Title of the master thesis

by

Filip Henrik Larsen

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2017

Abstract

This is an abstract text.

To someone

This is a dedication to my cat.

Acknowledgements

Flora Joelle Larsen
Anders Malthe-Sørenssen
Anders Hafreager
Henrik Sveinson
Kjetil

Contents

Chapter 1

Introduction

Why is the subject of this thesis of any interest?
What is our take on the problem?
What do we hope to accomplish?
How will this be of any contribution to anything?
How is the thesis laid out?

Part I

Theory and methods

Chapter 2

Molecular dynamics

Molecular dynamics is a method that uses Newton's equations of motion to simulate the movement of interacting atoms. Each atom is treated as a point particle with mass and charge, and the interaction between them is described by a force field. Thus, the force acting acting on a particle is computed as

$$\mathbf{F_i} = -\nabla U_i(\mathbf{r}^N), \tag{2.1}$$

where $\mathbf{r}^N = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ is the position of all atoms in the system. All the detail of the simulation lay in the interaction potential U. In order to obtain realistic simulations, the potentials often doesn't only consist of a pair-interaction, but many-body interactions as well. Also, methods for controlling the thermodynamic properties and boundary conditions pose additional complications. Lastly, the computational efficiency aspect of the simulation has to be evaluated. The computation time is predominantly spent in computing the forces. Thus, our methods must be able to ignore negligible parts of the force contributions, but still maintain a satisfactory level of accuracy.

Molecular dynamics aid our understanding of macroscopic phenomena by giving us insight in microscopic effects and behavior. It can be used if an experiment is to difficult, dangerous or expensive to do otherwise, for instance experiments at extreme temperatures or pressure. The areas of application are wide, and it has become a very popular field in material science, biochemistry and biophysics.

In this chapter we will present the basic principals of molecular dynamics, including essential algorithms, common methods and efficiency improvements.

2.1 Time-integration

The evolution of particles position and velocity is carried out through time integration. In molecular dynamics, atoms movement is governed by Newton's

second law

$$\sum_{i=1}^{N} \mathbf{F_i} = m_i \mathbf{a_i} \tag{2.2}$$

Theoretically, the computation of particles movement is easy to assimilate. We know that acceleration is the derivative of velocity, which in turn is the derivative of position. Thus we should be able to retrieve the velocities and positions using time-integration of the acceleration.

$$\mathbf{v_i}(t) = \mathbf{v_i}(0) + \int_0^t \mathbf{a_i}(t)dt \tag{2.3}$$

$$\mathbf{r_i}(t) = \mathbf{r_i}(0) + \int_0^t \mathbf{v_i}(t)dt \tag{2.4}$$

When solving an integral numerically, we're forced to discretize the problem and use an approximation method. This is done by dividing the domain into steps of size Δt and approximate the area of the function inside each such partition. Typically, the error of the approximation decreases when reducing the time step Δt. However, a smaller step length leads to higher computational expense, and may limit the size or time-frame we are able to compute. These are common considerations when doing computational research. One has to balance precision, size/time-frame, time and expense. It is therefore very important to choose a numerical scheme that has a satisfying level of precision as well as speed.

There are numerous numerical integration methods, but not all these have properties that are desirable for molecular dynamics simulation. The most fundamental features the scheme should have are high precision, computationally cheap, good energy conservation, reversibility and be deterministic. Conservation of energy is fundamental in physics, and in order to simulate the micro-canonical ensemble (NVE), the integration method cannot have an energy drift. The dynamics should be reversible, meaning one should be able to simulate the system in the opposite direction in time, and arrive at past states. It should be deterministic in the sense that by the information about a specific state $(r(t_0), v(t_0))$ one should be able to compute the the state at any other time $(r(t), v(t))$, be it future of past. Molecular dynamics simulations are computational expensive, and practically all the work lies in the time-integration loop. More precisely computing the sum of forces on each atom. The choice of integration scheme is therefore very important. As it turns out, probably the best scheme for the task is also one of the simplest.

2.1.1 Velocity Verlet

In molecular dynamics the most common scheme for time-integration is the *Velocity Verlet* algorithm, which is a specific type of Verlet integration. It is derived

by taylor expanding the position both one time step forward and one backwards as follows:

$$r(t+\Delta t) = r(t) + \frac{\Delta t}{1!}r'(t) + \frac{\Delta t^2}{2!}r''(t) + \frac{\Delta t^3}{3!}r'''(t) + \mathcal{O}(\Delta t^4) \tag{2.5}$$

$$r(t-\Delta t) = r(t) - \frac{\Delta t}{1!}r'(t) + \frac{\Delta t^2}{2!}r''(t) - \frac{\Delta t^3}{3!}r'''(t) + \mathcal{O}(\Delta t^4) \tag{2.6}$$

If we add these two together, the odd terms cancel. After rearranging, we are left with the very simple verlet algorithm.

$$r(t+\Delta t) = 2r(t) - r(t-\Delta t) + \Delta t^2 r''(t) + \mathcal{O}(\Delta t^4) \tag{2.7}$$

Or written in a compact notation where

$$\mathbf{r}^n = r(t) \qquad \mathbf{r}^{n\pm1} = r(t \pm \Delta t) \qquad \mathbf{a}^n = \frac{\partial \mathbf{v}^n}{\partial t} = \frac{\partial^2 \mathbf{r}^n}{\partial t^2} \tag{2.8}$$

this can be expressed as

$$\mathbf{r}^{n+1} = 2\mathbf{r}^n - \mathbf{r}^{n-1} + \Delta t^2 \mathbf{a}^n \ . \tag{2.9}$$

Notice that we do not rely on information about the velocity in order to predict the next position. The algorithm in equation (2.9) is known as Strömer-Verlet. It's not self-starting, since you need to know the previous and current steps to predict the next. Thus, it needs to be initialized in a way, e.g. by using algorithm (2.11) for the first step. It has a high order truncation error, proportional to $\mathcal{O}(\Delta t^4)$, however it's vulnerable to round-off errors. This is because the last term is proportional to Δt^2, and is potentially much smaller than the other terms. When trying to add it with the other terms, this may be a source of round-off error [4]. As a remedy to this problem, the Velocity Verlet algorithm incorporates a second order differential approximation of the velocity.

$$\mathbf{v}^n = \frac{\mathbf{r}^{n+1} - \mathbf{r}^{n-1}}{2\Delta t} + \mathcal{O}(\Delta t^2) \tag{2.10}$$

Resulting in

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t \mathbf{v}^n + \frac{\Delta t^2}{2}\mathbf{a}^n \tag{2.11}$$

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t \left(\mathbf{v}^n + \frac{\Delta t}{2}\mathbf{a}^n \right) \tag{2.12}$$

We may recognize the parenthesis as $\mathbf{v}^{n+1/2}$, leaving us with

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t \mathbf{v}^{n+1/2} \ . \tag{2.13}$$

Next, we taylor expand the velocity and use a first order approximation of the acceleration.

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta t \mathbf{a}^n + \frac{\Delta t^2}{2}\dot{\mathbf{a}}^n + \mathcal{O}(\Delta t^3) \tag{2.14}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta t \mathbf{a}^n + \frac{\Delta t^2}{2}\frac{\mathbf{a}^{n+1} - \mathbf{a}^n}{\Delta t} \tag{2.15}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{\Delta t}{2}\left(\mathbf{a}^{n+1} + \mathbf{a}^n\right) \tag{2.16}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^{n+1/2} + \frac{\Delta t}{2}\mathbf{a}^{n+1} \tag{2.17}$$

Equations (2.13) and (2.17) define the Velocity Verlet algorithms. The final Velocity Verlet algorithm is thus:

$$\mathbf{v}^{n+1/2} = \mathbf{v}^n + \frac{\Delta t}{2}\mathbf{a}^n \tag{2.18}$$

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t \mathbf{v}^{n+1/2} \tag{2.19}$$

$$\mathbf{a}^{n+1} = -\nabla U(\mathbf{r}^{n+1})/m \tag{2.20}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^{n+1/2} + \frac{\Delta t}{2}\mathbf{a}^{n+1} \tag{2.21}$$

The Velocity Verlet algorithms has the same truncation error as the original, but a lower round-off error. This algorithm has good energy conservation, and is symplectic.

SOME MORE

2.2 Potentials

2.2.1 Lennard-Jones

One of the simplest and most known potentials is the Lennard-Jones potential. It was first proposed in 1924 by John Edward Lennard-Jones. It takes the form

$$V(r) = 4\epsilon\left[\left(\frac{\theta}{r}\right)^{12} - \left(\frac{\theta}{r}\right)^{6}\right], \tag{2.22}$$

where the first term represents Pauli repulsion due to overlapping electron orbitals, and the last represents the van der Waal force. The constant θ express the distance at which the inter-atomic potential is zero, while ϵ is the depth of the well, also regarded as the strength of the potential. r is of course the inter-atomic distance. A plot of the potential is shown in figure 2.1. The inter-atomic distance at which the potential is at it's minimum can easily be shown to be $r = 2^{1/6}\theta$.

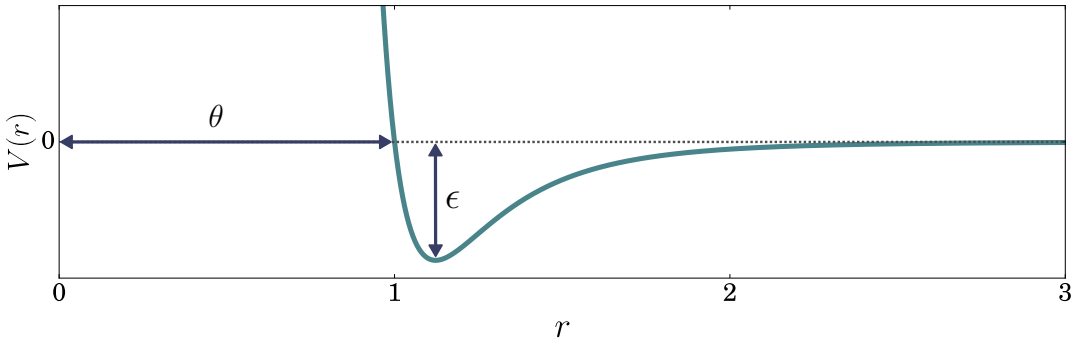


Figure 2.1: Lennard-Jones potential as function of inter-atomic distance, r, in units of θ.

Since the potential only account for the Pauli repulsion and van der Waal forces, its applicability is limited to systems consisting of neutral atoms or molecules, where there are no bonds present. It would suffice for studying noble gases, for instance, but that quickly becomes quite dull.

2.2.2 Stillinger-Weber

2.2.3 Vashishta

The Vashishta potential consists of two terms that represent two- and three-body interactions respectively. These incorporate physical effects such as steric repulsion, coulomb interaction, dipole interaction, Van der Waal interaction and energy related to covalent bonds. It can be expressed as

$$V = \sum_{i<j} V_{ij}^{(2)}(r_{ij}) + \sum_{i<j<k} V_{ijk}^{(3)}(\mathbf{r}_{ij}, \mathbf{r}_{ik}), \tag{2.23}$$

where $\mathbf{r}_i$ represents the position of the i-th atom, $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$, and $r_{ij} = |\mathbf{r}_{ij}|$ is the distance between atom i and atom j. The two-body term, denoted by superscript (2), sums over all atoms j within a cutoff range r_c from atom i, and is given as

$$V_{ij}^{(2)}(r) = \frac{H_{ij}}{r^{\eta_{ij}}} + \frac{Z_i Z_j}{r} e^{-r/r_{1s}} - \frac{D_{ij}}{2r^4} e^{-r/r_{4s}} - \frac{W_{ij}}{r^6}. \tag{2.24}$$

where H_{ij} and η_{ij} are the strength and exponent of the steric repulsion respectively, Z_i is the effective charge of atom i, r_{1s} and r_{4s} are screening lengths for the coulomb and dipole interaction respectively, D_{ij} is the strength of the dipole interaction, and W_{ij} is the strength of the Van der Waal interaction.

The three-body term of the potential, denoted by superscript (3), sums over all atoms j and k , that are covalently bonded to atom i, within a cutoff range

of r_0, and is expressed as

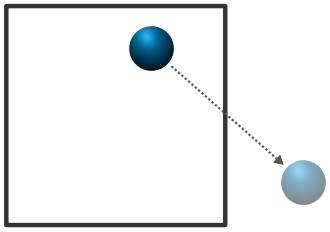
$$V_{ijk}^{(3)}(\mathbf{r}_{ij}, \mathbf{r}_{ik}) = B_{ijk} \exp\left(\frac{\xi}{r_{ij} - r_0} + \frac{\xi}{r_{ik} - r_0}\right) \frac{(\cos\theta_{ijk} - \cos\theta_0)^2}{1 + C_{ijk}(\cos\theta_{ijk} - \cos\theta_0)^2} \tag{2.25}$$

where B_{ijk} is the strength of the three-body term, ξ and C_{ijk} are constant parameters, and θ_{ijk} is the angle between $\mathbf{r}_{ij}$ and $\mathbf{r}_{ik}$. The values of all these parameters depend on the system at hand. For instance, the strength of the steric repulsion, H_{ij}, for Si-Si interaction is different from that of Si-O interactions. For the case of silica, these metrics have been computed already, and when using the Vashishta potential in LAMMPS there is a file containing them. As for the Lennard-Jones potential, the two-body potential term is truncated and shifted to prevent unphysical jerks, and retain stability. This is done the same way as in equation (2.34).

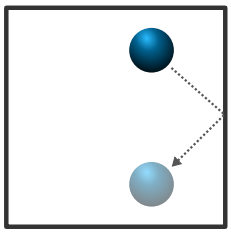
2.3 Boundary conditions

Boundary conditions is a crucial detail to decide upon when setting up a molecular dynamic experiment. The way we treat atoms at the boundary can have an immense effect on their behavior and how physically reasonable the results will be. There are several types of boundary conditions one may desire, and one may define custom conditions. A few examples are listed below.

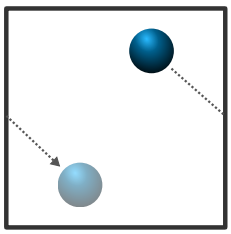
No boundary conditions. Particles are not subject to any special rules at any boundry. The system size may be regarded as infinite. This might be a reasonable choise when studying explosions for instance, or in experiments that is on a really short time scale.



Reflecting boundary conditions, which act as hard walls. In stead of passing through the boundary, the velocity component in the direction normal to the face of the boundary changes sign, thus causing the atoms to be confined within the simulation box.



