UNIVERSITY OF HELSINKI

Faculty of Science
Programming Projects in Molecular Modelling
Prof. Alex Bunker

# Project 3

# Energy Minimization and the Linked Cell Algorithm

Iris Tamsalu
Ruoyu Tang
Luca Mössinger

March 2025

# Contents

# 1 Introduction

This project focuses on simulating the behavior of particles interacting via the Lennard-Jones potential. The molecular dynamics simulation includes two main functionalities:

1. Simulation of Particles: The system of particles is modeled under the influence of the Lennard-Jones potential using the Velocity Verlet algorithm. The simulation supports both periodic boundary conditions (PBC) and hard-walled boundary conditions.

2. Energy Minimization: This functionality optimizes the configuration of particles to minimize the potential energy of the system by adjusting the particle positions. The minimization is performed using the steepest descent method.

Additionally, the project introduces the Linked-Cell Algorithm (LCA) to enhance computational efficiency when calculating particle interactions in large systems. By narrowing down the search for interacting particle pairs to nearby subregions of the simulation box, LCA enables the simulation of much larger systems in both 2D and 3D.

The project begins with a 2D simulation and gradually moves to 3D, observing how the system evolves and the computational time scales as the number of particles increases. The simulation output, like particle trajectories, energy evaluation, and computational times are visualized.

The code and animations are provided in the supplementary material, available at the following GitHub repository: https://github.com/iristamsalu/KEM381_project3. All the usage instructions are provided in the README file.

# 2 Lennard-Jones Interaction in 2D and 3D

We built the new program on the code from Project 2 (`GitHub` link to the previous project) but also added several improvements for stability and code extensibility. Some of them were taken from `Alexplot` [2]. Most notably, the simulation was extended from 2D to 3D, and the code was rewritten in a modular, object-oriented structure.

Particle positions are now initialized on a lattice with small random noise. Velocities are assigned from a uniform distribution [-0.01, 0.01) and also scaled to match a target temperature while ensuring zero net momentum.

Periodic boundary conditions and force calculations using the Lennard-Jones potential were extended to 3D, and energy tracking was improved through dedicated functions. In addition to energy tracking, the program tracks computational time and, of course, particle trajectories for visualization.

The mentioned updates made it possible to run the simulation with parameters specified on the slide: $\sigma = \varepsilon = 1$, cutoff at $2.5\sigma$, timestep of 0.0001, and target density 0.8. These were used for all the computational experiments in this project report.

Relevant programs for tasks 1-3 in the `src/` directory on `GitHub` are:

- `main.py`
- `simulation.py`
- `forces.py`
- `config.py`
- `output_and_plots.py`

## 2.1 Deepest Descent Energy Minimization

The new program includes an energy minimization option, which is performed by the `minimize_energy()` function in the `simulation.py` module. To analyze the stable configuration of the particle system, the periodic boundary conditions are removed, allowing the system to evolve without artificial wrapping effects. Instead, hard-wall boundaries are used to avoid particles leaving the box.

The positions are then optimized using the steepest descent energy minimization method. In this approach, the kinetic energy is set to zero, and particles move along the direction of the negative gradient of the potential energy.

There are two ways to run minimization. One is to start from a lattice structure. The other option is to first run the Lennard-Jones MD simulation and then, at a certain timestep, start minimization from random positions.

The interaction between two particles is governed by the Lennard-Jones potential, which describes the balance between attractive and repulsive forces [2]:

$$U(r) = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

where:

- $r$ is the distance between particles $i$ and $j$,
- $\varepsilon$ determines the depth of the potential well,
- $\sigma$ is the finite distance at which the inter-particle potential is zero.

The force acting on particle $i$ is given by the negative gradient of the potential energy with respect to its position [2]:

$$\vec{F}_i = -\nabla_{\vec{r}_i} U \tag{2}$$

This force drives the particle in the direction of the steepest descent on the energy landscape.

The Steepest Descent method is used to iteratively update the positions of particles to minimize the total potential energy. The position update rule is given by [3]:

$$\vec{r}_{n+1}^{(N)} = \vec{r}_n^{(N)} + \frac{\vec{D}^{(N)}}{\max\left(\left|\vec{D}_{il}\right|\right)} \cdot \Delta t \tag{3}$$

where:

- $\vec{r}_n^{(N)}$ is the position of particle $N$ at iteration $n$,
- $\vec{r}_{n+1}^{(N)}$ is the updated position at iteration $n+1$,
- $\vec{D}^{(N)}$ is the descent direction for particle $N$,
- $\max\left(\left|\vec{D}_{il}\right|\right)$ is the maximum absolute value among all components $l \in \{x, y, z\}$ of the descent vectors $\vec{D}_i$ across all particles $i$,
- $\Delta t$ is the step size.

The energy minimization process is considered converged when the change in total potential energy between two consecutive time steps is smaller than a specified threshold $\epsilon$:

$$\Delta U = |U_{n+1} - U_n| < \epsilon \tag{4}$$

In this project, $\epsilon$ was set to $10^{-6}$, as this value ensured that no significant changes in the particle configuration were observed in the `OVITO` visualizations. The value of $\epsilon$ might vary depending on the system and how precise the final configuration has to be.

