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):  
 """  
 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'  
  
 # 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, rho=rho, taur=taur)  
 # else:  
 # print(f"File {filename} does not exist.")

Computed tau\_r (persistence time): 104.16666666666667

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

# 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.
* **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 dispersed at random in the box  
    
  # Pair potential  
  pair\_style lj/cut 3.0 # LJ potential truncated at rc  
  pair\_coeff \* \* ${wcaepsilon} ${sigma} ${rc} # set WCA parameters for ALL particles (indicated by the two asterisks)  
  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 'minimize' document  
  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 at random in the box  
    
  # 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 (indicated by the two asterisks)  
  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 'minimize' document  
  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

# 4. OVITO

## 4.1 Coordination Analysis

To visualise a meaningful local density, I used the **Coordination Number** (the number of neighbours within a defined radius) as a property for each particle. The **Coordination Analysis** modifier in OVITO provides an approximate measure of local density by counting the neighbours within a specified cutoff radius. In my case, the neighbour cutoff radius, derived from the LAMMPS simulation setup, is , where is the buffer added to the neighbour list with neighbor 0.3 bin.

The local density is calculated as:

where is the effective cutoff radius. The denominator represents the volume of a sphere with radius , corresponding to the local volume used for density estimation. In the **Compute Property** modifier, I used the following expression to calculate the density:

Coordination / (4/3 \* pi \* 1.422462^3)

This output property was named **Density**, as it was subsequently used for colour coding in the visualisation.

|  |
| --- |
| Modifcations |

Modifcations

To visualise the density, I applied a **Color Coding** modifier to map the density property to a colour scale. Regions with high coordination numbers (dense regions) are highlighted in brighter colours, while sparse regions appear dimmer.

The number of histogram bins in the Coordination Analysis was carefully chosen to balance resolution and computational efficiency. The number of bins is given by:

The choice of a bin width of LJ units is grounded in standard practices for molecular dynamics simulations. While the specific value is not prescribed in the literature, texts like Frenkel and Smit’s *Understanding Molecular Simulation, 3rd Edition, Chapter 5.1* emphasises the importance of sufficient resolution to capture sharp features in dense systems.

A bin width of represents 0.2% of the particle size, which is sufficient to resolve narrow features, such as the first peak in the RDF. This strikes a balance between resolution and computational cost. A smaller bin width, such as , would provide even higher resolution but require significantly more memory and computational resources (unless I’m studying very sharp features, such as crystalline structures). Conversely, a bin width of would reduce the number of bins to approximately 284, which I found empirically oversmooths the RDF in dense systems like mine ().

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>.