Periodic boundary conditions, which allow atoms on separate sides of a boundary to interact through the boundary as if they were neighbors, and atoms crossing the boundary reappears on the other side of the simulation box. This is useful when studying bulk atoms of a material or materials that has a periodic structure.



The boundary conditions may include more details than only how to handle the trajectory of atoms intersecting the boundaries. For instance with periodic boundaries, atoms that are far apart are effectively close. These atoms should therefore interact, since their motion would seem artificial otherwise. Thus, the interpretation of atoms positions must in some cases be included in the boundary conditions. In order to interpret atoms positions in the case of periodic boundaries, one has to remember that an atom should not interact with multiple images of another atom. Which image, if not the original, must therefore be decided. Also, an atom should not interact with an image of itself. That could potentially cause strange correlations. The standard way of determining atoms positions is by using the *minimum image convention*.

2.3.1 Minimum image convention

The minimum image convention simply states that the periodic image of an atom closest to the reference atom is the one to consider as its position. Example cases of 1D and 2D systems are shown in figure 2.2 and figure 2.3 respectfully. In the

examples we look at systems consisting of two atoms and we try to determine the minimum image of the smaller, blue atom with respect to the larger, red, reference atom. The original system is distinguishable from the periodic copies by the background color; they have a slightly darker background. Also, we only bother showing the periodic copies of the atom of consideration, the blue one. The systems depicted are of size L and $L \times L$ in the 1D and 2D cases respectfully.

By studying figure 2.2 it becomes apparent that if the atom of consideration is closer to the reference atom than half the system length, the original image will be used to interpret its position. Otherwise, if it's farther apart than half the system length, a periodic image will be closer and therefor used as the atoms position. This principle is applicable at higher dimensions as well. We simply decompose the positions and check each dimension separately. For a system of arbitrary number of dimensions the minimum image convention is assured by using the progression of lisiting 2.1. Figure 2.3 illustrate how we decompose the positional separation of the atoms in the 2D case.

```
for (int k=0; k<numberOfDimension; k++){
  delta[k] = atom[j].pos[k] - atom[i].pos[k]
  if (delta[k] > L/2) {
    delta[k] = delta[k] - L
  }
  else if (delta[k] < -L/2){
    delta[k] = delta[k] + L
  }
}
```

Listing 2.1: Loop to compute the position of the closest periodic image of atom j with respect to the referance atom i.

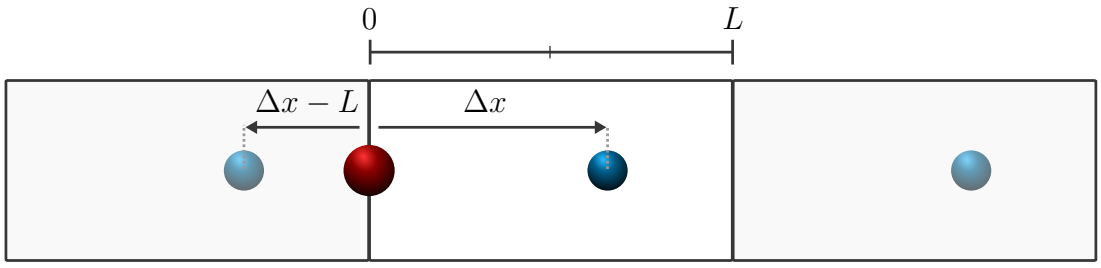


Figure 2.2: Minimum image convention in 1 dimension. If an atom is separated from the reference atom (red) by more than half the system length, $L/2$, the minimum image convention states that the distance between the two is the distance to the periodic copy, which is $|\Delta x - L|$. The white area indicates the system, while the gray areas are periodic replications of the system. Atoms in the gray areas are periodic images of the atom of consideration in the white area.

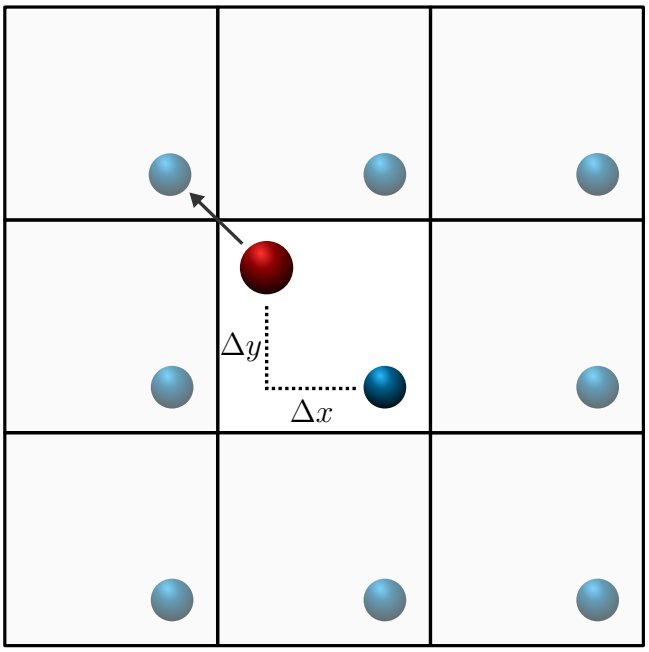


Figure 2.3: Minimum image convention in 2 dimensions. The position of an atom, with respect to the reference atom (red), is the position of the original or any of the replicated images that has the shortest distance to the reference atom. The minimum image is found by decomposing the atoms position, and carry out the The shortest distance in this case is marked with an arrow. Thus, when computing positional dependent quantities it's the position of the north-west replica that will be considered as the position of the blue atom in this case.

2.4 Measuring physical quantities

2.4.1 Energy

In molecular dynamics we sample kinetic energy classically as

$$E_K = \frac{1}{2}mv^2. \tag{2.26}$$

The potential energy is off course dependent on the potential force field in use U, and is computed as

$$E_P = \sum_{i=1}^{N}\sum_{j>i}^{N} U(\mathbf{r}_i, \mathbf{r}_j). \tag{2.27}$$

2.4.2 Temperature

From thermodynamics we have that the kinetic energy, or total thermal energy is

$$E_k = \frac{f}{2} N k_b T, \tag{2.28}$$

for a system of N atoms with f kinetic degrees of freedom and temperature T. k_b is the Boltzmann constant. Since we assume point particles, we take only the translational degrees of freedom into account and ignore the rotational degrees of freedom. If we assume Newtonian motion, the total kinetic energy in the system is found as

$$E_k = \frac{1}{2} \sum_{i=1}^{N} m_i {\mathbf{v}_i}^2. \tag{2.29}$$

We can equate these to find the instantaneous temperature given as

$$T = \frac{\langle m_i {\mathbf{v_i}}^2 \rangle}{f k_b}. \tag{2.30}$$

Thus, we can approximate the instantaneous temperature using the atoms velocities. This value will fluctuate about the real value, so time-averaging is often used when measuring. In many cases it is desirable to control the temperature of the system, e.g. when using the canonical ensemble (NVT). Methods for doing this are called *thermostats*, and are described in section 2.5.

2.4.3 Pressure

There are many methods for computing the pressure of the system. Normally for a N-body system of volume V and particle density $\rho = N/V$ it is taken from the virial equation for pressure.

$$P = \rho k_B T + \frac{1}{dV} \Big\langle \sum_{i<j} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \Big\rangle \tag{2.31}$$

where d is the number of dimensions, $\mathbf{F}_{ij}$ is the force contribution on atom i from atom j positioned at $\mathbf{r}_{ij}$ relative to atom i. In this expression for the pressure we assume constant N,V and T, i.e. the canonical ensemble. The equation will not be the same for the micro-canonical ensemble. There are methods for converting averages from one ensemble to another [3], but that is outside the scope of this thesis.

2.4.4 Something else?

2.5 Thermostats

Often, we want to control the temperature of our simulated system. This is done by fixing the temperature in equation (2.30) and manipulate the atoms velocities. There are several methods of manipulating the velocities, and these are called *thermostats*. There are many different types of thermostats, and they all have their strengths and weaknesses.

We will look at two quite basic and different thermostats, as well as one that is common in professional use.

2.5.1 Berendsen

The Berendsen thermostat rescales the velocity of all atoms in the system so that the temperature approaches that of the surrounding heat bath. It rescales the atoms velocity by multiplying them by a scaling factor γ, given as

$$\gamma = \sqrt{1 + \frac{\Delta t}{\tau}\left(\frac{T_{bath}}{T} - 1\right)}, \tag{2.32}$$

where Δt is the time step length, and τ is a relaxation time, which tune the coupling to the heat bath. T_{bath} and T is the temperature of the heat bath and the current temperature of the system, respectfully. Typically the relaxation time is set to $\tau \approx 20\Delta t$. Using $\tau = \Delta t$ would result in the velocities rescaling so the temperature would be exactly the target temperature, T_{bath}, every time step. This thermostat suppress temperature fluctuations efficiently, rendering it well suitable for equilibration procedures. However, because it rescales the velocities, it produce dynamics that are inconsistent with the canonical ensemble.

2.5.2 Andersen

The Andersen thermostat simulates the coupling between the system and an imaginary heat bath as collisions between their atoms. Atoms that collide are assigned a new normally distributed velocity with standard deviation $\sqrt{3k_B T_{bath}/m}$ about the target temperature, T_{bath}. The procedure of the thermostat goes as follows: For each atom in the system, we generate a random uniformly distributed number in the interval $[0, 1]$. If this number is less than $\Delta t/\tau$, the atom is considered to be colliding, and is assigned a new velocity. Here, τ is regarded as a collision time, and its value should be similar to τ in the Berendsen thermostat ($20\Delta t$). The Andersen thermostat is useful when equilibrating systems, but because randomly chosen particles are assigned randomly distributed velocities, it disturbs the dynamics of e.g. lattice vibrations. Also, this disturbance will

decorrelate the system, rendering this thermostat ill-equipped when measuring e.g. diffusion coefficient. As a final note, if this thermostat is to be used, the newly assigned velocities may cause the systems center of mass to drift.

2.5.3 Nosé-Hoover

The most widely used thermostat is the Nosé-Hoover thermostat. It produces accurate dynamics and realistic constant temperature conditions. This thermostat does not rescale the velocity of atoms. Instead it alters the acceleration by adding a fictitious force of friction

$$\mathbf{F}_i = -\nabla U_i(\mathbf{r}^N) - \xi m \mathbf{v_i} \tag{2.33}$$

2.6 Efficiency improvements

The major part of the CPU time is spent in the force loop. At every time step we must recompute the force acting on each individual atom. When doing so, we should in theory include the contribution from all other atoms. Having a system consisting of N atoms would result in $N(N-1)/2 \propto N^2$ computations, if we apply Newtons third law. In this section we will look at the most fundamental efficiency improvements applied in molecular dynamics simulations.

2.6.1 Cut-off

Depending on the potential in use, the forces become negligible at certain distances. For instance if one uses the Lennard-Jones potential (introduced in section 2.2.1) the contributions are practically zero for atoms positioned at a distance $r \geq 2.5\sigma$. Therefore, during a simulation we choose to only account for the contributions from atoms closer than this *cut-off* distance, denoted r_c. For the Lennard-Jones potentials the cut-off range is usually set to 2.5θ. Past this range, the pair-potential is practically zero. It's not zero, though. Without any alterations, particles intersecting the cut-off limit will experience an unphysical jerk, which may render the simulation unstable. To counteract this effect, we raise the potential, ensuring it and its derivative to be zero at the cut-off. We use the following adjustment.

$$V(r) = \begin{cases} V(r) - V(r_c) - \frac{\partial V(r)}{\partial r}\Big|_{r=r_c} (r - r_c) & , r \leq r_c \\ 0 & , r > r_c \end{cases} \tag{2.34}$$

which implies the force

$$F(r) = \begin{cases} F(r) = F(r) - F(r_c) & , r \leq r_c \\ 0 & , r > r_c \end{cases} \tag{2.35}$$

In practice we need to keep track of which atoms are within the cut-off range of each atom. This is achieved using cell lists and neighbor lists.

2.6.2 Cell lists and neighbor lists

The main purpose of the cell list is to make the building of neighbor lists more efficient. We need to check which atoms are neighboring atoms, but obviously we do not need to check the entire domain, since the cut-off length is relatively small. Therefore, we partition the system into several cubes of size equal to the cut-off length. We store the atoms contained by each cell in a *cell list*. Finally, when we build the neighbor lists we check only the atoms within the neighboring cells and those in the same cell, 27 cells in total. In this sense, by neighboring cells we mean any cells that share a face, edge or vertex. This is illustrated in figure 2.4. The pay-off of using cell- and neighbor lists is tremendous. Since we now, for each atom, only include contributions from atoms within a constant volume of size $4\pi {r_c}^3/3$, the number of contributions will only depend on the density, which is an intensive[1] property. Thus, the number of computations is reduced to $\mathcal{O}(N)$, which is an immense relief in computational expense!

2.6.3 Parallelization

It might be misguiding to refer to parallelization as an efficiency improvement, when on the contrary it most likely increases the CPU time usage. However, the real time consumed may be greatly decreased. It is intuitive that partitioning the work and processing these simultaneously will decrease the time as compared to processing it serially. The speedup is defined as

$$S = \frac{T_s}{T_p}, \tag{2.36}$$

where T_s is the time used when executing the program on a single processor, and T_p the time used when running on p processors simultaneously. The time spent running a parallel implementation of a code using p processors is seldom trivially $T_p = T_s/p$. This is due to the fact that there is a certain amount of time used on *overhead*. This includes interprocess communications, idling and excess

[1] Physical property of a system that does not depend on the system size.

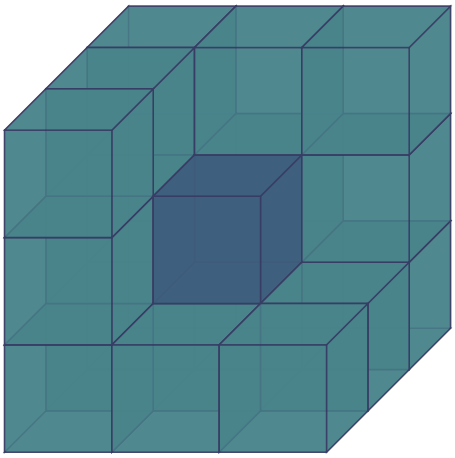


Figure 2.4: Illustrative figure of cells of concern when building the neighbor lists. The lighter cubes are neighboring cells; the darker cube is the cell containing the reference atom. The 7 cells in front of the dark cell are removed from the figure, but are also included.

computations. In molecular dynamics simulations there is communication between processors when building the cell- and neighbor lists, and when computing thermodynamical properties such as energy, pressure, temperature, etc..