## 2.2 Results of Deepest Descent Energy Minimization

The total, kinetic, and potential energy evaluation during minimization can be seen in the following plots provided for both 2D and 3D systems (see figures 1 and 4). The command lines with system parameters are provided in the figure titles. The first 7000 time steps were spent performing LJ simulation, and the next 10,000 steps were used to get potential energy as low as possible using the deepest descent minimization method. During minimization, particle velocities and, therefore, kinetic energy are equal to zero. Therefore, kinetic energy drops to zero when the minimization starts, and total energy is equal to the configuration potential energy.

As expected, the potential energy and, therefore, total energy decrease during the minimization and eventually stabilize as the system settles into a local or global minimum. Figures 2-3 and 5-6 are `OVITO` snapshots of simulation at different stages of the simulation. After minimization, the particles are more structured, and the distances between particles are similar throughout the configuration. However, there are holes in the final 2D configuration that indicate that the configuration is at a local energy minimum, not a global minimum. For comparison, the minimization simulation that started with a lattice structure (figure 3) resulted in an almost perfectly uniform and compact structure. For 3D, the configurations after minimization looked quite similar to the naked eye.

The `.mp4` files generated from minimization `.xyz` files are uploaded to the `videos/` on `GitHub`. For each dimension, there are two videos: minimization from the lattice and minimization from random positions:

- `2D_minimization.mp4` - Lennard-Jones PBC simulation followed by minimization
- `3D_minimization.mp4` - Lennard-Jones PBC simulation followed by minimization
- `2D_minimization_only.mp4` - only minimization from the initial lattice
- `3D_minimization_only.mp4` - only minimization from the initial lattice

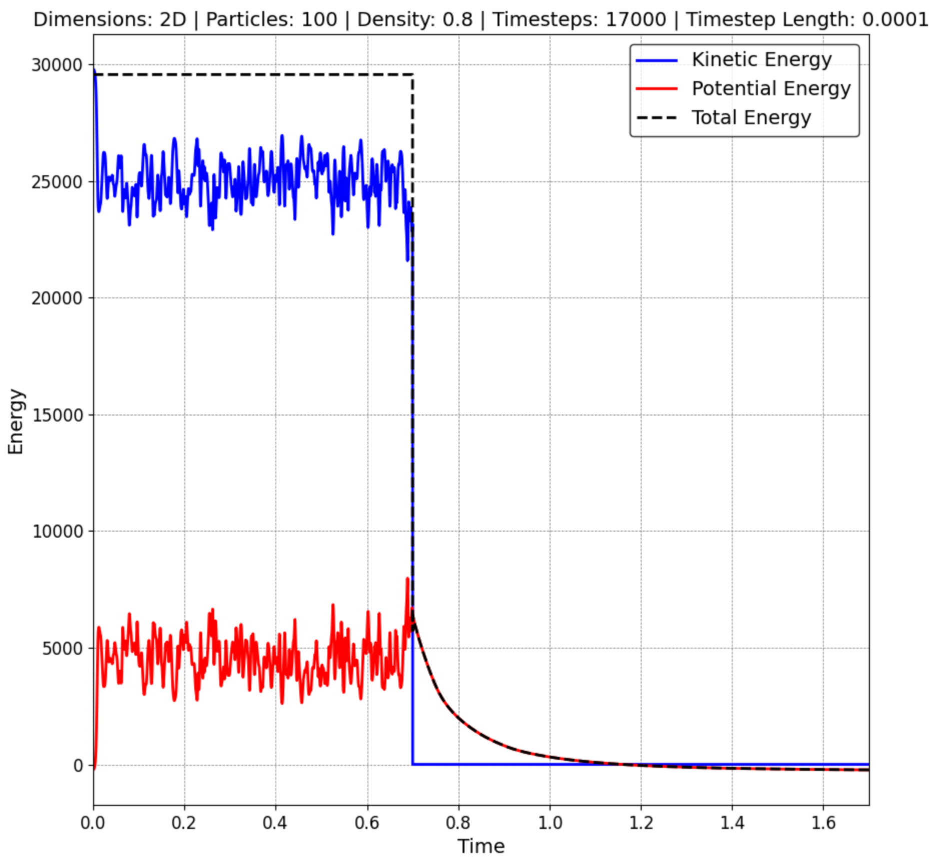

Figure 1: Potential, kinetic, and total energy during Lennard-Jones simulation with PBC followed by deepest descent energy minimization in 2D.
Command line: `python3 main.py --dimensions 2 --steps 7000 --dt 0.0001 --density 0.8 --n_particles 100 --use_pbc --minimize --minimization_steps 10000`

