Lamina - A 2D Molecular Dynamics Simulator

Harish Charan, Durham University, Durham, UK[*†]

Description

Lamina is a modular and extensible molecular dynamics (MD) simulation package written in C, designed to model a wide variety of soft and condensed matter systems. It supports time evolution using robust integrators and a range of thermostats, with accurate force evaluations for bonded and non-bonded interactions. Originally built for 2D simulations of bonded systems, **Lamina** has grown to support broader research goals including active matter, granular solids, and complex fluids.

Why "Lamina"?

The word *Lamina* comes from Latin, meaning "a thin layer", "a plate", or "a sheet". In nature and science, laminae often refer to structural elements that are flat and extended in two dimensions, such as leaves, thin metal sheets, or tissue membranes.

This name reflects both the **two-dimensional** (2D) nature of the simulations and the types of materials **Lamina** is built to study: **liquids**, **soft solids**, and **networked structures** confined to thin sheets or layers. Just as natural laminae exhibit rich structural and dynamic behaviors in a seemingly simple geometry, this code is designed to explore the complexity of emergent phenomena in 2D materials and soft matter systems.

[*]Email: charan.harish@gmail.com, harish.charan@durham.ac.uk
[†]August 12, 2025

Input File Format

The input file must follow a strict format to ensure proper parsing and simulation setup. It begins with a header section that defines global parameters:

- `timeNow`: Current simulation time.
- `nAtom`, `nBond`: Number of atoms and bonds in the system.
- `nAtomType`, `nBondType`: Number of atom and bond types.
- `region[1]`, `region[2]`: Dimensions of the simulation region in the x and y directions.

This is followed by an `Atoms` section, which begins with a comment line indicating the data columns. Each subsequent line describes one atom with the format:

```
atomID molID atomType atomRadius rx ry vx vy
```

Then comes the `Bonds` section, also starting with a comment line indicating column labels. Each bond is specified as:

```
BondID BondType atom1ID atom2ID kb ro
```

The input file should contain no extra lines or formatting inconsistencies, as the parser expects this exact structure.

Example Input File

```
timeNow 0
nAtom 4
nBond 2
nAtomType 2
nBondType 1
region[1] 20
region[2] 10
Atoms #atomID molID atomType atomRadius rx ry vx vy
1 1 2 0.0 -1.0 0.0 -1.0 0.0
2 1 2 0.0  1.0 0.0 0.0 0.0
3 2 3 0.6  3.0 0.0 0.0 0.0
4 2 3 0.5  4.0 0.0 0.0 0.0
Bonds #BondID BondType atom1ID atom2ID kb ro
1 1 1 2 1.00 2.1
2 1 2 3 1.00 2.1
```

Key Features

Interaction Potentials

- **Yukawa potential**: (Screened Coulomb interactions).
  The Yukawa interaction potential used in the simulation is defined as:

$$V(r) = A \cdot \frac{1}{r} \exp\left(-\kappa r\right),$$

  where A is the interaction strength and κ is the screening parameter. Both parameters are user-defined and must be specified in the input file. This form captures screened interactions, making it suitable for modeling effective forces in systems such as colloids, dusty plasmas, and other soft matter systems with short-range repulsion.

- **Lennard-Jones potential**: Standard 12-6 potential for atomic interactions The Lennard-Jones (LJ) potential is also supported in the simulation and is commonly used to model van der Waals interactions between neutral atoms or molecules. It is given by the standard 12-6 form:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right],$$

  where ϵ determines the depth of the potential well (interaction strength), and σ is the finite distance at which the inter-particle potential is zero. Both ϵ and σ must be specified in the input file. The LJ potential captures both the short-range repulsion and long-range attraction typical of atomic interactions.

- **Harmonic bond potential**: For elastic network models The simulation uses a harmonic bond potential to model interactions between bonded particles, typically used in elastic network models. The potential is defined as:

$$V(r) = \frac{1}{2} k_b (r - r_0)^2,$$

  where k_b is the bond stiffness and r_0 is the equilibrium bond length. Both parameters must be specified in the input file for each bond.

Additionally, the behavior of the force calculation can be modified using the `normFlag` parameter. If `normFlag` is set to 1 (enabled), the bond interaction is normalized by the equilibrium bond length r_0, and the potential takes the form:

$$V(r) = \frac{1}{2} k_b r_0 \left(\frac{r - r_0}{r_0} \right)^2 .$$

This normalization effectively scales the interaction strength relative to its rest configuration, which is particularly useful for modeling systems where relative deformation plays a more critical role than absolute displacement. If `normFlag` is set to 0 (disabled), the unnormalized standard form is used.