During this project we have mainly been using the local supercomputer at the department of physics at the University of Oslo. It provides users with the possibility to run up to 256 processes at once. Though, before doing so, it is considered as good practice to check the speedup obtained by using several numbers of cores. In order to compute the speedup we initialized a system containing $15 \times 15 \times 15$ unit cells of beta-cristobalite and saved it as a restart file. We then remotely ran the input script shown in Listings 2.2 from the supercomputer using 1, 2, 4, 8, 16, 32 and 64 processors in the same fashion as shown in Listing 2.3. The resulting speedup of using the respective number of processors is plotted in figure 2.5.

```
include "system.in.init"
read_restart ${filename}
include "system.in.settings"

variable N equal 1
variable T equal 293

neighbor  0.3 bin
neigh_modify  delay 10

timestep  0.002

dump myDump slabUpperGroup atom 1 dumpFiles/surface_*.dump

```

```
15 fix nvt all nvt temp ${T} ${T} 1.0
16 run ${N}
```

Listing 2.2: LAMMPS input script executed using several numbers of processors, and timed separately.

```
1 mpirun -n 8 lmp_mpi -in speedup.in -var filename
    speedup.restart
```

Listing 2.3: Command used to excecute the input script speedup.in on 8 parallel processors and set the filename variable to speedup.restart.

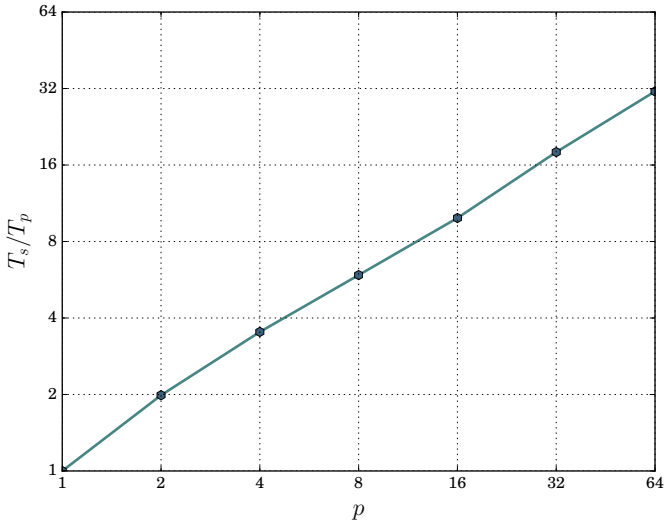


Figure 2.5: Speedup as a function of number of processors.

The result indicate that the speedup is in fact not simply $S = p$. Anyhow, we clearly see the advantage of using 64 processors in parallel as opposed to 1. We can finish a job that would have taken an hour in two minutes! Also, we clearly see that the initial claim holds; this is not more efficient when regarding CPU time. In fact this result suggest that using 1 processor is twice as energy efficient as using 64.

Chapter 3

Friction, elasticity and contact mechanics

Friction can be defined as *The force that resists relative motion between two bodies in contact* [1]. It is a well known phenomena, though still not completely understood. Despite the fact that we learn about friction in introductory courses to physics, it is not at all easy to comprehend. It's very complex and still a field of research. The effects of friction is affected by surface roughness, material properties, ambient conditions, the geometry of contact area, and many other factors. In this chapter we will study some elementary aspects of friction, elasticity and contact mechanics.

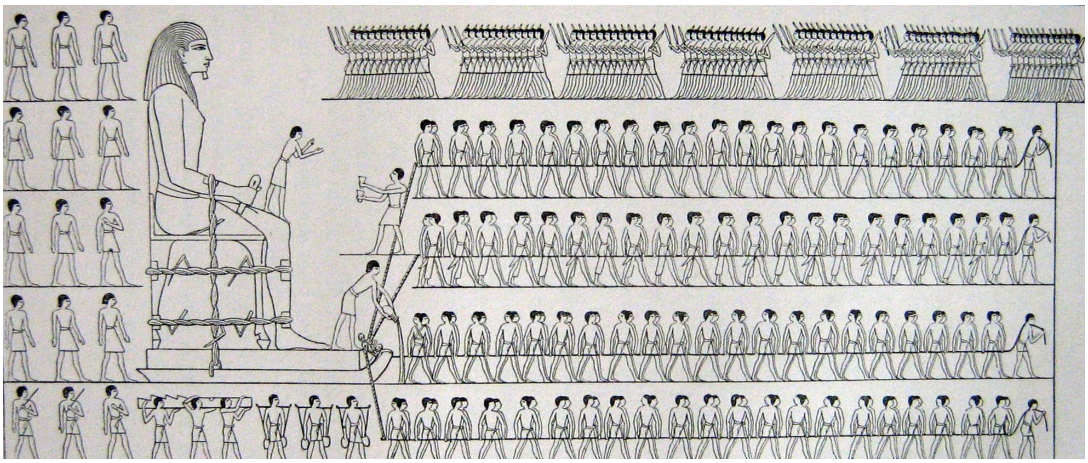


Figure 3.1: Painting from the tomb of Tehuti-Hetep, showing the transportation of an Egytian colossus.

3.1 Historical note

On the macroscopic level, friction has been observed since the dawn of man. Early actions that show awareness of the effects of friction was to chip stone in order to make tools, or rubbing wood in order to create fire. In the tomb of Tehuti-Hetep there was found the painting shown in figure 3.1, which show the moving of an Egyptian colossus. The statue is depicted pulled on a sledge, with officers standing on the front end of the sledge poring water on the ground. This is evidence that they perceived the effect of lubrication even then, 1900 B.C..

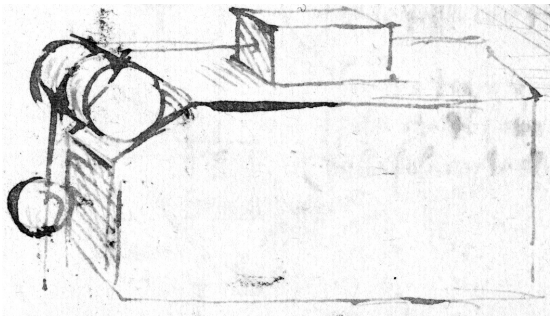


Figure 3.2: Drawing from Leonardo da Vinci's notebook. From Codex Arundel, British Library, London (Arundel folio 41r).

The first documented studies of friction was carried out during the renaissance by Leonardo da Vinci (1452-1519) [2]. Among his experiments he attaching a block of wood to another object with a cord, and placed the block on a smooth surface. He was able to adjust the weight of the second object, which he let hang over a fixed cylinder, free to rotate. This experiment was found in his notebooks, and is shown in figure 3.2. He experimented with tilted planes and placed the block on different sides, changing the apparent area of contact. His results made him deduce the following statements, also found in his notes:

- Friction produces double the amount of effort if the weight be doubled.
- Friction made by the same weight will be of equal resistance at the beginning of the movement though the contact may be of different breadths or lengths.

This is now essentially known as the two first laws of friction. The linear relation described in the first postulate is commonly known as the coefficient of friction

$$\mu = \frac{F}{L}, \tag{3.1}$$

though Leonardo interpreted L as the weight of the load and F as the weight of the object pulling in a scenario as in figure 3.2. The concept of force was not

commonly recognized until 200 years later. Isaac Newton's publication of *Principia* paved the way for studies of friction, there among the most comprehensive study carried out by Charles Augustin Coulomb (1736-1806). He investigated the influence of several factors on friction, namely [5]:

- The nature of the materials in contact and their surface coatings.
- The extent of the surface area.
- The normal force.
- The amount of time that the surfaces remained in contact prior to the applied force.
- Ambient conditions.

Among his results he found that the static friction force increased with the time the surfaces were in contact before applying shear force. Today this relation is known as

$$F_s(t) = A + B\ln(t). \tag{3.2}$$

Coulomb explained this behavior by regarding the materials as fibrous. Basically like a hairbrush, having fibers which get entangled into a mesh when in contact. If the materials were in contact over a short period of time, fewer fibers would have gotten into their position in the mesh. When a shear force is applied, the fibers get tilted until they loosen from the mesh, at which point sliding occurs. This impression of the sliding mechanism is quite similar to the one we have now.

3.2 Macroscopic sliding motion

3.2.1 Coefficient of friction

The coefficient of friction, μ, is a dimensionless scalar describing the ratio between the force of friction and the normal force, as given in equation (3.1). The value of μ is dependent on the material of the objects that are in contact, ambient conditions, presence of lubrication, etc.. For most materials the coefficient is higher if the objects are stationary, than if they are moving relative to each other (sliding). We may refer to them separately as the coefficient of static friction μ_s, and the coefficient of kinetic friction μ_k. Thus, $\mu_s > \mu_k$ for most materials[1].

[1] Teflon-on-teflon seems to have the same value for static- and kinetic friction.

3.2.2 Steady sliding

If we attach a spring to a block and pull on the spring with a constant velocity, we may get the evolution of friction force as shown in figure 3.3. We assume the block to have a constant normal force from the surface it rests upon. We observe that the friction force increases linearly during the loading of the spring, as expected from Hooke's law. Once the force from the spring overcomes the maximum static friction , F_s, the block begins to move (slide) and the force of friction is reduced due to the change in coefficient of friction. The coefficient of static friction is found by using the maximum static friction force, F_s, in equation (3.1). Similarly the coefficient of sliding friction is found by using the value of the friction force in the sliding domain, F_k. In this scenario, the motion of the block resulted in a steady motion. As we shall see, that is not always the case.

Keep in mind that the forces we refer to are considered to be working on the center of mass of the block. The coefficient of friction is a macroscopic observation. It wouldn't make any sense to discuss a coefficient of friction between lets say two "blocks" containing 10 atoms each. How would one even define a normal force?

Steady sliding motion can be observed for instance if a car driver pulls on the break at high speed, so that the wheels are unable to turn. The wheels will then slide along the road until the car comes to a stop. At cites of traffic accidents it is common to measure the length of skid marks, in order to approximate a lower limit of a vehicles velocity.

3.2.3 Stick-slip motion

If we pull on a body with a spring with constant and reasonably low velocity, it will be subject to a linearly increasing force (Hooke's law). Because the object does not slide until we reach the maximum static friction, it will remain its position until it does. Once the force from the spring overcomes the maximum static friction, the body will begin to slide and the coefficient of friction changes to its kinetic state. It will accelerate the most in the beginning, and continue increasing its velocity until it passes an equilibrium position. Once it passes the equilibrium position, the friction force will overcome the spring force, and the velocity will decrease. When the velocity is sufficiently low, the object will stop abruptly. As the spring is still being moved, the spring force on the object will rise again (load), and we have a complete cycle. The time evolution of the force applied by the spring will be similar to the sketch shown in figure 3.4. If the surface of the block or substrate is non-homogeneous, the stick-slip motion may be more chaotic, than illustrated by the sketch. In fact, it may be difficult to find any periodicity in the motion at all. We are not going to concern our self with chaotic stick-slip motion, since we will have perfectly smooth, dry and pure surfaces when doing molecular dynamics simulation. Stick-slip motion can be

observed for instance when a bow slides over the strings of a violin, thus making them vibrate. By changing the pressure and speed of the bow, the performer can modify the sound produced. Earth quakes is another example, though this is of the chaotic sort.

Whether the motion is of the steady type or the stick-slip type, is found experimentally to be depending on the velocity and stiffness of the spring. At sufficiently high speeds, we would always get a steady motion. Similarly, with a stiff enough spring we would also get a steady motion. While with a velocity, $v < v_s$, and stiffness $k < k_s$ we would expect a stick-slip motion. This can be illustrated by a phase diagram of velocity and spring stiffness (v, k), as in figure 3.5. The change in behavior from stick-slip to steady motion can be observed in our daily lives. The creaking of a door is due to stick-slip motion at the hinges. However, if we open the door quickly, the door will not creak.

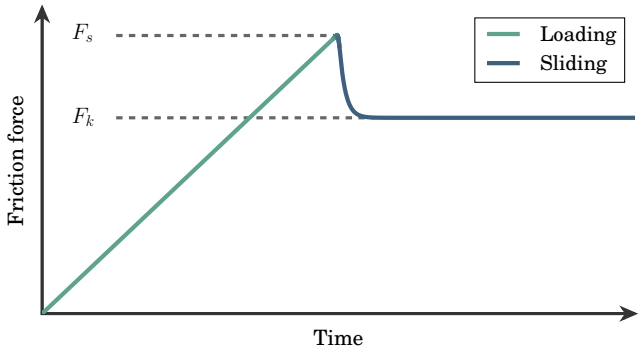


Figure 3.3: Behavior of friction force as function of time, for a block pulled at constant velocity and under constant load, resulting in a steady sliding motion.

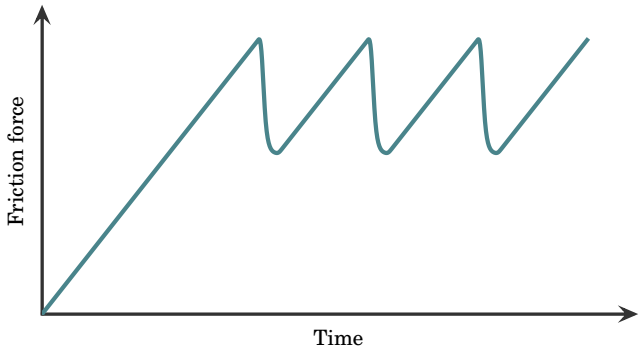


Figure 3.4: Behavior of friction force as function of time, for a block pulled at constant velocity and under constant load, resulting in a stick-slip motion.

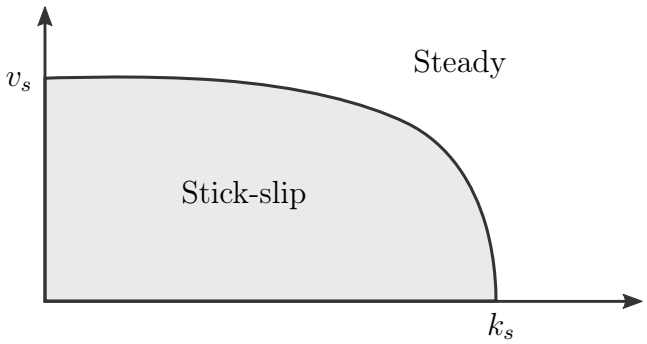


Figure 3.5: Phase diagram of velocity and spring stiffness (v, k). All combinations within the gray area will result in a stick-slip motion. All outside will result in a steady sliding motion.