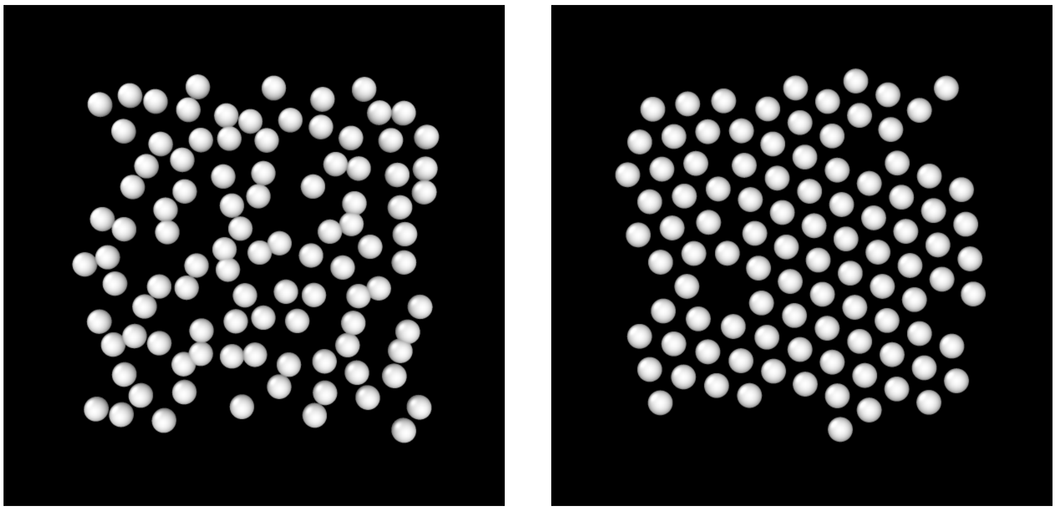

Figure 2: Configuration before and after running minimization. Minimization started after running the Lennard-Jones simulation with PBC for a while. The minimization had not converged after 100000 steps.
Command line: `python3 main.py --dimensions 2 --steps 7000 --dt 0.0001 --density 0.8 --n_particles 100 --use_pbc --minimize --minimization_steps 100000`.

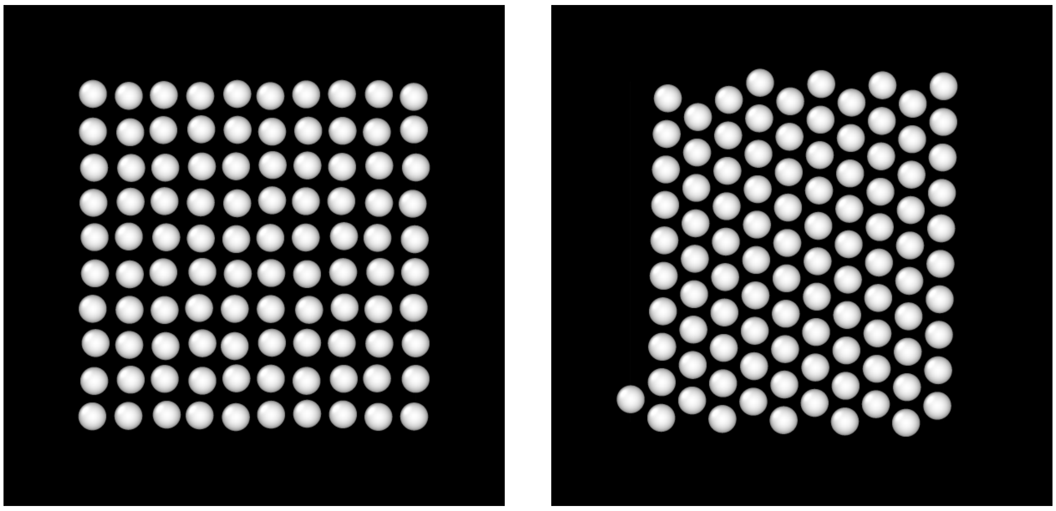

Figure 3: Configuration before and after minimization. Minimization started with a uniform 2D lattice and converged after 22911 steps.
Command line: `python3 main.py --dimensions 2 --steps 7000 --dt 0.0001 --density 0.8 --n_particles 100 --minimize_only --minimization_steps 100000`

