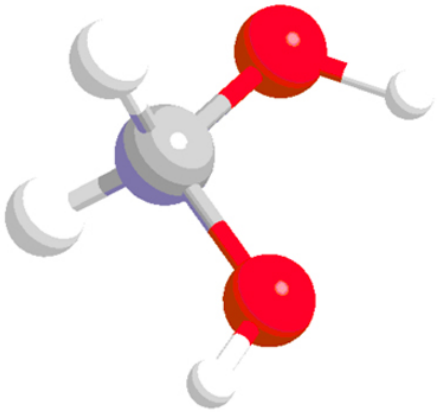
Monte Carlo Simulation of Surface Tension in a Lennard-Jones Fluid



[3]

Ivar Eftedal
Christopher Ljosland Strand
Frederic Ljosland Strand

Project Task in INF203 at the University of Life Sciences

June 2025

1 Introduction

Group composition and collaboration

The group consists of Ivar Eftedal, Christopher Strand and Frederic Strand. For a successful project, we first established the personas and their user stories to guide what we wanted to accomplish with the project. From this, we found epics, large bodies of work related to multiple user stories to guide our work forward. We then split them into smaller bodies of work and used a kanban board to assign and keep track of the progress on different tasks. For collaboration we used the agile methods, with daily meetings and sprints (in this case, worksheets). For a seamless version control, we utilize git (github). Each member should work on their own branch with their assigned subtask, and then merge to the main branch. Ideally, this should be done in person with the other members, in case of merge conflict. Notice however, that towards the end of the project we got a more messy git log due to multiple people working and catching errors in the same places. We knew that this might happen, so towards the end it was important to gather to work together on these issues.

Requirements Register

To guide the process it is a good idea to identify some personas and user stories. This allows for a clear picture of what the application must do, should do and could do.

Personas

- **Researcher Ole**: A computational physicist who wants to run large-scale simulations and analyze system properties.
- **Student Geir**: A Master's student who is learning how molecular simulations work and needs an intuitive interface.
- **Developer Arkadi**: A developer maintaining and extending the simulation codebase.

User stories

ID	Persona	User Story	MoSCoW	Implemented?
US1	Researcher Ole	Wants to run simulations with varying densities to compare molecular behavior.	Must	Yes, can be done through JSON file
US2	Student Geir	Wants to visualize molecule positions to better understand spatial distribution.	Should	Partly
US3	Developer Arkadi	Needs unit tests for molecule initialization to ensure correctness after changes.	Should	Yes
US4	Researcher Ole	Needs to log potential energy during simulation to analyze convergence.	Must	Yes
US5	Researcher Ole	Wants to save XYZ files of the initial, final, and intermediate states for external visualization.	Should	Yes (no visalization, can use JMol)
US6	Developer Arkadi	Implements the Metropolis Monte Carlo algorithm so the system reaches equilibrium.	Must	Yes
US7	Student Geir	Wants access to stats like average energy and surface tension post-equilibration to assess results.	Should	Yes
US8	Developer Arkadi	Uses abstract classes or callable objects where suitable to keep the codebase flexible and modular.	Could	Yes

Table 1: Agile user stories with MoSCoW prioritization

Epics

- **EP1 – Core Simulation Engine:** Implement the foundational components for Monte Carlo simulations, including other important support functions. Related to: US1, US6
- **EP2 – Equilibration and Sampling Control:** Design and implement mechanisms for

managing equilibration and production phases, including reset points and logging strategies. Related to: US4, US7

- **EP3 – Data Analysis and Uncertainty Quantification:** Enable reliable post-simulation analysis through statistical sampling, mean/std computation, and block averaging. Related to: US4, US7
- **EP4 – Output and Visualization:** Ensure that XYZ output files, trajectory snapshots, and relevant physical quantities are written in a format compatible with tools like JMol or VMD. Related to: US2, US5
- **EP5 – Software Architecture and Extensibility:** Maintain a clean and modular structure using object-oriented design, abstract classes, and factory patterns to allow future extensibility. Related to: US3, US8
- **EP6 – Testing and Validation:** Implement comprehensive unit and integration tests to verify correctness and stability of key components such as potential energy, molecule moves, and overall simulation behavior. Related to: US3, US6