3.3 Elasticity

We distinguish between elastic and plastic deformation. If the deformed body recovers its initial structure after the force causing the deformation is removed, it is referred to as an *elastic* deformation. If it does not, it is regarded a *plastic* deformation. Obviously, there is a limit to the applied stress that separates the two; the so-called yield stress. In this section we will only consider elastic deformations.

3.3.1 Strain and stress

Strain ϵ, is a measure of deformation. It represents the displacement between particles in a body relative to a reference length, and can be expressed as

$$\epsilon = \frac{\Delta l}{l_0} \tag{3.3}$$

where Δl and l are the displacement and original length, respectively. For *normal* strain, Δl represents an elongation of the distance between two facing end planes of an infinitesimal cubic element. For *shear* strain, it represents a component of an in-plane displacement of the midpoints of said plains. It is common to discern the components of stress by using two indices, e.g. ϵ_{xy}. The first index indicates that we consider the planes normal to the x-axis, while the second that we consider the shear stress in the y-direction of these planes. Double indices, such as ϵ_{xx} refer to the normal strain in the x-direction, and is commonly abbreviated to ϵ_x.

Stress σ, is a force density acting on a body due to internal forces, caused by strain. For stress on a plane, the *normal* and *shear* stress is defined as

$$\sigma_\perp = \frac{F_\perp}{A} \qquad \text{and} \qquad \sigma_\parallel = \frac{F_\parallel}{A} \tag{3.4}$$

respectively, where $F_\perp$ and $F_\parallel$ are the force components perpendicular and parallel to the plane. We use an equivalent notation for stress as for strain, with two indices distinguishing the components. This is illustrated in figure 3.6. The collection of stress and strain components are often composed into the so-called Cauchy stress tensor and strain tensor

$$\sigma = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{pmatrix} \qquad \epsilon = \begin{pmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{pmatrix} \tag{3.5}$$

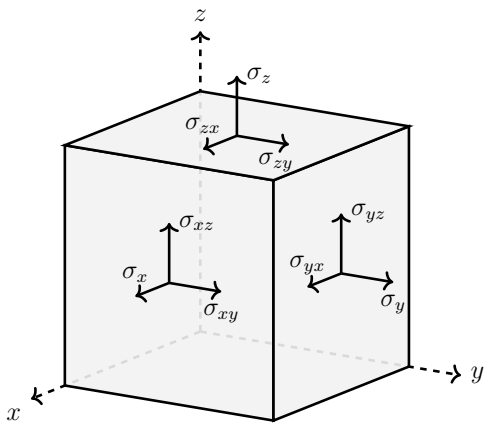


Figure 3.6: Graphical illustration of the components of the Cauchy stress tensor.

3.3.2 Hooke's law

As mentioned, stress is caused by strain. However, strain is also caused by stress. The relation between the two is known as Hooke's law. If we consider a infinitesimal cubical body subject to a tensile force in the x-direction, Hooke's law states that the relation between stress and strain is given as

$$\epsilon_x = \frac{\sigma_x}{E}, \tag{3.6}$$

where E is a material property known as Young's modulus, which express the stiffness of the material. Here we assume that the deformation is elastic.

Often the faces of the infinitesimal cubic body are subject to forces in several direction. The general form of Hooke's law is expressed as

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl} \tag{3.7}$$

which relates the Cauchy stress tensor σ_{ij} to the strain tensor ϵ_{kl}, through the linear dependence of the stiffness tensor c_{ijkl}. Note that each component of the Cauchy stress tensor is dependent on all elements of the strain tensor. The stress and strain tensors are both of rank 2 and have $3^2 = 9$ elements, while the stiffness tensor has rank 4 and contains $3^4 = 81$ elements. However, since both the Cauchy stress tensor and the strain tensor are symmetric,

$$\begin{aligned} \sigma_{ij} = \sigma_{ji} \Rightarrow C_{ijkl} = C_{jikl} \\ \epsilon_{kl} = \epsilon_{lk} \Rightarrow C_{ijkl} = C_{ijlk} \end{aligned} \tag{3.8}$$

In total there will be only 6 independent components for the stress and strain tensors, and 36 in the stiffness tensor. We may choose to express the stress and strain as vectors by using Voigts method for rank reduction [6].

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{xz} \\ \sigma_{yz} \end{pmatrix} \equiv \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ 2\epsilon_{xy} \\ 2\epsilon_{xz} \\ 2\epsilon_{yz} \end{pmatrix} \equiv \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \end{pmatrix} \tag{3.9}$$

Similarly the stiffness tensor reduces to a 6×6 matrix (rank 2)

$$\mathbf{c} = \begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} & C_{46} \\ C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{56} \\ C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66} \end{pmatrix}. \tag{3.10}$$

Hooke's law can then be expressed as a matrix product

$$\boldsymbol{\sigma} = \mathbf{C}\boldsymbol{\epsilon} \tag{3.11}$$

3.4 Hertz' theory

3.5 Modern knowlage

Today we know that most surfaces are rough, at least at a microscopic level, having some degree of asperities. It is actually the asperities that are in contact with the other body. When there is contact between two surfaces, the asperities are pushed into the other material, and tries to occupy the position with the lowest potential. The asperities may move even though the macroscopic body is still. This is known as thermally activated creep. Thus, longer duration of resting contact gives a higher rate of asperities that settle in positions with the lowest potential, effectively making the coupling of the bodies stronger.

Chapter 4

LAMMPS

LAMMPS is an acronym for *Large-scale Atomic/Molecular Massively Parallel Simulator*. It is a classical molecular dynamics simulation code designed to run efficiently on parallel computers. It's development began in the mid 1990s at Sandia National Laboratories, with funding from the U.S. Department of Energy. It was a cooperative project between two DOE labs and three private companies. The development is still ongoing and contributions are revised thoroughly. Today LAMMPS is an open-source code with extensive and user friendly documentation. This is one of the main reasons why we have chosen to use LAMMPS as opposed to other molecular dynamics software.

In this chapter we will be presented with a guide to install LAMMPS, a brief explanation of commands and syntax, and an example of a valid input script.

4.1 Installation

Installing LAMMPS is a fairly simple procedure if only the basic settings are needed.

4.1.1 Linux

Users with a Unix based OS may download the lammps distribution as a tarball from LAMMPS' download page[1] and then unpack it from the command line.

```
gunzip filename.tar.gz
tar xzvf filename.tar
```

The user should then change directory into `/path/to/lammps/src/`, and execute the following commands in order to list available packages.

```
make package-status
```

[1] http://lammps.sandia.gov/download.html

Installing specific packages is accomplished as shown below.

```
1 make yes-molecule yes-manybody yes-python yes-rigid
```

The above example installs the packages *molecule, manybody, python and rigid*. Next, the user can build LAMMPS using either of the lines below. Assuming the user has MPI installed, line 2 utilizes 8 processors to make the resulting executable compatible with parallelization in MPI. The argument `-j8` is optional, but saves time.

```
1 make serial
2 make -j8 mpi
```

At this point there should be an executable in the `/path/to/lammps/src/` directory named `lmp_serial` or `lmp_mpi`, depending on the previous choice. These are now ready to run. To use it one has to point to this file from the command line at every run. It may be practical to set up a symlink as shown below.

```
1 sudo ln -s /path/to/lammps/src/lmp_mpi
     /usr/local/bin/lmp_mpi
```

Then the executable will be available as `lmp_serial` or `lmp_mpi` from any directory.

4.1.2 Mac OS X with Homebrew

Mac users can follow the procedure described above, however they may also install even easier using *Homebrew*[2].

```
1 brew tap homebrew/science
2 brew install lammps              # serial version
3 brew install lammps --with-mpi   # mpi support
```

Where the user obviously should choose either line 2 or line 3, depending on whether the user wants MPI comparability. This will install an executable named "lammps", a python module named "lammps", and resources with standard packages. This is basically it. LAMMPS is now ready to run, however, not all packages are installed. The location of the resources and available packages can be found using the following command.

```
1 brew info lammps
```

Specific packages are available as options, and may be installed using the following syntax.

```
1 brew install lammps --enable-manybody
```

[2] http://brew.sh/

In the example shown we installed the package manybody.

4.2 Input scripts

The executable made in the previous section can be used to read so-called input scripts. The input scripts contain LAMMPS commands to configure the simulation. This includes all settings and actions. This is naturally partitioned into two sections: *system configurations* and *run-time commands*. The scripts are executed on 8 CPU's in parallel as

```
mpirun -n 8 lmp_mpi -in in.system
```

Despite the widely common convention of letting the file type be determined by the letters following the punctuation mark, it's common in the LAMMPS manuals to name the input scripts with the name last. i.e. `in.name`. Luckily, both conventions are meretricious since the executable accepts any name/type for the input script.

4.2.1 System configurations

The first part of every input script contains information of the system properties. This includes: size, number of dimensions, boundary conditions, potentials, unit convention, information on the containing atoms, time step length, neighbor list update frequency and possibly a lot more. Listing 4.1 show the configurations that has been used for the entire duration of this project. Below there is a description of each command and its purpose.

```
units         metal
boundary      p p p
atom_style    atomic
read_data     "system.data"
pair_style    vashishta
neighbor      0.3 bin
neigh_modify  delay 10
timestep      0.002
```

Listing 4.1: Typical system configurations applied in this project.

`units` determines what unit convention should be used for the simulation. It might not seem like a difficult decision, but if you import data from a file for instance, it is crucial that LAMMPS interpret the data correctly. In this project we have chosen *metal* as the unit convention. The units are therefore as shown in table B.1.

`boundary` sets the style of boundaries for the simulation box in each dimension. There are four options: `p`, `f`, `s` and `m`. In the example above, we have chosen to use periodic boundaries through all the faces of the simulation box. It's possible to set different conditions in separate dimensions, e.g. `boundary p f p` sets fixed boundaries on the faces normal to the y-direction. One can even set different conditions on the two faces in the same dimension by using two letters, e.g. `boundary p fs p`, where the first letter indicates the boundary to be used on the *low face* and the last on the *high face*.

`atom_style` tells LAMMPS the structure of atom related data stored in a data file. This includes information about particle types, positions, charges, mass, bonds, angles, and potentially more, depending on the `atom_style`.

`read_data` reads a data file containing information as described above and additional information about the system, such as its size and shape. This is one of three ways to distribute initial atom positions. Another is the `read_restart` command, which is used extensively to load saved states from restart files. The last option is to use `create atoms`, which distributes atoms in a predefined way, i.e. in a lattice or a random collection. To use this last option, one has to first create a simulation box using the `create box` command.

`pair_style` determines which potential to use in the simulation.

`neighbor` sets the additional range of the neighbor list cutoff, and the method of constructing the neighbor lists. The consequence of having a larger range for the neighbor lists is that there are more particles to check for force contribution, however, we don't have to update the neighbor lists as often.

`neigh_modify` determines how often to update the neighbor lists.

`timestep` self-explanatory.

4.2.2 Run-time commands

Variables

LAMMPS offer the ability to store values in variables. These can be constants or dependent on other variables, like the time step for instance. Declaration of variables is done using the following syntax.

```
1 variable A equal 1000
2 variable B equal step/${A}
3 variable C equal ${B}
```

```
4 variable D equal v_B
```

Listing 4.2: Declaration of variables.

All of the above are valid variables. The `step` variable is a LAMMPS standard for current time step. We attend for `B` to be a linear function running from 0 to 1 over the course of `A` time steps. `C` evaluates `B` at the time step it is declared and returns that value whenever called upon. Thus, `C` will be a constant value. `D` will evaluate `B` whenever called upon, and return the current value of variable `B`. The variables `C` and `D` are of course superfluous, since we could only use `B` directly. The user can use math operations on variables, such as: addition, subtraction, multiplication, division, as well as square roots, logarithms, exponentials, and lots more.

Region & Group

In almost all simulations it is necessary to define regions and groups in order to give certain parts of the domain special properties. If the user wish to create a sphere of atoms, the typical process is to start from a block of atoms, define a spherical region in the block and remove exterior atoms. The regions specified can take the shapes: block, cone, cylinder, plane, prism or sphere, and regions can be combined. An example of removing atoms within a combined region is shown in listing 5.3. This example also show how to assign a group to atoms within a region. Each atom can be part of up to 32 different groups at a time. The user can do tons of actions on groups. Most computes and fixes act on specific groups. There is only one standard group in LAMMPS; the group `all`.

Computes

A compute defines a computation that will be performed on a group of atoms. This does not affect the dynamics in any way. The returned values are instantaneous. That is, they are computed from information about atoms on the current time step. The returned values can be global, local or per-atom quantities. These can be retrieved by an output commands using the following syntax:

```
1 c_computeID
2 c_computeID[1]
3 c_computeID[*][2]
```

The first alternative would return all values (scalars, vectors, arrays) computed. The second would return only the first element of a vector, or row of an array, and the last one would return the second element of all rows of an array. Note that LAMMPS counts from 1, which also deviate from other established standards. In section 4.2.3 we will see this in several example, where we refer to the `compute com` command.

```
compute comID groupID com
```

This computes the position of the center of mass of all atoms within the specified group, and stores the result in a 3-element vector.

Fix

A fix is an operation that's applied to the system during time iteration. Examples include time evolution of atoms, i.e. updating their positions and velocities, using thermostats, applying external force on atoms, averaging values, etc. There are numerous fixes and computes in LAMMPS, and you can easily add new ones if you knows the structure of the framework. Some Fixes also compute and store values that can be retrieved by output commands. For instance the `fix addforce` command

```
fix addforceID groupID addforce fx fy fz
```

adds a given force to all atoms within the specified group. This `fix` stores the total force acting on the group before the force was added. This is stored as a 3-element vector. Its values can be obtained in the same manner as for computes:

```
f_fixID
f_fixID[1]
f_fixID[*][2]
```

4.2.3 Output

In order to benefit from our simulations, we need to be able to extract the data of interest somehow. Here we will present the four most basic types of output.

Thermodynamic output

prints computed values to the screen and logfile every N time steps. It is activated by the command

```
thermo N
```

where N should be replaced by the desired frequency of the output. This can be a variable. The default quantities given in the output are: time step, temperature, pairwise energy, molecular energy, total energy and pressure. The user can specify what values should be included by using the `thermo_style` command with the syntax of the following example.

```
thermo_style custom step c_comID v_Fz f_addforceID[3]
```

This line specifies that when the thermodynamic output is printed, it should contain the time step, all quantities returned by the compute `comID`, the current value of the variable `Fz` and the third element of the vector retrieved by the fix `addforceID`. The naming of the compute- and fix names can be whatever desired, even numbers. I just find it convenient to name them by what they represent.