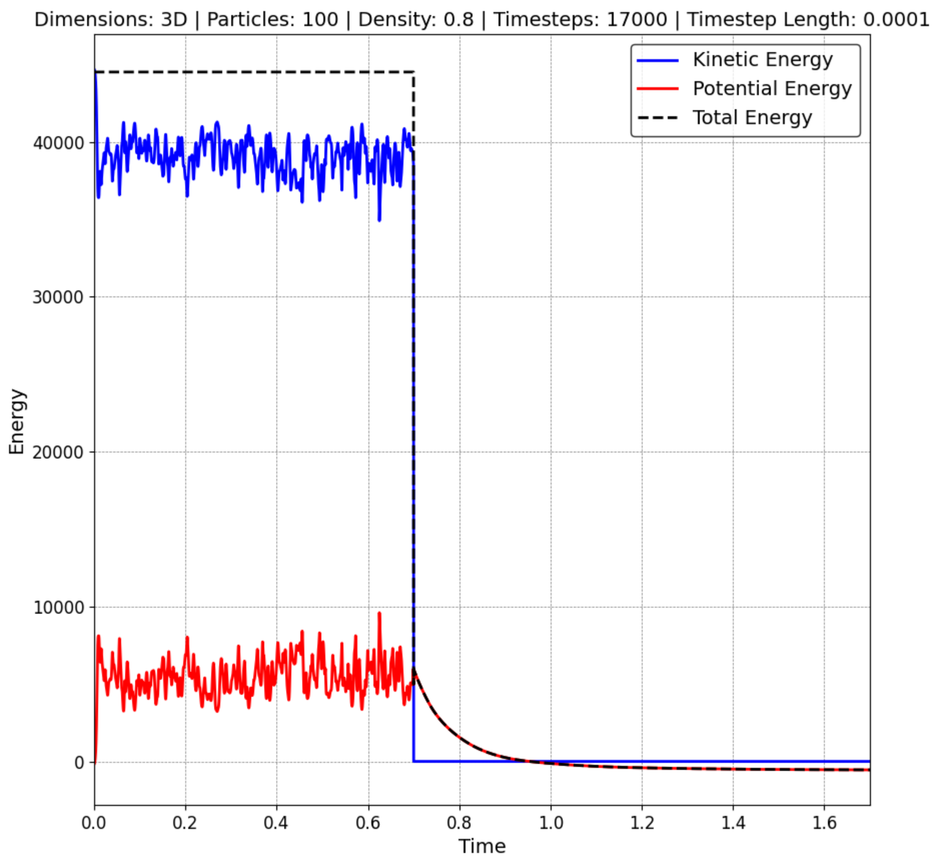

Figure 4: Potential, kinetic, and total energy during Lennard-Jones simulation with PBC followed by deepest descent energy minimization in 3D.
Command line: `python3 main.py --dimensions 3 --steps 7000 --dt 0.0001 --density 0.8 --n_particles 100 --use_pbc --minimize --minimization_steps 10000`

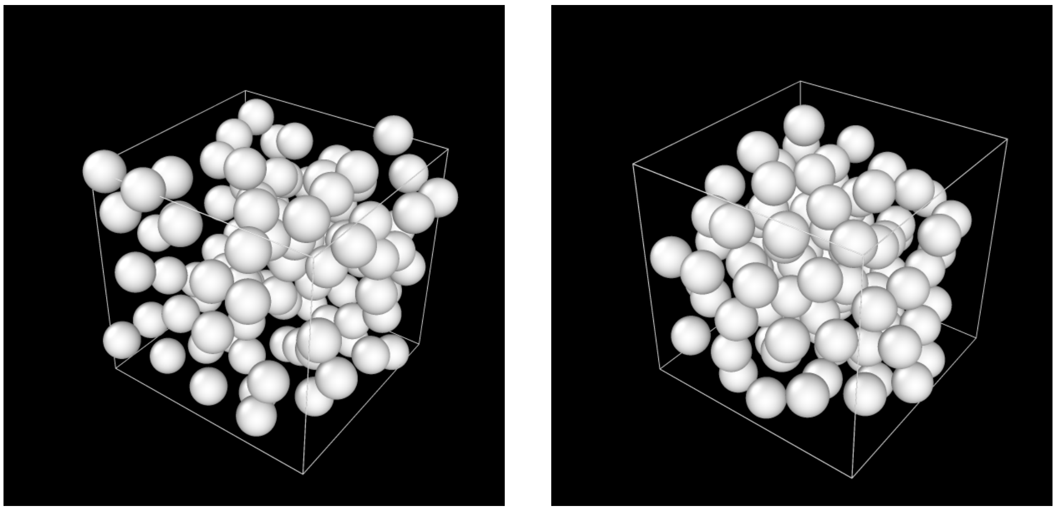
Figure 5: Configuration before and after minimization. Minimization started from random positions after performing Lennard-Jones with PBC first.
Command line: `python3 main.py --dimensions 3 --steps 7000 --dt 0.0001 --density 0.8 --n_particles 100 --use_pbc --minimize --minimization_steps 100000`.