User guide

To use the project, you should make a JSON file containing the parameters you want to use. If you do not have a JSON file, the program will have a default file. To run the program you write this in the terminal:

With a JSON file

```
python main.py <JSON file, for example "vle-0.80-reference.json">
```

without JSON file:

```
python main.py
```

2 Implementation

Code and class structure

Folder structure

The project is organized as a python package, which later could be published for others to use. Therefore it follows a general structure. Note that we have tried to make an own file for most classes for clarity.

- **main.py**: The central script to run the simulation. mesh and solver, and organizes the entire process.
- **src/**: Contains the source code with subdirectories for the code. For a larger program there could be multiple subdirectories, but in our case it is just one.
  - **ljts/**: Contains the simulation files

    - `box.py`: Contains the box class
    - `molecule.py`: Contains the molecule class
    - `potential.py`: Contains Potential class with abstract subclasses for different approaches.
    - `distortion.py`: Function for computing distortion and determining surface tension
    - `orchestor.py`: Class for running the simulation
    - **timeseries**: Contains the Uncertainty analysis
    - `box average.py`: calculates the mean and standard deviation and plots the data in a histogram/distribution.
  - **tests/**: Contains unit tests or integration tests that verify the correctness of various modules.
  - **config.py**: A Python script for parsing the configuration files so they can be used in the main.py

ER - diagram

An ER - diagram shows the relations between the different classes and is a good way of getting an overview. Because our was too big it is given in the appendix.

Advanced OOP techniques

We use both abstract methods and a factory for a more modular and extendable program. In our case, this allows for an easy extension to a different way of calculating potential energy, for example harmonic. Refering to the personas, this saves Developer Arkadi from a lot of headaches when extending the program.

Python package

Our program is built as a package, which means that it follows a structured and modular design with the necessary metadata and configuration files. This allows it to be published, installed and reused across different systems or projects.

3 Functionality and validation

Testing and Validation

We have comprehensive tests using pytest for easy use. We have both unit and integration tests to make sure our program works as intended. For the unit tests, we for example have:

- A test to check that potential energy is symmetric
- That the box initialization works as intended
- A check that getters work

But for the higher level tests we have:

- **Integration tests** to check that multiple components work together and work as intended. For example that the MetropolisMC works, since this integrates a lot of the other functions.
- **Acceptance test** to check that the program works as a whole, this is the highest "order" test.

Together this is an important validation that the program gives accurate results, without bugs or bias.

Initial configuration setup and input/output

This functionality is fully implemented. The initial configurations are created randomly and gives output files like `init.xyz` and `final.xyz`. These can be visualized with JMol to ensure that the particles are properly distributed and simulation volume is correct.

Pair potential and Monte Carlo method

Both the pair potential and Monte Carlo method is implemented with the Lennard-Jones truncated-shifted (LJTS) potential. We verify the correctness through energy comparisons and symmetry tests. The Metropolis Monte Carlo method integrates this potential and updates configurations. We also monitor the acceptance ratio, which stabilizes within expected ranges. This is a good indication of proper implementation and a good balance.

Test area method and surface tension calculation

We implemented the test area method to estimate surface tension γ by applying small, volume-conserving distortions to the simulation box. Two deformations are used:

- **Area increase:** Scale x and z by $\sqrt{\zeta}$, y by $1/\zeta$.
- **Area decrease:** Inverse scaling.

The potential energy change ΔU under each distortion is used to compute:

$$\gamma = -\frac{T \cdot \log\langle \exp(-\Delta U/T) \rangle}{\Delta A}$$

These values are updated at regular intervals during the simulation. Final surface tension estimates for both distortions are logged and show consistent, physically reasonable results in line with literature values.