Dump

Dump files store data about the state of atoms in a specified group with the given frequency. Below we illustrate two quite different dump commands.

```
1 dump dumpID1 all atom 100 name.dump
2 dump dumpID2 groupID custom 50 name*.dump id x y vx fx
```

Line 1 creates a file named `name.dump`. Every 100 time steps it writes data to this file in the *atom* format, i.e. `id type xs ys zs`, where `xs`, `ys` and `zs` are scaled coordinates relative to the size of the simulation box. Line 2 creates a file for every time step, where the asterisk in the name is replaced by the current time step. Also we define the format of the output to be unscaled x- and y-coordinates, as well as velocity and force in the x-direction.

Common visualization software can read most of the predefined formats in LAMMPS: atom, xyz, cfg, etc..

Fix

Certain fixes can also write specified quantities to files. For instance the commands

```
1 fix timeAvgID all ave/time 100 5 1000 c_comID file name.avg
2 fix spatialAvgID all ave/chunk 100 10 1000 chunkID c_comID
     file vel.profile
```

do respectfully time- and spatial averaging of the values returned by the compute `comID`. The numbers indicate how many values should be sampled, number of time steps between each sample, and how often to write this average to file. Note that the last command refers to a `compute chunk`, which governs how to grid the system. There is also a `fix print` command that writes single-line output to screen or file with a prescribed frequency.

```
1 fix printID all print 100 "z-component of COM: c_comID[3]"
```

This acts practically like the `thermo` command.

Restart files

In many cases it is practical to save certain checkpoints in a simulation, in order to be able to start a new simulation from said point. The `restart` command does just that.

```
1 restart N name.*.restart
```

This command saves the state of the system every `N` time step to individual restart files, distinguishable by the number representing the time step of the save. By "state of the system" we mean positions and velocities of all atoms, information of what groups each atom belongs to, the size and shape of the simulation box, boundary conditions, potential, etc.. When starting a new simulation, one of these "snapshots" can be loaded using the `read_restart` command as illustrated below.

```
1 read_restart name.timeStep.restart
```

4.3 Visualization

The LAMMPS package does not provide high-quality visualization software. However, the default formats of the dump command are well known to most of the ones out there. A couple of so-called high-quality visualization tools are listed below:

- VMD
- AtomEye
- OVITO
- ParaView
- PyMol

In this section we will discuss hos to visualize an MD simulation and some of the features of one of the softwares listed, namely Ovito. Lastly we will be presented with a new concept that is currently under development here at the University of Oslo; a software named Atomify.

4.3.1 OVITO

Chapter 5

Preparing a molecular dynamics simulation

We wish to construct a system consisting of two elements made out of silica: a slab and a sphere cap. In order to do this we need to generate the spacial position coordinates (x,y,z) of every single atom. Considering that we are making a system consisting of about 10^5 atoms, this is obviously not done manually. We have chosen to use a tool named *Moltemplate*[1], which is included in the LAMMPS distribution.

The main idea is to manually enter the coordinates of only the atoms in a unit cell of the material one wish to generate, and then simply copy this unit cell wherever desired. The software will shift the coordinates of the copied unit cell by the displacement from the original image. In addition it will generate files containing data such as which atoms they share bonds with, if any, and angles between such bonds.

5.1 Silica

Silica is a chemical compound also known as Silicon dioxide, having the chemical formula SiO_2. It has several polymorph structures, the most common being quartz, which is one of the most abundant minerals in the Earth's crust. Other polymorphs include cristobalite, tridymite, coesite and more. For our purpose it is insignificant which one we choose. Once the material is melted, it is indifferent which configuration we started from, as long as the density is correct. In this project we will build the constituents of the system from a type of cristobalite named β-cristobalite. This is mainly because it has a simple structure and a cubical unit cell.

[1] http://www.moltemplate.org/index.html

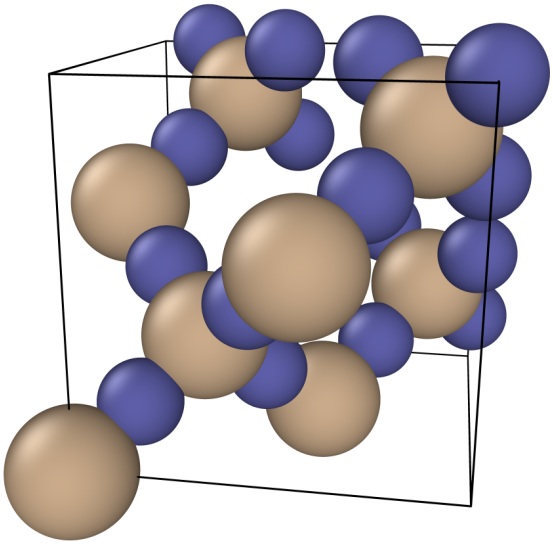


Figure 5.1: Unit cell of b-cristobalite. Tan and blue spheres represent silicon and oxygen atoms respectively. The unit cell is cubical, with edges of length 7.12Å.

5.1.1 Unit cell of β-cristobalite

In order to construct the unit cell of a material, one should look up the coordinates of the atoms in a crystallography database. We have used the unit cell of β-cristobalite found at *Crystallography Open Database*[2]. At this site one can download a `.cif`-file (Crystallographic Information Framework) containing information about the spatial positions of each atom, the length of the unit cell edges and angles between faces of the cell. In the case of β-cristobalite the unit cell is cubical with edges of length 7.12Å. It contain 8 silicon atoms and 16 oxygen atoms. The density of the unit cell can easily be computed and is 2.2114 g/cm^3. The format of th `.cif`-file is not easily readable. To extract the information we have used a tool named `cif2file`[3], developed by Torbjörn Björkman.

5.2 Building a crystal

The coordinates gotten from the `.cif`-file can now be implemented into *moltemplate* together with whatever bond and angle data required by the potential. In our simulations we will use the Vashishta potential, which does not require these.

[2] http://www.crystallography.net/cod/1010944.html
[3] http://www.cif2cell.com-about.com/

Moltemplate has its own structure and syntax. The first step to build up a larger material is, as mentioned, to create the unit cell. Data concerning the unit cell are placed in a `.lt`-file, which is readable by Moltemplate. Such a file is shown in Listing 5.1. For a more profound understanding of the structure and syntax of these files, the reader is advised to read the moltemplate manual[4].

```
# file "beta-cristobalite.lt"

beta-cristobalite {
  write("Data Atoms") {
    $atom:Si1    @atom:Si    0.00    0.00    0.00
    $atom:Si2    @atom:Si    0.00    3.56    3.56
    $atom:Si3    @atom:Si    1.78    1.78    1.78
    $atom:Si4    @atom:Si    3.56    0.00    3.56
    $atom:Si5    @atom:Si    1.78    5.34    5.34
    $atom:Si6    @atom:Si    5.34    5.34    1.78
    $atom:Si7    @atom:Si    3.56    3.56    0.00
    $atom:Si8    @atom:Si    5.34    1.78    5.34
    $atom:O1     @atom:O     0.89    0.89    0.89
    $atom:O2     @atom:O     6.23    4.45    2.67
    $atom:O3     @atom:O     2.67    2.67    0.89
    $atom:O4     @atom:O     4.45    0.89    4.45
    $atom:O5     @atom:O     0.89    4.45    4.45
    $atom:O6     @atom:O     4.45    4.45    0.89
    $atom:O7     @atom:O     2.67    6.23    4.45
    $atom:O8     @atom:O     2.67    0.89    2.67
    $atom:O9     @atom:O     4.45    2.67    6.23
    $atom:O10    @atom:O     6.23    2.67    4.45
    $atom:O11    @atom:O     2.67    4.45    6.23
    $atom:O12    @atom:O     0.89    6.23    6.23
    $atom:O13    @atom:O     0.89    2.67    2.67
    $atom:O14    @atom:O     4.45    6.23    2.67
    $atom:O15    @atom:O     6.23    6.23    0.89
    $atom:O16    @atom:O     6.23    0.89    6.23
  }

  write_once("Data Masses") {
    @atom:Si 28.0855
    @atom:O  15.9994
  }
} # end definition of beta-cristobalite molecule type
```

Listing 5.1: Typical Moltemplate file containing unit cell data. The columns of the "Data Atoms" section hold, from left to right, information of atom ID, atom type, x-, y- and z-position. The "Data Masses" section stores the weight of silicon and oxygen atoms in atomic mass units.

We use the unit cell as building blocks, placing them concurrently until we

[4] http://www.moltemplate.org/doc/moltemplate_manual.pdf

have a crystal of the desired size. For example purposes, we generate a large cube of $15 \times 15 \times 15$ unit cells. This is done in line 17-21 of listing **??** below. Uncomment the above listing! The *write_once* sections with "In" as the first word of its argument, creates the files `system.in.init` and `system.in.settings`, which contain exactly what is shown in the snippet. They are not necessary, but help making the LAMMPS input script cleaner, since we can `import` these files instead of having all the configurations in the input script file. The last *write_once* section with "Data" in its argument, incorporates the specified size of the simulation box in the `system.data` file. Line 19 creates a new unit cell at every point separated by 7.12Å in all three dimensions. The positions, and other properties defined by the *atomstyle*, of all the distributed atoms are stored in the file `system.data` as well.

The Moltemplate script can be run from the command line as

```
1  moltemplate -atomstyle "atomic" system.lt
```

Once complete, we can load the configurations and atomic data into our LAMMPS input script simply by including the file `system.in`.

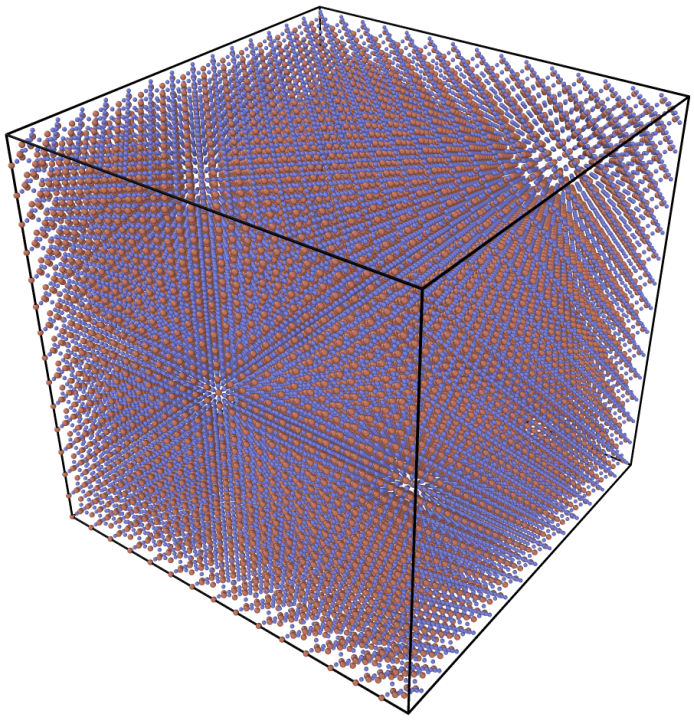


Figure 5.2: System built from $15 \times 15 \times 15$ unit cells of b-cristobalite.

5.3 Verifications

5.3.1 Melting point

Finding the melting point is a very important verification of our system. There are several factors that may affect the result. For instance, if the density of the system is too high, the melting point will be at a higher temperature than it normally would. Also, errors in the potential model may affect the temperature of the melting point. When increasing the temperature of the system that is in a solid state, we will eventually reach the melting point. At the phase transition from a solid state to a liquid, the atoms of the silica will have energy great enough to break the interatomic bonds. They will break loose from their regular arrangement and move about much more freely. An approach to computing the melting point is therefore to systematically increase the temperature stepwise and sample the mean square displacement of the atoms at each temperature level. The mean square displacement is the average of the square of the displacement every atom has from its initial position. It can be expressed as:

$$\langle r^2(t) \rangle = \frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{r}_i(t) - \mathbf{r}_i(0) \right)^2 . \tag{5.1}$$

where $\mathbf{r}_i(t)$ is the position of atom i at time t and N is the total number of atoms. In practice the mean square displacement was computed using a standard *compute* in LAMMPS, namely the *msd compute*[5]. At every time step it stores a vector of 4 elements; the first 3 are the squared dx, dy and dz displacements averaged over the atoms of the specific group, while the 4th is the total mean square displacement for the specific group, i.e. $(dx^2 + dy^2 + dz^2)$. The $15 \times 15 \times 15$ system is sufficient for this test. The procedure is rather simple. For β-cristobalite we expect the melting point to be about 1986K[6], so we start out at 1500K. After equilibration the following steps are repeated until we reach a target temperature.

- equilibrate for the current temperature
- compute the msd for N time steps
- increase the temperature by a given step size

```
label meltingLoop
variable i loop 11
    fix nvtID all nvt temp ${T} ${T} 1.0
    run ${N}
    compute msdID all msd
```

[5] http://lammps.sandia.gov/doc/compute_msd.html
[6] https://en.wikipedia.org/wiki/Cristobalite

```
6       fix msdDumpID all ave/time 1 1 3 c_msdID[4] file
            msd_N${N}_T${T}.txt
7       run ${N}
8       uncompute msdID
9       unfix nvtID
10      unfix msdDumpID
11      variable T equal ${T}+50
12  next i
13  jump SELF meltingLoop
```

Listing 5.2: Section of LAMMPS script illustration how to stepwise increase temperature and compute the mean square displacement of atoms.

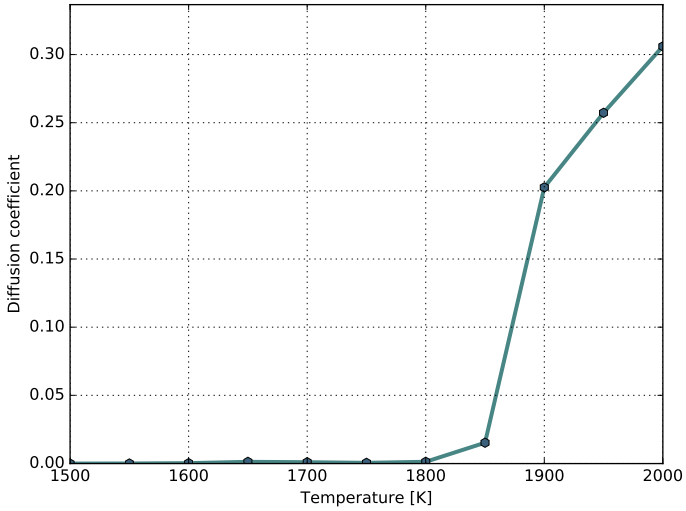


Figure 5.3: Mean square displacement as a function of temperature. The face transition occurs where we see a rapid change in the behavior.