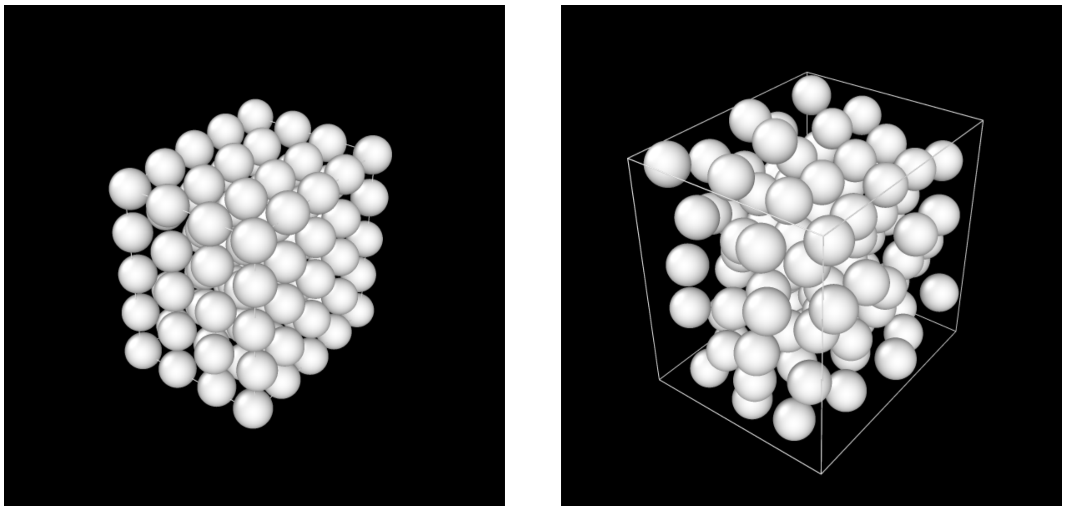
Figure 6: Configuration before and after minimization. Minimization started with a uniform lattice.
Command line: `python3 main.py --dimensions 3 --dt 0.0001 --density 0.8 --n_particles 100 --minimize_only --minimization_steps 100000`

## 2.3 Computational Time

The required computation time is tracked for different system sizes (see figure 7). It can be observed that the computation time increases as the number of particles increases, and of course, for the 3D simulation, the computational times are longer. The 3D simulation was also run with 600 particles and in that case, the run time exceeded one hour. We did not detect instabilities, and total energy was conserved throughout the simulation. Therefore, if we had more time, perhaps slightly more particles would also be possible.

Almost identical commands were given to the program in both dimensions. Only `--dimensions` and `--n_particles` values were changed. The commands used for collecting computational time data in 3D:

```
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8
--n_particles 10 --use_pbc
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8
--n_particles 50 --use_pbc
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8
--n_particles 100 --use_pbc
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8
--n_particles 200 --use_pbc
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8
--n_particles 300 --use_pbc
```

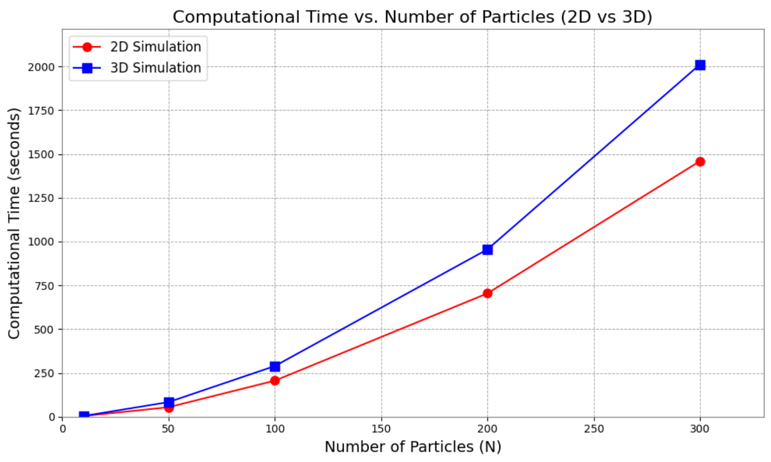

Figure 7: Computational time vs. number of particles in 2D and 3D Lennard-Jones simulation with PBC.

# 3 Linked Cell Algorithm

## 3.1 Overview of the Linked Cell Algorithm

The linked cell algorithm is a method used in molecular dynamics simulations to reduce the computational cost of evaluating pairwise interactions between particles, particularly when dealing with short-range potentials like the Lennard-Jones potential [2]. In a naive approach that we used in Tasks 1 and 2, each particle is compared with every other particle, leading to a computational cost that scales as $\mathcal{O}(N^2)$, where $N$ is the number of particles. The linked cell algorithm improves this by reducing the number of necessary comparisons and bringing the scaling down to approximately $\mathcal{O}(N)$.

In LCA the simulation box is divided into smaller subcells, each with a linear dimension at least as large as the Lennard-Jones interaction cutoff radius $r_c$. Particles are assigned to cells based on their positions, as interactions only occur within the cutoff radius. In 3D, this results in a cube of 27 cells: the particle's own cell and its 26 immediate neighbors. In 2D, the same principle applies, forming a grid around each particle. This spatial decomposition accelerates simulations for large systems.

The algorithm starts with an initialization, where the simulation box is divided into uniform subcells [4]. A list of relevant neighboring cell pairs is generated, and the corresponding displacement vectors are stored. Each particle is assigned to a specific subcell and added to the respective linked list. After initialization, the simulation moves to the force calculation phase, where, for each particle, neighbors within the same cell and adjacent cells are checked. If the distance between two particles is within the cutoff radius $r_c$, their interaction is computed. To ensure efficiency and correctness, each particle pair is considered only once, avoiding double-counting of forces.

In our program functions that perform LCA are located in the module `forces.py`: `build_linked_cells()` and `compute_force_lca()`.

