'$\mathcal{A}(t)$')  
 if tau\_c is not None:  
 plt.axvline(x=tau\_c, color='red', linestyle='--', label=r'Estimated $\tau\_c$')  
 plt.xlabel(r'$t / \tau\_r$', fontsize=14)  
 plt.ylabel(r'$\mathcal{A}(t)$', fontsize=14)  
 # plt.title(f'Asymmetry Order Parameter over Time\n($\epsilon\_w$ = {epsilon\_w}, $L\_y$ = {ly})', fontsize=16)  
 plt.legend()  
 plt.grid(True)  
 plt.tight\_layout()  
 plt.show()  
  
 return t\_over\_taur, asymmetry\_array, tau\_c  
  
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}")  
  
 # Parameters for steady-state detection  
 start\_threshold = 0.1 # fraction of max(A) for start detection  
 start\_window\_fraction = 0.2 # fraction of data for rolling window at start detection  
  
 # 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, tau\_c = plot\_asymmetry\_vs\_time(filename, lx=lx, rho\_ld=rho\_ld, rho=rho, taur=taur,  
 start\_threshold=start\_threshold,  
 start\_window\_fraction=start\_window\_fraction)  
 if tau\_c is not None:  
 print(f"Estimated steady-state relaxation time tau\_c: {tau\_c:.2f} tau\_r")  
 else:  
 print("Could not determine relaxation time tau\_c.")  
 # else:  
 # print(f"File {filename} does not exist.")

Computed tau\_r (persistence time): 104.16666666666667  
threshold: 0.08804001207000334, np.max(asymmetry\_array): 0.8804001207000334  
window\_size: 199

|  |
| --- |
| Figure 2.1: Asymmetry order parameter over time for = 20 |

Estimated steady-state relaxation time tau\_c: 199.01 tau\_r

# 3. Slurm Jobs

* **slurm-10797760**: It took 20 minutes to compute 2,500,000 steps.  
  → This means I need about 27 hours to compute 4 × 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.

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>.