This result infer a melting point somewhere in the range 1800K-1900K. Believe it or not, this is actually a pass on the test! In fact, anything deviating less than 20% from the experimental results is considered a pass, as stated by the developers of the potential. Unfortunately, this also show that there are some qualitative detail that is lost, and that when using this potential we should probably focus on the quantitative behavior.

5.4 Shaping the system

The huge cube of silica can be carved however we like by defining regions from which we delete the containing atoms. In LAMMPS this is done using the `region`, `union`, `intersect` and `delete_atoms` commands. Our implementation is stated

in Listing 5.3, which is very simple due to the way we are going to treat the boundary conditions. We start out by defining a spherical region labeled `sphereRegion`, described by the xyz-coordinates of its center and a radii. The atoms within this region are assigned to a group labeled `sphereGroup`. Next, we define a cuboid (block) region named `slabRegion`, described by the position of its faces in x-, y- and z-direction. The atoms within this region are assigned to a group, which we label `slabGroup`. We combine these two regions using the `union` command and label the region outside of these regions `outRegion`. Finally, we delete the atoms that are not in the sphere nor the slab; we delete the ones contained by `outRegion`.

```
region sphereRegion sphere 53.4 53.4 226.8 150
group  sphereGroup region sphereRegion

region slabRegion block 0 INF 0 INF 0 35.6
group  slabGroup region slabRegion

region outRegion union 2 sphereRegion slabRegion side out
delete_atoms region outRegion
```

Listing 5.3: Defining regions to keep or delete from a system of dimensions $106.8 \times 106.8 \times 106.8$ Å.

For the purpose of deleting atoms, the creation of groups was superfluous. However, at a later stage we will utilize them and this is an appropriate place for them to be assigned. The appliance of the script in Listing 5.3 on the system is shown in figure 5.2 is shown in figure 5.4, where our perspective is along the y-axis.

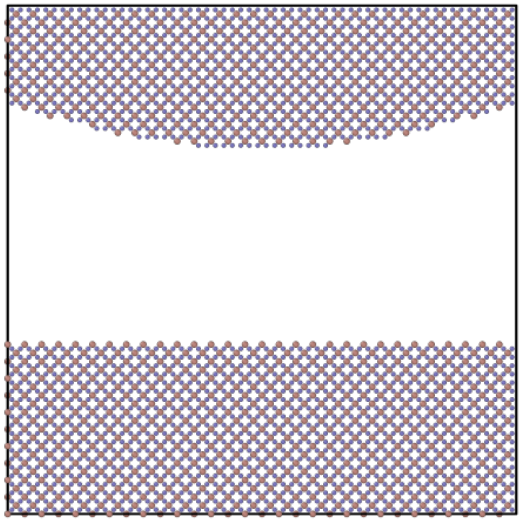


Figure 5.4: xz-perspective on a system built from $15 \times 15 \times 15$ unit cells of b-cristobalite, with certain regions carves out. This is a result from applying Listing 5.3 to the system shown in figure 5.2. The top shape is a sphere cap, while the bottom is a slab.

5.5 Moving the sphere towards the slab

We wish to push the sphere down onto the slab in order to create a deformation on the slab. There are many approaches to accomplish this. One could apply an external force, pushing the sphere with a controlled force. Instead, I have chosen to move the sphere at a controlled, constant velocity. We have settled on the following strategy: We apply periodic boundary conditions in all three dimensions. Secondly, we freeze the atoms at the bottom of the slab so that their positions are fixed. They still interact with other atoms, but we restrain them from moving. This will, due to the periodic boundary conditions, allow us to consider the sphere cap and the slab as if they are not connected to each other, but to independent blocks of silica glass. For every N time steps we decrease the hight of the system, while remapping the positions of the atoms. The remapping is a very important procedure. It ensures that we do not loose any atoms that elsewise would be lost when moving the z-boundary. A side-effect of the remapping is that the atoms in the system will be somewhat compressed in the z-direction. Though, if we do the compression slowly, this will not be of any concern.

A technique to fix atom positions is to explicitly set their velocity and force to zero at every single time step. A much smarter method is to simply omit the atoms of consideration from time-integration. This way we do not bother computing the force acting on them, updating their velocity or position at all! To do this, we simply create a group that contains all atoms except the once who

should be fixed. Listing 5.4 show how to tell LAMMPS to only time-integrate the atoms in the group `excludingBT`, which excludes the bottom and tom regions of our system. The omitted atoms are still considered to be present, we just don't bother updating their movement.

To deform/resize the system, we may use the command shown in listing 5.5. This command changes the position of the upper boundary of in the z-dimension, by the length `compressionLength` during a `run`. Thus if we choose the run to last N time steps, the boundary will move with the velocity

$$\mathbf{v}_z = -\frac{\texttt{compressionLength}}{N \cdot 0.002} \quad [\text{Å/ps}] \tag{5.2}$$

since our time steps are 0.002 ps.

```
1 fix nvtID excludeBT nvt temp ${T} ${T} 1.0
```

Listing 5.4: Time-integrating only atoms in a specified group, `excludeBT`, effectively fixing all others.

```
1 fix ID all deform 1 z delta 0 -${compressionLength} remap x
```

Listing 5.5: LAMMPS command for deforming the simulation box, and remaping atom positions.

Part II

Simulations

Chapter 6

Computing the normal force distribution

The normal force is defined as the force exerted on an object that is perpendicular to the contact surface. In this chapter we will make an attempt to find the distribution of the normal forces. This is not a trivial thing to compute in LAMMPS. As a matter of fact, to achieve this we have expanded the LAMMPS library by creating a custom compute class. For the sake of completeness, the details of that procedure will be described.

Our strategy is simple, but not necessarily easy. First of, we divide the system into a 2D-grid in the xy-plane. Secondly, we compute the average force exerted on one body from another within each cell of the grid. We approximate the slope of the contact surface within the cells using a least squares regression method on the atoms positioned at the surface. Finally, we project the average force of the atoms in a cell onto the normal vector of the approximated surface.

6.1 Creating a custom compute

A *compute* is a LAMMPS command that defines a computation that will be performed on a group of atoms. The *computes* produce instantaneous values, using information about the atoms on the current time step[1]. In LAMMPS there are more than 100 computes and chances are, they already have what you're looking for. If not, one might treat the data from other computes in some way to get the desired information. However, if there are no compute command that does the desired task, it is possible to create an own custom class and thereby expanding the LAMMPS library.

In order to compute the normal forces acting on the sphere, we have written a custom compute class. The objective was for the class to save the force acting on each atom in one group from atoms of another group.

[1] http://lammps.sandia.gov/doc/compute.html

In this section we will try do give a brief summary on how this was done. [MAYBE AN ILLUSTRATIVE FIGURE HERE]

6.1.1 Look for similar computes

Obviously, before writing any code we should know what we want the compute to calculate and how this should be done. Before starting off with a blank sheet in the editor, one should definitely search for similar computes in LAMMPS. This can potentially save hours of hard work! Our case serves as a good example. There is a compute named *group/group*[2] which computes the total energy and force interaction between two groups of atoms. This is almost what we want, but we need to know the force acting on each atom from atoms of other groups. Also, It should work with the Vashishta potential, which *group/group* currently does not. Thus, there are minor modifications needed and because of the similarities we chose to make our compute a subclass of this one.

6.1.2 Creating the class

All computes in LAMMPS are subclasses of the class named *compute*. From this superclass they inherit a bunch of variables, functions and flags, which the user may decide to set. Functions are of course declared in the header file, while variables and flags are set in the source file. The source code of the *group/group* compute is shown in Appendix A.1. Since we will be making a subclass of it, we change the *private* property to *protected* so that we have access to all the variables and functions, from out subclass.

We start out by creating a header file and decide upon a name for our class. We have chosen the name *group/group/atom* since it is basically a per-atom version of the already existing compute *group/group*. A trailing *"/atom"* is the common naming of per-atom computes. A complete header-file is shown in Listing 6.2 and explained in detail below.

```
1  #ifdef COMPUTE_CLASS
2  ComputeStyle(group/group/atom,ComputeGroupGroupAtom)
3  #else
4
5  #ifndef LMP_COMPUTE_GROUP_GROUP_ATOM_H
6  #define LMP_COMPUTE_GROUP_GROUP_ATOM_H
7
8  #include "compute.h"
9  #include "compute_group_group.h"
10
11 namespace LAMMPS_NS {
12
13 class ComputeGroupGroupAtom : public ComputeGroupGroup {
```

[2] http://lammps.sandia.gov/doc/compute_group_group.html

```
14   public:
15    ComputeGroupGroupAtom(class LAMMPS *, int, char **);
16    ~ComputeGroupGroupAtom();
17    void compute_peratom() override;
18    int nmax;
19    double **carray;
20
21   private:
22    void pair_contribution() override;
23  };
24  }
25  #endif
26  #endif
```

Listing 6.1: Header file of our new compute: `compute_group_group_atom.h`.

As can be seen, there are not many additions to the variables and functions already existing in the superclass, as the main difference lies in the structure we store the computed values.

`ComputeStyle` defines the command to be used in the input script to be `group/group/atom`, and the name to be `ComputeGroupGroupAtom`. The name will be redundant to us.

`nmax` is the number of atoms which are subject to a non zero force from atoms of another group at the current time step; it may vary.

`carray` is a two dimensional array containing the force on atoms in one group induced by atoms of another group. Its dimension will necessarily be `nmax` $\times$ 3.

`compute_peratom()` and `pair_contribution()` are functions which will be described below the corresponding source file.

```
1  #include <mpi.h>
2  #include <string.h>
3  #include "compute_group_group_atom.h"
4  #include "atom.h"
5  #include "update.h"
6  #include "force.h"
7  #include "pair.h"
8  #include "neighbor.h"
9  #include "neigh_request.h"
10 #include "neigh_list.h"
11 #include "group.h"
12 #include "kspace.h"
13 #include "error.h"
14 #include <math.h>
15 #include "comm.h"
16 #include "domain.h"
17 #include "math_const.h"
18 #include "memory.h"
19
20 #include <iostream>
```

```
21 using namespace LAMMPS_NS;
22 using namespace MathConst;
23
24 #define SMALL 0.00001
25
26 ComputeGroupGroupAtom::ComputeGroupGroupAtom(LAMMPS *lmp,
       int narg, char **arg) :
27     ComputeGroupGroup(lmp, narg, arg),
28     carray(NULL),
29     nmax(0)
30 {
31     if (narg < 4) error->all(FLERR,"Illegal compute
          group/group command");
32
33     peratom_flag      = 1;  // Indicating a peratom compute
34     size_peratom_cols = 4;  // # of Columns per atom.
35     extarray          = 0;  // 0/1 if global array is all
          intensive/extensive
36     scalar_flag       = 0;
37     vector_flag       = 0;
38 }
39
40
41 ComputeGroupGroupAtom::~ComputeGroupGroupAtom()
42 {
43     memory->destroy(carray);
44 }
45
46
47 void ComputeGroupGroupAtom::compute_peratom()
48 {
49     // grow array if necessary
50     if (atom->nmax > nmax) {
51
52         memory->destroy(carray);
53         nmax = atom->nmax;
54         memory->create(carray, nmax, size_peratom_cols,
              "group/group/atom:carray");
55         array_atom = carray;
56
57     }
58
59     if (pairflag) pair_contribution();
60     if (kspaceflag) kspace_contribution(); // This doesn't
          happen though. See compute_group_group.cpp
          constructor.
61 }
62
63
64 void ComputeGroupGroupAtom::pair_contribution()
65 {
```

```
66      int i,j,ii,jj,inum,jnum,itype,jtype;
67      double xtmp,ytmp,ztmp,delx,dely,delz;
68      double rsq,eng,fpair,factor_coul,factor_lj;
69      int *ilist,*jlist,*numneigh,**firstneigh;
70
71      double **x = atom->x;
72      int *type = atom->type;
73      int *mask = atom->mask;
74      int nlocal = atom->nlocal;
75      double *special_coul = force->special_coul;
76      double *special_lj = force->special_lj;
77      int newton_pair = force->newton_pair;
78      double *columns;
79
80      // invoke half neighbor list (will copy or build if
            necessary)
81
82      neighbor->build_one(list);
83
84      inum = list->inum;
85      ilist = list->ilist;
86      numneigh = list->numneigh;
87      firstneigh = list->firstneigh;
88
89      // loop over neighbors of my atoms
90      // skip if I,J are not in 2 groups
91
92
93      for (ii = 0; ii < inum; ii++) {
94          i = ilist[ii];
95
96          array_atom[i][0] = 0;
97          array_atom[i][1] = 0;
98          array_atom[i][2] = 0;
99          array_atom[i][3] = 0;
100     }
101
102     for (ii = 0; ii < inum; ii++) {
103         i = ilist[ii];
104
105         // skip if atom I is not in either group
106         if (!(mask[i] & groupbit || mask[i] & jgroupbit))
                continue;
107
108         xtmp = x[i][0];
109         ytmp = x[i][1];
110         ztmp = x[i][2];
111         itype = type[i];
112         jlist = firstneigh[i];
113         jnum = numneigh[i];
114
```

```
115           for (jj = 0; jj < jnum; jj++) {
116               j = jlist[jj];
117               factor_lj = special_lj[sbmask(j)];
118               factor_coul = special_coul[sbmask(j)];
119               j &= NEIGHMASK;
120
121               // skip if atom J is not in either group
122               if (!(mask[j] & groupbit || mask[j] &
                      jgroupbit)) continue;
123
124               int ij_flag = 0;
125               int ji_flag = 0;
126               if (mask[i] & groupbit && mask[j] & jgroupbit)
                      ij_flag = 1;
127               if (mask[j] & groupbit && mask[i] & jgroupbit)
                      ji_flag = 1;
128
129               // skip if atoms I,J are only in the same group
130               if (!ij_flag && !ji_flag) continue;
131
132               delx = xtmp - x[j][0];
133               dely = ytmp - x[j][1];
134               delz = ztmp - x[j][2];
135               rsq = delx*delx + dely*dely + delz*delz;
136               jtype = type[j];
137
138               if (rsq < cutsq[itype][jtype]) {
139                   eng = pair->single(i, j, itype, jtype,
                          rsq, factor_coul, factor_lj, fpair);
140
141                   // energy only computed once so tally full
                          amount
142                   // force tally is jgroup acting on igroup
143
144                   if (newton_pair || j < nlocal) {
145                       array_atom[i][0] += eng;
146                       if (ij_flag) {
147                           array_atom[i][1] += delx*fpair;
148                           array_atom[i][2] += dely*fpair;
149                           array_atom[i][3] += delz*fpair;
150                       }
151                       if (ji_flag) {
152                           array_atom[j][1] -= delx*fpair;
153                           array_atom[j][2] -= dely*fpair;
154                           array_atom[j][3] -= delz*fpair;
155                       }
156
157                       // energy computed twice so tally half
                              amount
158                       // only tally force if I own igroup
                              atom
```

```
159                     }
160                     else {
161                         array_atom[i][0] += 0.5*eng;
162                         if (ij_flag) {
163                             array_atom[i][1] += delx*fpair;
164                             array_atom[i][2] += dely*fpair;
165                             array_atom[i][3] += delz*fpair;
166                         }
167                     }
168                 }
169             }
170         }
171 }
```

Listing 6.2: Source file of compute: `compute_group_group_atom.cpp`.