## 3.2 Computational Time with Linked Cells

The required computation time using the linked cell algorithm was evaluated for both the 2D and 3D systems across different system sizes, in order to compare the efficiency to the previous naive method.

As can be seen from figures 8 and 9, for a small number of particles LCA is not necessarily faster. For 2D simulations, the LCA started to be more efficient as the number of particles was around 50. For 3D simulations, the number of particles had to be significantly bigger, and LCA became more efficient only for the 600-particle simulation. That is because of the extra time needed to create linked cells.

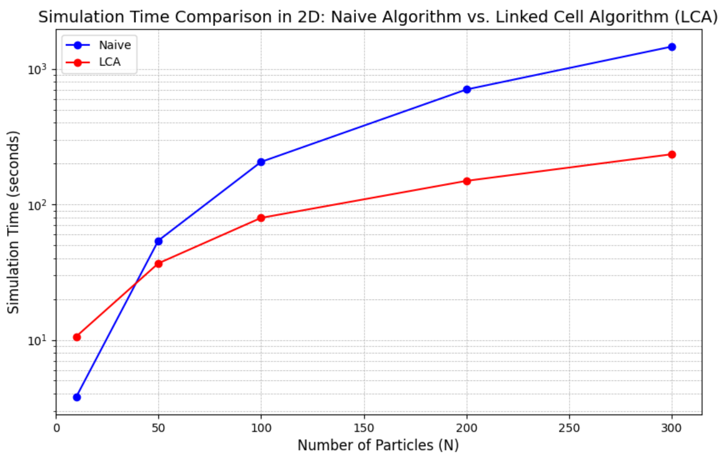

Figure 8: Computational time with LCA and with the naive algorithm for different system sizes in 2D. The LCA was faster for systems with 50 and more particles. The command line structure for LCA runs:

```
python3 main.py --dimensions 2 --steps 10000 --dt 0.0001
--density 0.8 --n particles <nr of particles> --use_pbc --use_lca
```

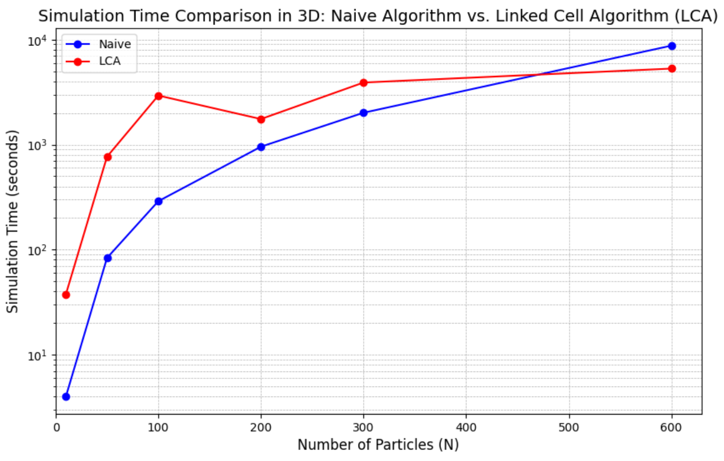

Figure 9: Computational time with LCA and with the naive algorithm for different system sizes. The LCA becomes significantly faster for systems with more than 400-500 particles. An interesting observation was that running LCA with 100 particles was slower than with 200 particles. The command line structure for LCA runs:

```
python3 main.py --dimensions 3 --steps 10000 --dt 0.0001
--density 0.8 --n particles <nr of particles> --use_pbc --use_lca
```

As shown in Figure 9, the performance of LCA with 100 particles is worse than with 200 particles. We reason that it might happen for the following reasons:

In `build_linked_cells()`, the number of cells ($lc$) is set as:

$$lc = \max(1, \lfloor \text{box\_size}/r_{\text{cutoff}} \rfloor) \tag{5}$$

Even with fewer particles, the domain is split into many cells, most of which are empty. LCA sets up data structures like the `head` array and the linked list `lscl`, which have a fixed setup cost. At $N = 100$, this overhead is large compared to the total run-time, making LCA less efficient. At $N = 200$, more particles fill the cells, so LCA works better.

LCA speeds up force calculations by checking only nearby cells instead of using an $O(N^2)$ approach. However, when $N = 100$, many cells are empty, but the algorithm still loops over them. The `head` array is checked for each cell, and most entries are $-1$, meaning the cell is empty. Empty neighbor cells are processed, leading to extra computations. At $N = 200$, cells have more particles, so these checks are more useful and waste less time.

In `compute_forces_lca()`, each particle loops over 9 (in 2D) or 27 (in 3D) neighboring cells using: `for offset in neighbor_offsets`. When $N = 100$, most neighboring cells are empty. The algorithm still checks boundaries, applies periodic boundary conditions, and looks up the `head` array. These extra steps slow down the computation. At $N = 200$, cells have more particles, so fewer empty cells are checked, making the algorithm faster.