- **Hookean granular contact potential**: For simulating soft granular matter The simulation supports a Hookean granular contact potential, widely used to model repulsive interactions in soft granular matter. The potential is given by:

$$V(r) = \frac{1}{2} k_n \delta^2,$$

where k_n is the contact stiffness and $\delta = R_i + R_j - r$ is the overlap between two interacting discs with radii R_i and R_j, separated by a distance r. The potential is active only when $\delta > 0$, i.e., when the particles are in contact.

If the normalization flag is enabled (`normFlag` $= 1$), the interaction is scaled by the sum of the radii $R_{ij} = R_i + R_j$, and the potential becomes:

$$V(r) = \frac{1}{2} k_n R_{ij} \left(\frac{\delta}{R_{ij}} \right)^2 .$$

This normalization is useful when simulating polydisperse systems, ensuring that the interaction strength appropriately accounts for particle size and maintains consistent physical behavior across different contact pairs.

Time Integration

- **Leap-Frog integrator** The Leap-Frog integrator is a cornerstone of Molecular Dynamics simulations, renowned for its efficiency and good energy conservation properties. It is a second-order accurate, explicit, and symplectic algorithm used to numerically solve Newton's equations of motion. The method's name arises from the way it updates positions and velocities at interleaved time steps. This staggering ensures that velocities at time $t + \frac{\delta t}{2}$ depend on positions at time t, and positions at time $t + \delta t$ depend on velocities at time $t + \frac{\delta t}{2}$, effectively "leaping" over each other in time. A key advantage of the Leap-Frog integrator is that it requires only one force evaluation per time step, contributing to its computational efficiency while maintaining reasonable accuracy for many simulation scenarios. [cite: 26]

  Equations of the Leap-Frog Integrator

  The standard formulation of the Leap-Frog integrator involves the following update equations:

$$v_i \left(t + \frac{\delta t}{2} \right) = v_i(t) + \frac{\delta t}{2} \frac{F_i(t)}{m_i} \tag{1}$$

$$r_i(t + \delta t) = r_i(t) + v_i \left(t + \frac{\delta t}{2} \right) \delta t \tag{2}$$

$$v_i(t + \delta t) = v_i \left(t + \frac{\delta t}{2} \right) + \frac{\delta t}{2} \frac{F_i(t + \delta t)}{m_i} \tag{3}$$

  where:

  - $r_i(t)$ is the position of particle i at time t.
  - $v_i(t)$ is the velocity of particle i at time t.
  - $F_i(t)$ is the force acting on particle i at time t.
  - m_i is the mass of particle i.
  - δt is the time step.

  The algorithm proceeds by first updating the velocities by half a time step based on the forces at the current time. Then, the positions are updated by a full time step using these half-step velocities. Finally, the velocities are updated by another half time step using the forces at the new time.

- **Velocity-Verlet integrator** We begin by briefly describe the velocity Verlet algorithm, which allows us to integrate the Newton equations of motion, characteristic of the NVE ensemble, namely

$$m_i \frac{d^2 r_i}{dt^2} = -\sum_{j \neq i} \nabla_i U(|r_i - r_j|)$$

In Eq. 1 m_i is the mass of particle i, $r_i = (x_i, y_i, z_i)$ is the position of particle i in 3-dimensional space, $\nabla_i = (\frac{\partial}{\partial x_i}, \frac{\partial}{\partial y_i}, \frac{\partial}{\partial z_i})$, while U denotes the potential energy. The most used algorithm used in practice to integrate Eq. 1 is the velocity Verlet algorithm, whose steps are implemented as follows:

$$\begin{aligned} r_i(t+\delta t) &= r_i(t) + v_i(t)\delta t + \tfrac{f_i(t)}{2m_i}\delta t^2 \\ v_i(t+\delta t/2) &= v_i(t) + \frac{\delta t}{2}\frac{f_i(t)}{m_i} \\ f_i(t+\delta t) &= f_i(r_i(t+\delta t)) \\ v_i(t+\delta t) &= v_i(t+\delta t/2) + \frac{\delta t}{2}\frac{f_i(t+\delta t)}{m_i} \end{aligned}$$

where r_i, v_i and f_i denote the position of the i-th particle, its velocity, and the force it is subjected to. As there are no dissipative forces, the energy is conserved within this algorithm.

