libmd-intro

- By Benny van Zuiden -

April - 8 - 2014

last revised April - 8 - 2014

1. Introduction to libmd

libmd is a molecular dynamics library optimized for theoretical soft condensed matter physics. It is developed by:

Benny van Zuiden[1] (zuiden@ilorentz.org)
Jayson Paulose (paulose@ilorentz.org)
Thomas Beuman (beuman@ilorentz.org)

under the supervision of Vincenzo Vitelli. libmd is a object-oriented library written in C++11 using nothing but the standard libraries and is tested to work on gcc and LLVM/clang. It uses git as a version control system, and GNU make as a build system. Parallel implementations for cpu is underway and implementations gpu planned. What makes libmd different from all the other molecular dynamics packages is that it is designed to solve typical soft matter problems, it designed to work in arbitrary dimensions and allows, is *extremely* flexible, has multiple ways to make particles interact and constrain them and it can do dynamics in curved space (Monge patches).

2. Introduction to molecular dynamics

Molecular dynamics in the eyes of a theoretical softmatter physicist is nothing but the classical N-body problem –solved computationally. The classical N-body problem is basically solving Newton's equations of motion:

$$a_i^\mu = \frac{1}{m_i} F_i^\mu \tag{2.1}$$

Here m_i is the mass of particle i, a_i^μ is the geodesic equations[2] and F_i^μ is probably non-trivial and based on lots of interactions. The classical N body problem was analytically solved by Sundman and Wang and is of little practical use. Fortunately, it easy to solve the problem computationally –for most cases– as time discretized. The error remains small if symplectic integrators –integrators that conserve Lagrangian symmetries are conserved – are used. Typical integrators in flat space are symplectic Euler or velocity Verlet. For curves space we have developed our own.
As molecular dynamics is nothing less than an initial value problem, it requires initial conditions (possibly boundary conditions) and rules about how to calculate the forces.

[1] Maintainer.
[2] In flat space: $a_i^\mu = \ddot{x}_i^\mu$, in curved space it's given by: $a_i^\mu = \ddot{x}_i^\mu + \Gamma^\mu_{\rho\sigma} \dot{x}_i^\rho \dot{x}_i^\sigma$.

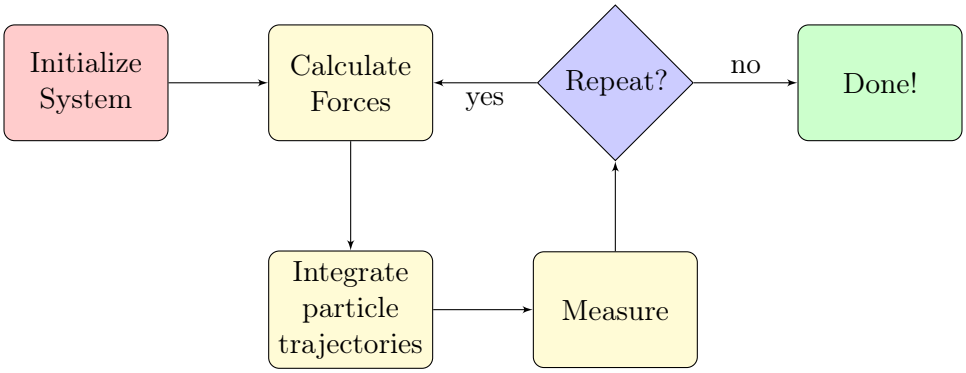

Figure 1: A very basic schematic depicting how molecular dynamics is roughly done

Once this is defined the integrator can calculate the particle trajectories sequentially using small time increments, called timesteps. After a bunch of timesteps one typically wants to measure a property after which the process can be repeated or terminated. This process is very schematically depicted in fig. 1.

3. `libmd`-specific

In the following section we will explain how `libmd` is designed, what it can –and perhaps cannot– do. Bear in mind that everything discussed in this is implemented regardless of dimensionality or presence of curvature –unless specifically stated otherwise.

a. The simulation box

We start by building a stage we the actors –particles– can act. The stage is called the simulation box –simbox for short– and has a hyper volume:

$$V = \prod_{\delta}^{D} L^{\delta} \tag{3.a.1}$$

where D is the dimension of the system and L^{δ} is the size of the box in each component. The origin 0^{μ} is placed right in the middle of the box such that the boundaries of the box are present at every $\pm\frac{1}{2}L^{\delta}$. A two dimensional box is depicted in fig. 2. The boundaries can have boundary conditions per dimension. Currently, the implemented boundary are `NONE`, `PERIODIC` and `HARD` boundary conditions make perfectly bounce the particles. Additionally, the box can shrink, expand and shear.