Setting flags

In the constructor we set specific flags that LAMMPS uses to interpret what structure our data should have, and how to store them. We set the `peratom_flag` to be `True`, which indicates that we desire to store some data for each atom. `size_peratom_cols` defines the number of data values to store for each atom. The flag `extarray` is set to `False`, indicating that the per-atom value is an intensive value. Also, we set the `scalar_flag` and `vector_flag` to `False`, since we do not wish to return a vector or scalar value.

Destructor

Following the constructor is the destructor on line 40. Its only task is to free the memory occupied by the array once it is no longer needed.

compute_peratom()

If necessary, this function will resize the array to the number of atoms of concern, `nmax`. It does this using internal functions in the superclass *compute*, which we will not care to describe here. Finally it calls upon the function `pair_contribution()`, which fills the array with the pair-force from the potential. We omit the many-body part!

pair_contribution()

Note that we should only compute the force for one of the groups. factor 2...

6.2 Least squares plane regression

The method of least squares aims to find parameters which minimize the sum of the squared residuals, where residuals are the difference between observed values and the approximated value. We will use this method to approximate the slope of the surface of the substrate. This will be done by partitioning the system in a grid and do a plane approximation on each cell of the grid. In other words, we seek the coefficients in the plane equation

$$z = ax + by + c \tag{6.1}$$

that minimizes the sum of the squared residuals

$$S = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \left(z_i - f(x_i, y_i, \boldsymbol{\beta})\right)^2, \tag{6.2}$$

where $f(x_i, y_i, \boldsymbol{\beta})$ is the right hand side of the plane equation and $\boldsymbol{\beta}$ is the set of coefficients. The minima has the property that the differential with respect to any coefficient is zero.

$$\frac{\partial S}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial r_i^2}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial r_i^2}{\partial r_i} \frac{\partial r_i}{\partial \beta_j} = -2 \sum_{i=1}^{n} r_i \frac{\partial f(x_i, y_i, \boldsymbol{\beta})}{\partial \beta_j} = 0, \ \forall \ \beta_j \in \boldsymbol{\beta} \tag{6.3}$$

When approximating a plane we have three coefficients to account for: a, b and c. This leaves us with the following set of equations:

$$-2 \sum_{i=1}^{n} \left(z_i - ax_i - by_i - c\right) \frac{\partial}{\partial a} \left(ax_i + by_i + c\right) = 0 \tag{6.4}$$

$$-2 \sum_{i=1}^{n} \left(z_i - ax_i - by_i - c\right) \frac{\partial}{\partial b} \left(ax_i + by_i + c\right) = 0 \tag{6.5}$$

$$-2 \sum_{i=1}^{n} \left(z_i - ax_i - by_i - c\right) \frac{\partial}{\partial c} \left(ax_i + by_i + c\right) = 0, \tag{6.6}$$

which corresponds to

$$\sum_{i=1}^{n} z_i x_i = a \sum_{i=1}^{n} x_i^2 + b \sum_{i=1}^{n} x_i y_i + c \sum_{i=1}^{n} x_i \tag{6.7}$$

$$\sum_{i=1}^{n} z_i y_i = a \sum_{i=1}^{n} x_i y_i + b \sum_{i=1}^{n} y_i^2 + c \sum_{i=1}^{n} y_i \tag{6.8}$$

$$\sum_{i=1}^{n} z_i = a \sum_{i=1}^{n} x_i + b \sum_{i=1}^{n} y_i + nc. \tag{6.9}$$

This can be expressed as a matrix equation.

$$\begin{bmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & n \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x_i z_i \\ \sum y_i z_i \\ \sum z_i \end{bmatrix} \tag{6.10}$$

where we have omitted the indices to better readability. Solving this linear system retrieves the optimal coefficients in the sense of the least squares method. The normal vector of the plane will be $\mathbf{n} = [a, b, 1]$. This vector will be used to compute the size of the normal force. Since we know the average force on an atom in the chunk, and the normal vector from the approximated slope of the surface, we can compute the normal force simply as

$$\mathbf{F_N} = |\mathbf{F}| \cos\theta \frac{\mathbf{n}}{|\mathbf{n}|}. \tag{6.11}$$

The cosine of the angle between the two vectors is given as

$$\cos\theta = \frac{\mathbf{F} \cdot \mathbf{n}}{|\mathbf{F}| \cdot |\mathbf{n}|}, \tag{6.12}$$

meaning that the normal force may be expressed as

$$\mathbf{F_N} = (\mathbf{F} \cdot \mathbf{n}) \frac{\mathbf{n}}{|\mathbf{n}|^2}. \tag{6.13}$$

We will assume that the normal vector $\mathbf{n}$ is always in the same general direction as the average force, though obviously it may just as well point in the opposite direction and still be a normal vector to the plane. Programatically this was done in python as shown in Listing 6.3.

```
1     def getAngle(self, v1, v2):
2         'Computes angle between a vector and a line
              parallel to another vector'
3
4         lv1 = np.linalg.norm(v1)
5         lv2 = np.linalg.norm(v2)
6
7         angle = np.arccos(np.dot(v1,v2)/(lv1*lv2))
8         angle = min(angle, abs(np.pi-angle))
```

Listing 6.3: Python function to compute the smallest angle between a vector and a line parallel to another vector.

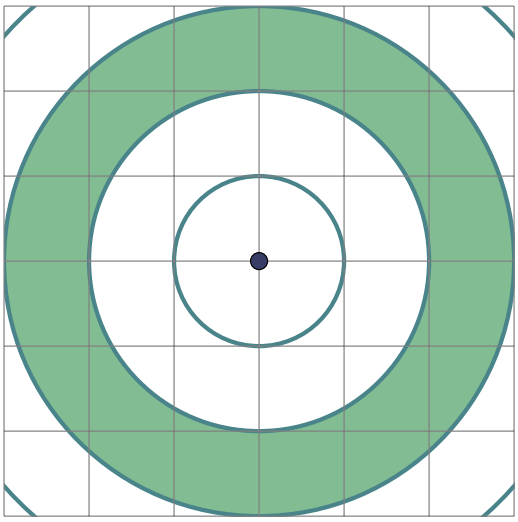


Figure 6.1: Radial binning. The third radial bin is colored, and its value will be the average value of the contribution within the bin.

6.3 Radial distribution

Our system is partitioned into a grid. Each cell in the grid holds an averaged value of the normal force in that cell. We would like to know the radial distribution of the normal force, which should express the averaged value of the normal force at a given radial distance from the center of the spherical indentation. We achieve this by defining discrete radial bins with finite bin width, and average contributions within each bin. A coarse method of doing this is to average the weights of the cells who's center is within the bin. This is illustrated in figure 6.2. In many applications where the bin width can be large compared to the length of the cells, this method might suffice. However, the current radii of the contact area between the sphere and the slab is only about 15-20 unit cells, and therefor having a large bin width will result in very few data points.

A different, slightly more sophisticated approach is to compute the fraction of the area of each cell that actually is withing the bin, and multiply this by the value associated with the cell. We can sum all these contributions and average them by dividing by the area of the bin. This means that even cells that do not have their center within the bin might contribute to the resulting bin value. How much, however, will depend on the fraction of the cell's area that is within the bin. This may be regarded as a smoothing of our coarse force distribution, and will give a more correct result than the coarse binning method already described. An illustration of this binning method is shown in figures 6.3. The figures clearly show that some cells have a larger area within the bin than others, and thus contribute more.

By exploiting the symmetry we are only required to compute the weights for

cells in a section of the domain, and not all cells. I have chosen to regard only the upper right corner of the system, meaning $x \geq x_c$ and $y \geq y_c$, where (x_c, y_c) is the coordinates of the center of the indentation. As the actual computations are easy to follow from the source code, the reader is recommended to take a look at it if this is of interest. Should I explain the computation of the weights or is that too intuitive?

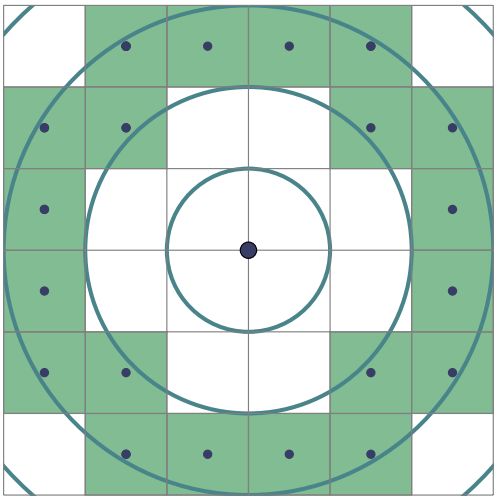


Figure 6.2: Coarse radial binning. The value appointed to the radial bin is the average of the weights of the cells who's center is within the bin. The cells with center within the third radial bin are colored, and their centers drawn.

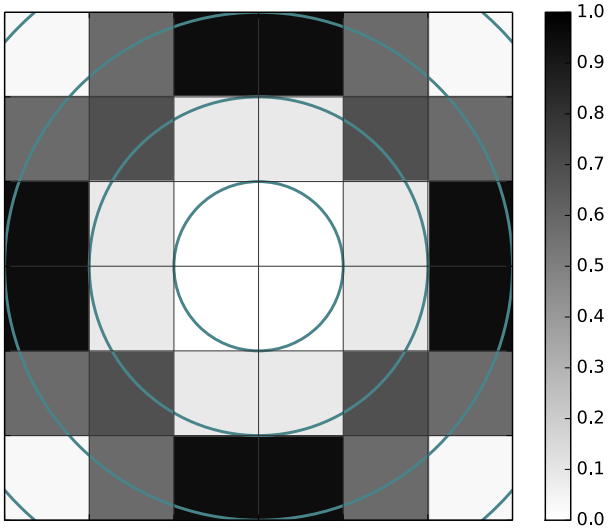


Figure 6.3: Radial binning based on weighted contributions of intersecting cells. The fraction of cell's area that are within the third radial bin is color coded; black being 1 and white being 0.

Chapter 7

Computing the coefficient of friction

As described in section 3.2.1, the coefficient of friction has a changing value depending on the relative motion of the objects in contact. If they move (slide), the coefficient is usually significantly lower than if they are stationary. In our simulations, we have done the same experiment as described in said chapter, only in a molecular dynamics simulation. We will experience that the velocity dependence of the coefficient of friction impose on us a challenge in interpreting our results.

7.1 Maintaining a normal force

From the same simulation done when we pushed the sphere down onto the substrate, we systematically saved a restart file at every 10000 time step. These can then be reloaded separately in order to compute the static force at different values of normal force. Throughout the simulation, we must be certain to remain the normal force acting on the rigid top at the same value as when the restart file was written. In our simulations we have used `fix addforce` with a constant value in the z-direction. When lowering the sphere, the load was linearly increased. Because we change the circumstance from a linear increase to a sudden constant value, we equilibrate a short while before measuring, in order to avert effects of pressure waves.

7.2 Adding a shear force

In order to add a shear force, we fix a spring with stiffness k to the rigid top layer of the sphere cap using the `Fix smd` command

```
fix ID sphereUpperG smd cvel ${k} ${v} tether 500 NULL
    NULL 0.0
```

Listing 7.1: LAMMPS command `Fix smd`, used to add a spring force to a group of atoms. The spring is considered fixed to the center of mass of the group and to a teatherd point, which we may move either with contant velocity of constant force.

which start out with the spring in its resting length, and begin pulling it with a constant velocity v toward the position $x = 500$. We disregard other dimensions, meaning we omit them from the distance computation and force application of the spring. At this point there should be two external forces acting on the rigid top of the sphere cap: the normal force in the z-direction, and the spring force in the x-direction. Each atom i will be assigned an external force

$$\mathbf{F}_i = -k(\mathbf{r}_i - \mathbf{r}_0)\frac{m_i}{M}, \tag{7.1}$$

where k is the spring constant, $\mathbf{r}_i$ and $\mathbf{r}_0$ are the position of atom i and the tethered point, respectively, m_i is the mass of atom i, and M is the total mass of the group of atoms. As we only regard the x-dimension, the position vectors are evaluated as $\mathbf{r}_i = x_i$ and equivalently $\mathbf{r}_0 = x_0$.

7.3 Procedure

In the completed simulations we have used a time frame long enough for the spring force to start decreasing (slip). Typically they were of the order of 50000 time steps (0.1ns). The procedure was equivalent for all time steps, only the velocity at which we pull the spring was increased between simulations. We increase the velocity between simulations because we expect the static force F_s to increase when the normal force increases. In order for the spring to exert a force equal to F_s, we have to extend the spring farther. By increasing the velocity at which we pull the spring we reduce the number of time steps needed, saving time and computational expense. This can be seen in figure 7.3.

`fix smd` computes 7 quantities, that are stored as a vector. As described in the documentation[1], these quantities are: the x-, y-, and z-component of the pulling force, the total force in direction of the pull, the equilibrium distance of the spring, the distance between the two reference points, and finally the accumulated PMF (the sum of pulling forces times displacement), this order. We are interested in the first value, i.e. the total pulling force component in the x-direction. By setting up a custom format for the *thermo_style*, we consistently store the value of the total force component in the x-direction every 100 time step. This way, we can monitor the force applied by the spring as a function of time.