In practice, the following, equivalent, version of the velocity Verlet is probably the most used:

$$\begin{aligned} v_i(t+\delta t/2) &= v_i(t) + \frac{\delta t}{2}\frac{f_i(t)}{m_i} \\ r_i(t+\delta t) &= r_i(t) + v_i(t+\delta t/2)\delta t \\ f_i(t+\delta t) &= f_i(r_i(t+\delta t)) \\ v_i(t+\delta t) &= v_i(t+\delta t/2) + \tfrac{\delta t}{2}\tfrac{f_i(t+\delta t)}{m_i} \end{aligned}$$

As a simple exercise, you should convince yourself that the two versions in Eq. 2 and Eq. 3 are equivalent.

Thermostats and Temperature Control

- **Gaussian thermostat** The choice of thermostats and barostats depends on whether the final states prepared are true ensembles or not. A Gaussian thermostat be used in Lamina. This thermostat is based on the Gauss's principle of least constraint and results in an isokinetic ensemble. In the limit of large number of degrees of freedom a Gaussian thermostat produces the correct canonical ensemble. The Gaussian thermostat can be build into the Leap-Frog integrator. Because the Gaussian thermostat controls the temperature by the constraint method it gets very close to the desired temperature, and the temperature constraint is only preserved to the accuracy of numerical integration. This can be very useful if one is simulating very large systems. We give here a very simple formulation of the Gaussian thermostat for a system of N particles.

$$\frac{1}{2}\sum_{i=1}^{N} \dot{r}_i^2 = N E_k. \tag{4}$$

Where N and E_k are the number of particles and the kinetic energy, respectively. Then the constrained equation of motion is

$$\dot{r}_i = F_i + \alpha \dot{r}_i \tag{5}$$

and since $\dot{E}_k = 0$ or equivalently the value of the Lagrange multiplier α is

$$\alpha = -\frac{\sum_i^{N_i} F_i \cdot \dot{r}_i}{\sum_i^{N_i} \dot{r}_i{}^2}. \tag{6}$$

The isothermal version of the Leap-Frog integrator is then readily seen to be

$$\dot{r}_i(t+\Delta t/2) = (1+\alpha\Delta t)\dot{r}_i(t-\Delta t/2) + \Delta t(1+\alpha\Delta t/2)F_i(t) \tag{7}$$

$$r_i(t+\Delta t) = r_i(t) + \dot{r}_i(t+\Delta t/2)\Delta t \tag{8}$$

Note: The only (or one) drawback of Gaussian thermostats is that it can not change the temperature of system dynamically during the run. It just keeps the temperature constant. So, if we "start" the simulation by thermalizing our system in canonical ensemble in NVT run using Gaussian thermostat, we MUST set the initial velocities of the particles in the initial configuration input file.

- **Kinetic Nosé-Hoover thermostat** The Nose-Hoover thermostat provides a way to simulate a system which is (asymptotically, i.e. at large times) in the NVT ensemble. The idea is to introduce a fictitious dynamical variable, whose physical meaning is that of a friction, (, which slows down or accelerates particles until the temperature (measured through the kinetic energy and the equipartition function, see below) is equal to the desired value. The equations of motions (in 3D) are:

$$m_i \frac{d^2 r_i}{dt^2} = f_i - \zeta m_i v_i$$
$$\frac{d\zeta(t)}{dt} = \frac{1}{Q}[\sum_{i=1}^{N} m_i \frac{v_i^2}{2} - \frac{3N+1}{2} k_B T]$$

where Q in Eq. 5 determines the relaxation of the dynamics of the friction, $\zeta(t)$ while T denotes the target temperature. It can be seen that in steady state, where $\frac{d\zeta}{dt} = 0$, the kinetic energy is given by $\frac{3}{2}(N+1)k_B T$ as required by equipartition (there is a factor of $3N+1$ instead of 3N as there is one more degree of freedom, (). It is important to note that the temperature is therefore not fixed, rather it tends to the target value.