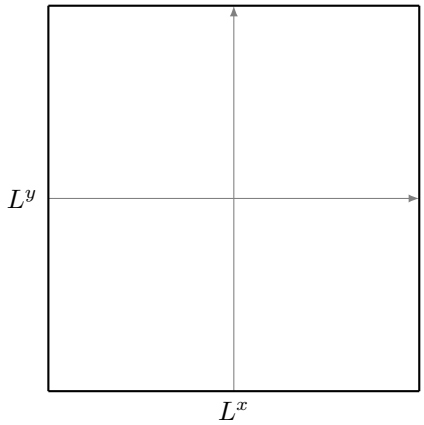

Figure 2: A two dimensional box with dimensions L^x by L^y. The origin of the box is where the grey axes cross.

b. Particles

Particles are the actor is in the box. Particles can be added and removed dynamically from the system. Additionally they can be pinned to one point. Every particle has the following properties:

- mass: m
- position: x^μ
- velocity: $\dot{x}^\mu$

Additionally particles have:

- type[3]
- fix
- forces acting on them: F^μ

Internally they contain additional *hidden* variables for better computational performance.

c. Interactions

Without interactions life is boring. Currently, the defined system is nothing but a box with a set of points, if we would calculate the equations of motion we would be simulating an ideal gas. To make it a bit more interesting –or realistic– we have to define interactions. Interactions can be defined in many ways, for this part of the interaction implementation we will assume interactions are position depended and pairwise potentials. Mathematically:

$$F_i^\mu = -\nabla \sum_{j>i} V(d(p_i, p_j)) \tag{3.c.1}$$

Where V is the pairwise potential and $d(p_i, p_j)$ is the distance between two particles. The potential V can be any kind of potential Lennard–Jones or Yukawa for instance. The Yukawa potential:

$$V_Y(r) = \frac{b}{re^{\kappa r}} \tag{3.c.2}$$

Typical values for are $b = 1$ and $\kappa = 100$. Note that for these parameters $V(1) \approx 10^{-44}$ so it does not make sense to iterate over at large distances. Additionally making every particle interact with every other particle makes the algorithm at least order N^2 which is slow for a large number of particles. We can do better by creating some kind of cutoff radius r_{co} and iterate over particles only within that cutoff vicinity, we can make the algorithm order $Nlog(N)$ which is typically better. In order to keep the potential continues we redefine the potential as:

$$\tilde{V}_Y(r) = V_Y(r) - V_Y(r_{co}) \tag{3.c.3}$$

[3]Types will be discussed in the next subsection.

The library can take any function $V : \mathbb{R} \to \mathbb{R}$ it generates the derivative automatically –without numerical errors and with the same complexity– using a nice computational trick called *automatic differentiation*.
Suppose you want certain particles to interact differently with other particles. A trivial way of doing this is by labeling particles with a type and then define interactions between types. Suppose I label a set of particles as type 0 and the remaining particles as type 1. I'll let 0–0 interaction be Yukawa, 0–1 interact Lennard–Jones and 1–1 interact Yukawa with different parameters again. Or suppose you want to make a random spring network then you will need as many types as particles. If you don't like to think in types however, the code can translate bond lists to types for you.
Suppose you want to simulate dimers, trimers or polymers (or other weird objects for that matter). It can be done. We have included a structure in the code called superparticles that overrides type defined interactions if an interaction in the superparticle is assigned. This may sound a little vague so let me give an example. Take two particles of type 0 and put them in the same superparticle, then within the super particle tie them together with a Hookean spring. The result is a dimer that interacts Yukawa with everybody else. Likewise you can create polymers and dimers.

d. External forces

Additionally there is an external forces structure that allows you to add forces like linear gravity or damping, as well as any other kind of interaction you can write a function for. You can have multiple external forces acting on each particle. You can have damped interactions, multibody interactions, random kicking etc. The implementation of such forces is, however, a little harder.

e. Curvature

Adding curvature in the form of a Monge patch is near trivial. The Monge functions can be any (external) function $f : \mathbb{R}^D \to \mathbb{R} : C^2$ the code are automatically differentiates them.