# 4 JIT Compiler for Faster Computing

Although the LCA keeps the computational time increase linear, simulations with large systems can still take a significant amount of time. For example, systems with more than 600 particles can take hours to run. To address this, we implemented just-in-time compilation (JIT) using the `Numba` package in `Python` [1].

JIT compilation allows functions to be converted into optimized machine code at runtime, drastically reducing execution time. In our implementation, we applied the `@jit` decorator from `Numba` to optimize key functions such as force calculations. This optimization led to significant speed improvements in both the naive force update algorithm and the LCA. The functions with `@jit` are located in the module `forces_jit.py`.

The improvements in computational speed are shown in Figures 10 and 11, where we compare different methods used in this project. The initial naive method with $\mathcal{O}(N^2)$ has significantly longer run times. The LCA algorithm brings the computational cost down to $\mathcal{O}(N)$ and makes it possible to run with a couple of hundred particles in a reasonable time. Implementing JIT reduced computational time significantly for both methods, and LCA with JIT is, as expected, the fastest. An interesting observation that simulating 200 particles with LCA took less time than 100 particles, occurred also now with JIT. As was previously shown in figures 8 and 9, also here for 3D the number of particles has to be 400-600 for LCA to be faster than naive. For 2D, LCA is better already for small systems.

It is difficult to determine the maximum system size since the program can run for extended periods, but time constraints in this project limit such runs. To demonstrate that we attempted larger system sizes, we present a 3D Lennard-Jones simulation with PBC and 3000 particles (figure 12). Throughout the simulation, the energies were monitored, and the plot shows good total energy conservation. The simulation took 1632 seconds to complete, and further run details are provided in the figure caption.

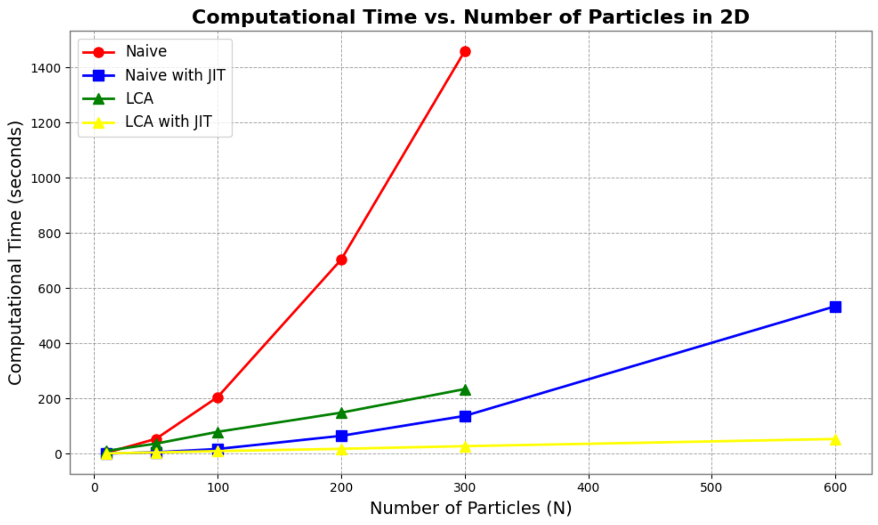

Figure 10: Computational times with the naive method, LCA, JIT with naive, and JIT with LCA for different system sizes in 2D. The command line structure for JIT runs (`--use_lca` was used only for LCA runs) is as follows: `python3 main.py --dimensions 2 --steps 10000 --dt 0.0001 --density 0.8 --n particles <nr of particles> --use_pbc --use_lca --use_jit` .

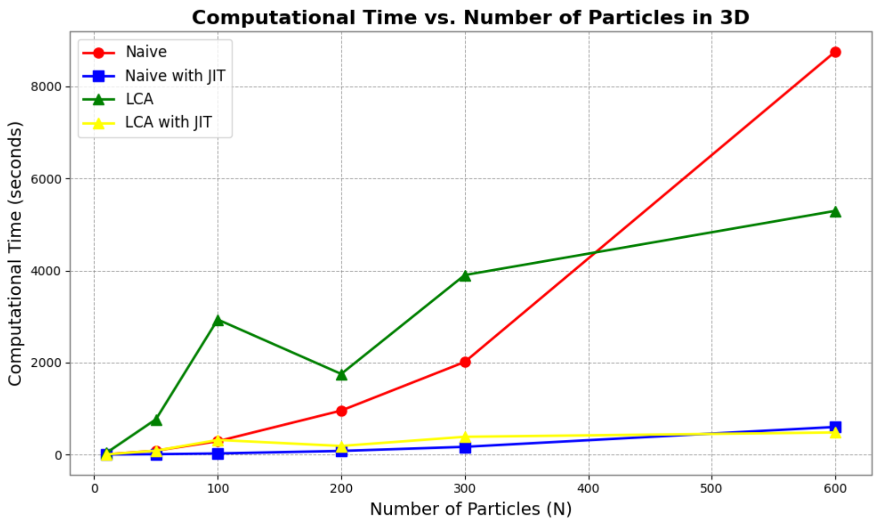

Figure 11: Computational times with the naive method, LCA, JIT with naive, and JIT with LCA for different system sizes in 3D. The command line structure for JIT runs (`--use_lca` was used only for LCA runs) is as follows: `python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8 --n particles <nr of particles> --use_pbc --use_lca --use_jit` .