Sampling and Data analysis

System Equilibration and Production

We don't have hard coded number of equilibrium steps, instead, we parse "reset sampling at" from the JSON file. This splits the simulation into an equilibration phase where the system relaxes to a stable physical state, and a production phase, where we have production samples. The "reset

sampling at" resets all statistical averages for the surface tension estimation after a given number of steps.

In the reference JSON file it uses first 40 steps to remove the most unsteady first samples, then finds the equilibrium from the first 10 000 steps. After this we reset at 20 000 for a fresh production phase. This makes sure that only statistically meaningful samples are included in the final results. This approach also follows best practices in Monte Carlo simulations and improves reliabiloty.

Uncertainty analysis

To quantify the uncertainty of our simulation results, we calculate both the mean and standard deviation during the production phase. These values reflect the central tendency and variability of the data.

Output and visualization

We generate a file logging the parameters and the values for epot etc for each "log interval" steps. From this we calculate the uncertainty analysis, but also plot the development of the different values.

Results

Vapor only simulation

For the Vapour only simulation we get a potential energy of -13,9 and a surface tension of -0.0055. A low gamma makes sense when there is not boundary betweem them.

Liquid-Vapor simulation

For the Liquid-Vapour simulation we get a potential energy of -915 and a surface tension of 0,56.

Comparison with literature

Even though our result is not exactly like any of the literature provided to us, they are in the correct area. The literature seems to identify an area, but not a specific value, ca. 0.35 - 0.80 depending on the setup for their simulation. For example their temperatures, densities, etc. In Simon Stephan's [2] article they achieve a gamma around 0.40, but in Gloor's [1] article they get a reduced surface tension of 0.615 For more accurate results, we should make a larger system (both in size and number of molecules). To do this effectively we need a low level language, like c++ and ideally a parallelization library like OpenMP. Then we would be more confident in our results.