The equations of motion of the Nose-Hoover thermostat can be implemented by a small modification of the velocity Verlet algorithm an option is given below. The first four steps in the modified discretisation algorithm are:

$$\begin{aligned} r_i(t+\delta t) &= r_i(t) + v_i(t)\delta t + (\tfrac{f_i(t)}{m_i} - \zeta(t)v_i(t))\tfrac{\delta t^2}{2} \\ v_i(t+\delta t/2) &= v_i(t) + \tfrac{\delta t}{2}(\tfrac{f_i(t)}{m_i} - \zeta(t)v_i(t)) \\ f_i(t+\delta t) &= f_i(r_i(t+\delta t)) \end{aligned}$$

$$\zeta(t+\delta t/2) = \zeta(t) + \frac{\delta t}{2Q}\left[\sum_{i=1}^{N} m_i \frac{v_i(t)^2}{2} - \frac{3N+1}{2} k_B T\right]$$

To match the two-step character of the velocity Verlet algorithm, note that also $\zeta(t)$ is first updated at time $t + \delta t/2$. The final steps of the Nose-Hoover-Verlet algorithm are:

$$\zeta(t+\delta t) = \zeta(t+\delta t) + \frac{\delta t}{2Q}\left[\sum_{i=1}^{N} m_i \frac{v_i(t+\delta t)^2}{2} - \frac{3N+1}{2} k_B T\right]$$
$$\mathbf{v}_i(t+\delta t) = \frac{\left[\mathbf{v}_i(t+\frac{\delta t}{2}) + \frac{\delta t}{2}\frac{\mathbf{f}_i(t+\delta t)}{m_i}\right]}{1 + \frac{\delta t}{2}\zeta(t+\delta t)}$$

where the last equation is slightly more complicated than its counterpart in the NVE velocity Verlet algorithm, because the dissipative force over mass term, $\zeta\mathbf{v}$, is computed at time t + δt.

- **Configurational Nosé-Hoover thermostat**
- **Langevin thermostat**

- **Brownian (overdamped) dynamics**

Physical Observables

- **Radial Distribution Function (RDF)**
- **Velocity Autocorrelation Function (VACF)**
- **Root-Mean-Square Velocity (VRMS)**
- **Stress tensor and momentum**
- **Center-of-mass motion**
- **Space-time correlation functions**

Output and Utilities

The output files are saved in the `../output` folder. Ensure that you create this directory from where you run the program.

- Structured output files: `.xyz`, `.bond`, `.pair`, `.com`, `.result`
- Restart and resume capability with `.restart` and `.state` files
- Clean separation of source code, unit tests, and output
- Support for Lees-Edwards boundary conditions for sheared systems
- Configurable halting conditions (based on VRMS or custom metrics)
- Modular design for easy extension of potentials and features

Installation Instructions

Prerequisites

1. **GCC** compiler: Ensure that `gcc` is installed for compiling C code. You can install it using:

   - **Ubuntu/Debian**: `sudo apt-get install build-essential`
   - **Fedora/CentOS**: `sudo dnf install gcc`
   - **macOS** (via Homebrew): `brew install gcc`

2. **MPICH**: Ensure you have the **MPICH** library installed for MPI-based operations. This is required for parallel computation.

   - **Ubuntu/Debian**: `sudo apt-get install libmpich-dev`
   - **Fedora/CentOS**: `sudo dnf install mpich`
   - **macOS**: `brew install mpich`

Building Lamina

1. **Clone the Repository**

   If you haven't cloned the repository yet, use the following command to clone it from GitHub:

   ```
   git clone https://github.com/your-username/Lamina.git
   ```

2. **Navigate to the Project Directory**

   Change into the `Lamina` project directory:

   ```
   cd Lamina
   ```

3. **Edit the Makefile (Optional)**

   If needed, edit the `Makefile` to adjust the paths to your **MPICH** installation or any other custom settings. The current `Makefile` expects the following:

   - **MPICH** headers: `/usr/include/mpich-3.2-x86_64/` (edit `INCLUDE` variable if different)
   - **MPICH** libraries: Linked via `LIBS = -lm`

   If your **MPICH** installation differs, adjust the paths as needed.

4. **Build the Executable**

   Run the `make` command to build the executable:

   ```
   make
   ```

   This will compile the source files and generate the `main` executable. If the build is successful, the output will indicate that the executable has been created.

5. **Clean the Build (Optional)**

   If you want to remove the compiled objects and the `main` executable, you can run:

   ```
   make clean
   ```

Running Lamina

After successfully compiling **Lamina**, you can run the simulation as follows:

```
./main <output_prefix>
```

Where `<output_prefix>` is a string that will be used to create output files in the `output/` directory (e.g., `./main simulation_run`).

License

Lamina is open-source software, and you are welcome to modify and distribute it under the terms of the MIT License.