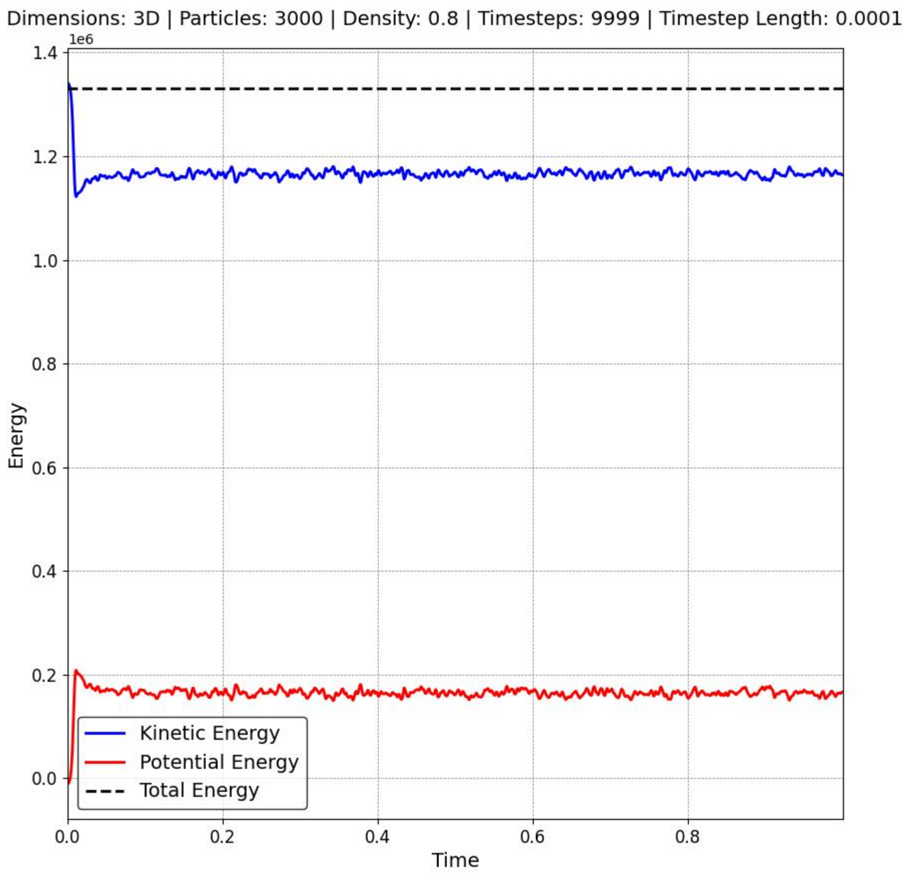

Figure 12: Lennard-Jones simulation with PBC in 3D with 3000 particle. Both LCA and JIT were used. The computational time was 1632 s.
Command line: `python3 main.py --dimensions 3 --steps 10000 --dt 0.0001 --density 0.8 --n_particles 3000 --use_pbc --use_lca --use_jit`

# 5 Conclusion

This project focused on simulating the behavior of particles interacting via the Lennard-Jones potential, with an emphasis on minimizing potential energy and improving computational time. The project implemented both 2D and 3D simulations to observe system behavior and analyze how computational time scales with increasing particle numbers.

The project introduced an energy minimization function into the program using the steepest descent method, which allows for the optimization of particle configurations by reducing the system's potential energy. The minimization was performed either from a random initial state or from a structured lattice configuration. The minimization process successfully moved the system toward energy minima, with the resulting configurations showing clearer structural patterns and more uniform distances between particles.

To address the challenge of scaling the simulation to larger systems, we introduced the LCA, which significantly improved the efficiency of particle interaction calculations. The naive $\mathcal{O}(N^2)$ method was inefficient for large systems, whereas the LCA reduced the computational time to $\mathcal{O}(N)$, allowing for much larger systems to be simulated. We observed that LCA showed its effectiveness after a certain threshold in both 2D (around 50 particles) and 3D (after 400-00 particles), where the overhead of building the cell structure becomes manageable.

In addition to optimizing the simulation, JIT compilation with `Numba` further accelerated the computation by compiling critical functions into machine code, leading to significant performance improvements. This made it possible to simulate larger systems in a reasonable amount of time.

## References

[1] Numba Development Team. Numba: A just-in-time compiler for python, 2025. Accessed: 2025-03-25.

[2] Alex Bunker. Molecular modelling. *Lecture at University of Helsinki (KEM 342)*, 2025.

[3] Alex Bunker. Programming projects in molecular modelling. *Lecture at University of Helsinki (KEM 381)*, 2025.

[4] ETH Zurich. Computational statistical physics – exercise 8, 2025. Accessed: 2025-03-25.