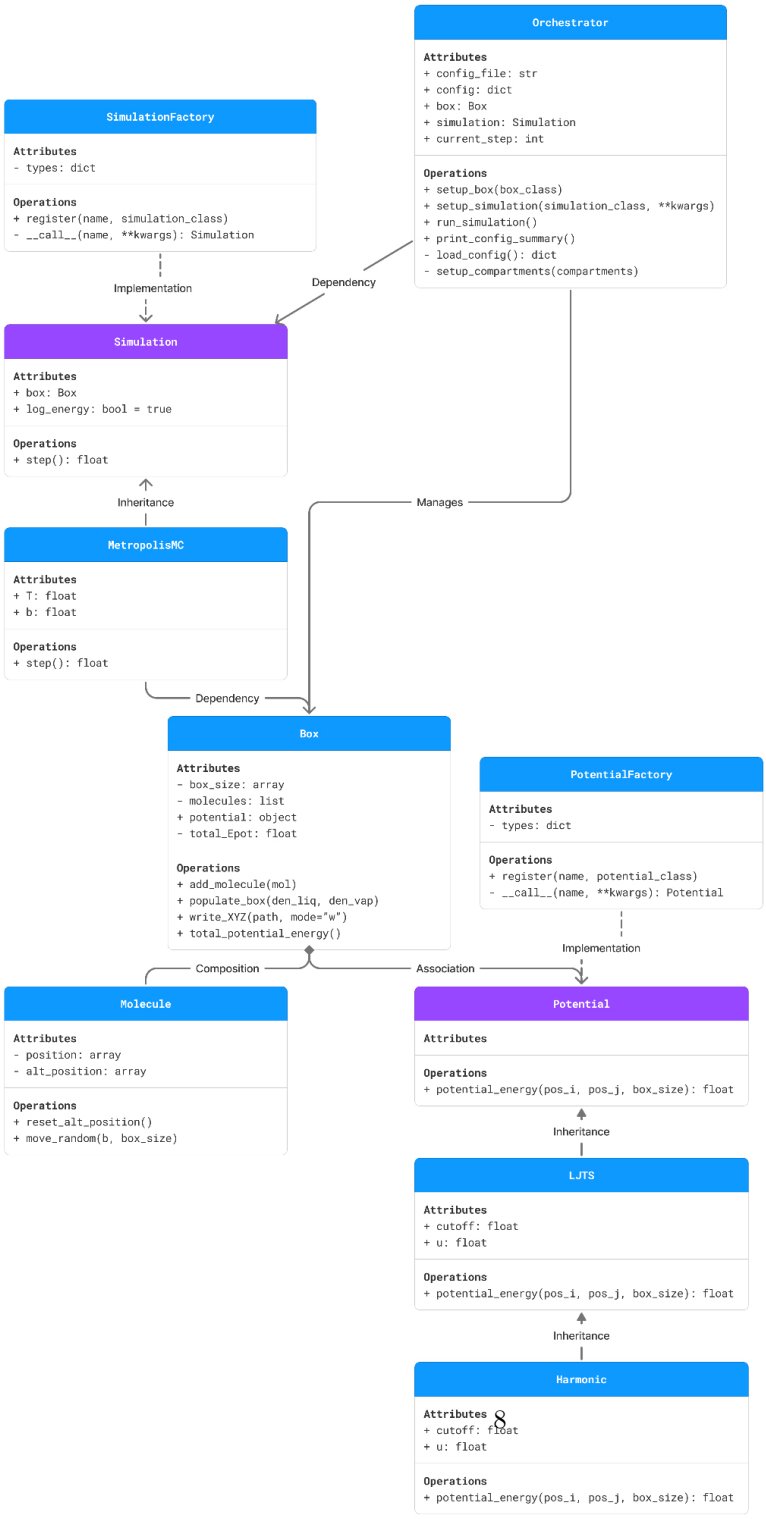
References

[1] Felipe J. Bias Enrique de Miguel Guy J. Gloor, George Jackson. Test-area simulation method for the direct determination of the interfacial tension of systems with continuous or discontinuous potentials. *The Journal of Physical chemistry*, 2005.

[2] Walter G. Chapman Simon Stephan Jinlu Liu, Kai Langenbach and Hans Hasse. Vaporliquid interface of the lennard-jones truncated and shifted fluid: Comparison of molecular simulation, density gradient theory, and density functional theory. *The Journal of Physical chemistry*, 2018.

[3] University of Hawaii at Mānoa. Elusive atmospheric molecule methanediol discovered, 2021. Accessed: 2025-06-18.

4 Appendix


Orchestrator
Attributes
+ config_file: str
+ config: dict
+ box: Box
+ simulation: Simulation
+ current_step: int
Operations
+ setup_box(box_class)
+ setup_simulation(simulation_class, **kwargs)
+ run_simulation()
+ print_config_summary()
- load_config(): dict
- setup_compartments(compartments)
SimulationFactory
Attributes
- types: dict
Operations
+ register(name, simulation_class)
- __call__(name, **kwargs): Simulation
Implementation
Dependency
Simulation
Attributes
+ box: Box
+ log_energy: bool = true
Operations
+ step(): float
Inheritance
Manages
MetropolisMC
Attributes
+ T: float
+ b: float
Operations
+ step(): float
Dependency
Box
Attributes
- box_size: array
- molecules: list
+ potential: object
- total_Epot: float
Operations
+ add_molecule(mol)
+ populate_box(den_liq, den_vap)
+ write_XYZ(path, mode="w")
+ total_potential_energy()
PotentialFactory
Attributes
- types: dict
Operations
+ register(name, potential_class)
- __call__(name, **kwargs): Potential
Implementation
Composition
Association
Molecule
Attributes
- position: array
- alt_position: array
Operations
+ reset_alt_position()
+ move_random(b, box_size)
Potential
Attributes
Operations
+ potential_energy(pos_i, pos_j, box_size): float
Inheritance
LJTS
Attributes
+ cutoff: float
+ u: float
Operations
+ potential_energy(pos_i, pos_j, box_size): float
Inheritance
Harmonic
Attributes
+ cutoff: float
+ u: float
Operations
+ potential_energy(pos_i, pos_j, box_size): float