[1] http://lammps.sandia.gov/doc/fix_smd.html

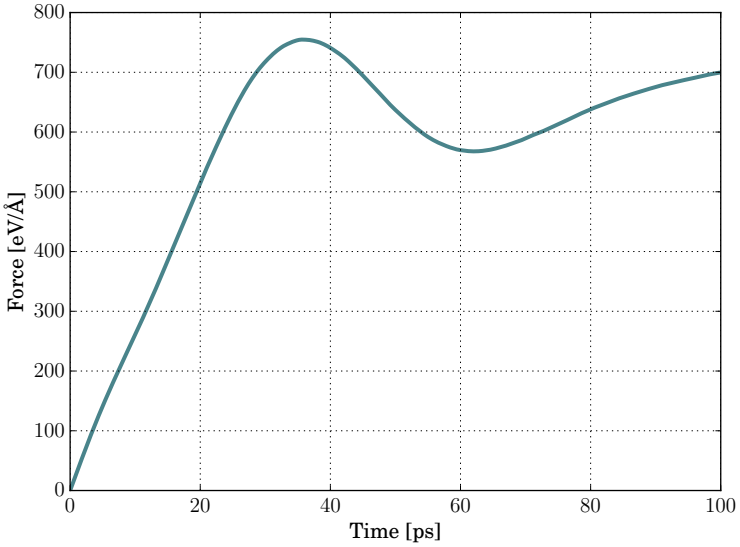


Figure 7.1: Spring force as a function of time, where we pull the spring by a constant velocity $v = 3$ [Å/ps].

7.4 Interpreting the results

The result from a single simulation is shown in figure 7.1. As we see, the spring force increases linearly as expected from Hooke's law. We are really only interested in finding the maximum value of spring force, namely the static friction F_s. Since we know the value of the normal force, we may be enticed to compute the coefficient of static friction as $\mu = F_s/N$ momentarily. However, this would be erroneous. It turns out that the static friction is dependent on the velocity at which we pull the spring. F_s increases with the velocity. Thus, the spring force will engage sliding at a lower value at slow pulling speeds, than at high. This can be deduced from our results as well. Figure 7.2 show the evolution of the spring force from 5 different simulations, all using a different velocity, and demonstrating that the maximum spring force is different in each case. We may rescale the x-axis by multiplying with the velocity to obtain the spring force as a function of the extension of the spring. So, how are we supposed to conclude a coefficient of friction from our simulations? The solution is to find the relation between the coefficent and the velocity. By using the dataset shown in figure 7.2 and 7.3 we may plot $\mu(v)$ by computing $\mu = F_s/N$ for every velocity. The result is shown in figure 7.4. The relation is obviously linear, at least for the considered range of velocities. By linear regression the approximated function for the velocity dependent coefficient of friction is

$$\mu(v) = 0.0702v + 0.3992. \tag{7.2}$$

By rescaling the axis of figure 7.3 by this function, we get data collapse as shown in figure 7.5. Note that we normalize the y-axis by dividing with the normal force N. Collapse of the curves indicate that at all these velocities the system is subject to the same physical process. Had the curves been very distinguishable, we could conclude that there is some other effect taking place. These curves overlap quite nicely, so I do not see any cause of concern.

The collapse is not of use to us in the pursue of computing the coefficient of static friction. The result that is of real interest is the relation posed in equation (7.2). This implies that the minimum value of the static coefficient of friction for SiO_2 on SiO_2 is $\mu = 0.4$. In macroscopic experiments the spring is loaded very slowly, and thus the constant term of the linear regression is the coefficient of static friction we have been looking for. Thus for every value of the normal force, we need to take a series of simulations using different velocities, and do a linear regression on the resulting points. It is not intuitive how $\mu(v)$ will change as we perturb N. However, we may still expect a linear dependence.

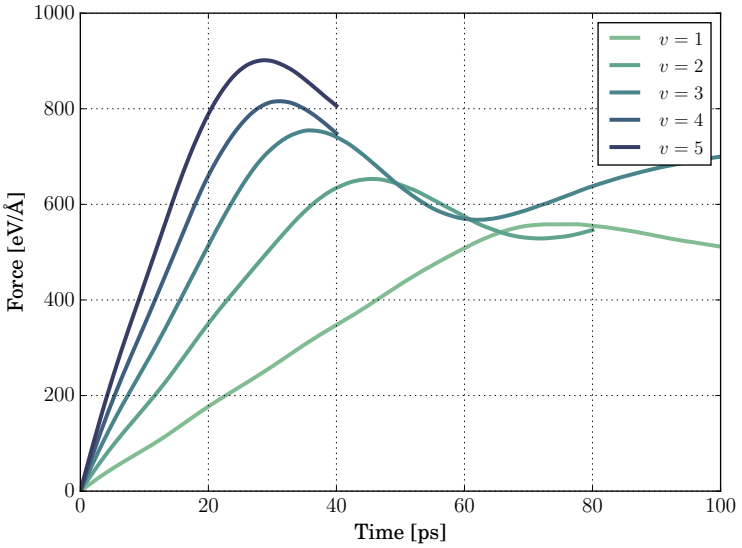


Figure 7.2: Spring force as a function of time for several values of velocity.

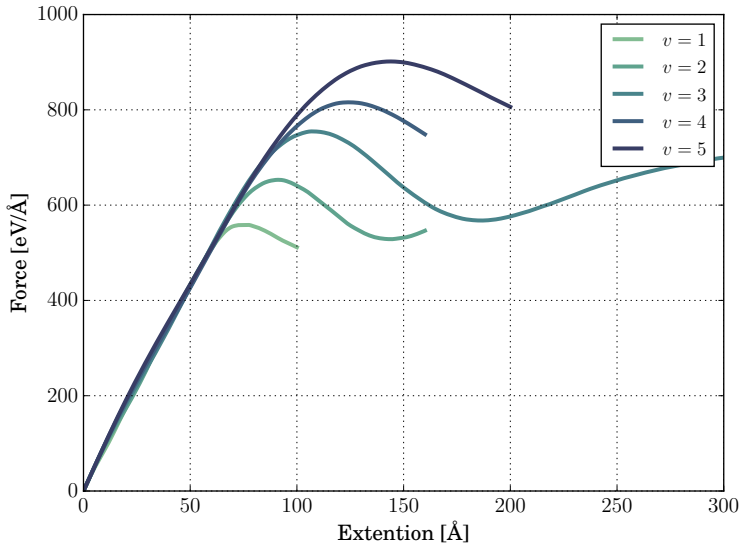


Figure 7.3: Spring force as a function of the extension of the spring for several values of velocity.

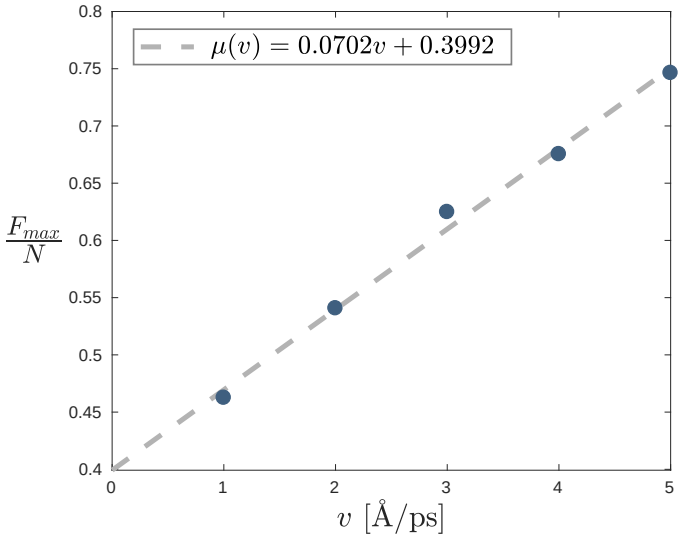


Figure 7.4: Coefficient of friction μ as a function of velocity v. Linear regression is suited to approximate the relation $\mu(v)$.

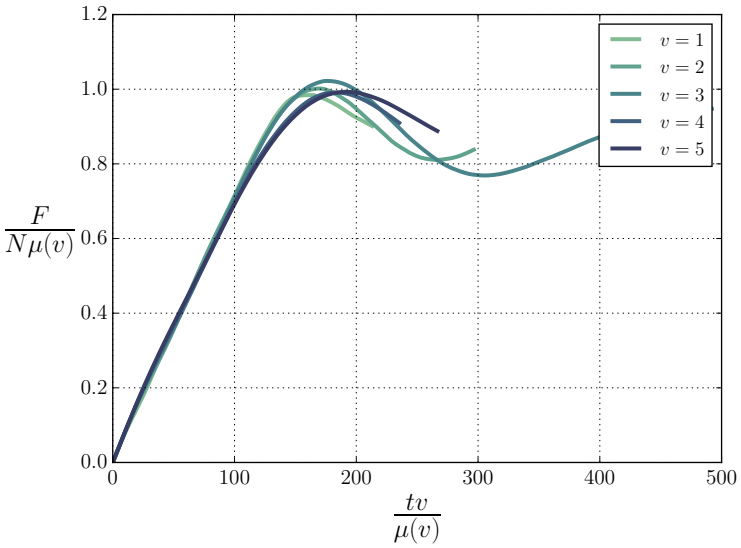


Figure 7.5: Data collapse of series of simulations utilizing different velocities.

Table 7.1: Measurement results for simulation with substrate of thickness $h = 58$Å. *Step* is the time step at which the loaded restart file was saved. k is the spring stiffness, v the velocity by which we pull the spring, N the measured normal force, and F_T the force applied to the rigid top of the sphere cap at the point slip occurs. The units are as described by the *metal* convention listed in table B.1.

Step	k	v	N	F_T
80000	10	-0.1	541	259
90000	10	-0.2	672	391
100000	10	-0.2	805	506
110000	10	-0.2	942	552
120000	10	-0.2	1077	608
130000	10	-0.2	1208	731
140000	10	-0.2	1341	853
150000	10	-0.2	1480	888
160000	10	-0.2	1613	936
170000	10	-0.2	1745	1142
180000	10	-0.2	1881	1400
190000	10	-0.2	-	-
200000	10	-0.2	2150	1502

Part III

Results

Part IV

Discussion

Appendix A

Source code

A.1 compute_group_group.h

```
#ifdef COMPUTE_CLASS
ComputeStyle(group/group, ComputeGroupGroup)
#else

#ifndef LMP_COMPUTE_GROUP_GROUP_H
#define LMP_COMPUTE_GROUP_GROUP_H

#include "compute.h"

namespace LAMMPS_NS {

class ComputeGroupGroup : public Compute {
public:
     ComputeGroupGroup(class LAMMPS *, int, char **);
     ~ComputeGroupGroup();
     void init();
     void init_list(int, class NeighList *);
     double compute_scalar();
     void compute_vector();

protected:  // private
     char *group2;
     int jgroup,jgroupbit,othergroupbit;
     double **cutsq;
     double e_self,e_correction;
     int pairflag,kspaceflag,boundaryflag;
     class Pair *pair;
     class NeighList *list;
     class KSpace *kspace;

     virtual void pair_contribution();
     void kspace_contribution();
     void kspace_correction();
};

}

#endif
#endif
```

A.2 compute_group_group_atom.h

```
#ifdef COMPUTE_CLASS
ComputeStyle(group/group/atom,ComputeGroupGroupAtom)
#else

#ifndef LMP_COMPUTE_GROUP_GROUP_ATOM_H
#define LMP_COMPUTE_GROUP_GROUP_ATOM_H

#include "compute.h"
#include "compute_group_group.h"

namespace LAMMPS_NS {

class ComputeGroupGroupAtom : public ComputeGroupGroup {
 public:
  ComputeGroupGroupAtom(class LAMMPS *, int, char **);
  ~ComputeGroupGroupAtom();
  void compute_peratom() override;
  int nmax;
  double **carray;

 private:
  void pair_contribution() override;
};
}
#endif
#endif
```

A.3 compute_group_group_atom.cpp

```
#include <mpi.h>
#include <string.h>
#include "compute_group_group_atom.h"
#include "atom.h"
#include "update.h"
#include "force.h"
#include "pair.h"
#include "neighbor.h"
#include "neigh_request.h"
#include "neigh_list.h"
#include "group.h"
#include "kspace.h"
#include "error.h"
#include <math.h>
#include "comm.h"
#include "domain.h"
#include "math_const.h"
#include "memory.h"

#include <iostream>
using namespace LAMMPS_NS;
using namespace MathConst;

#define SMALL 0.00001

ComputeGroupGroupAtom::ComputeGroupGroupAtom(LAMMPS *lmp,
    int narg, char **arg) :
     ComputeGroupGroup(lmp, narg, arg),
     carray(NULL),
     nmax(0)
{
     if (narg < 4) error->all(FLERR,"Illegal compute
        group/group command");

     peratom_flag      = 1;  // Indicating a peratom compute
     size_peratom_cols = 4;  // # of Columns per atom.
     extarray          = 0;  // 0/1 if global array is all
        intensive/extensive
     scalar_flag       = 0;
     vector_flag       = 0;
}


ComputeGroupGroupAtom::~ComputeGroupGroupAtom()
{
     memory->destroy(carray);
}

```

```
46
47  void ComputeGroupGroupAtom::compute_peratom()
48  {
49      // grow array if necessary
50      if (atom->nmax > nmax) {
51
52          memory->destroy(carray);
53          nmax = atom->nmax;
54          memory->create(carray, nmax, size_peratom_cols,
                "group/group/atom:carray");
55          array_atom = carray;
56
57      }
58
59      if (pairflag) pair_contribution();
60      if (kspaceflag) kspace_contribution(); // This doesn't
            happen though. See compute_group_group.cpp
            constructor.
61  }
62
63
64  void ComputeGroupGroupAtom::pair_contribution()
65  {
66      int i,j,ii,jj,inum,jnum,itype,jtype;
67      double xtmp,ytmp,ztmp,delx,dely,delz;
68      double rsq,eng,fpair,factor_coul,factor_lj;
69      int *ilist,*jlist,*numneigh,**firstneigh;
70
71      double **x = atom->x;
72      int *type = atom->type;
73      int *mask = atom->mask;
74      int nlocal = atom->nlocal;
75      double *special_coul = force->special_coul;
76      double *special_lj = force->special_lj;
77      int newton_pair = force->newton_pair;
78      double *columns;
79
80      // invoke half neighbor list (will copy or build if
            necessary)
81
82      neighbor->build_one(list);
83
84      inum = list->inum;
85      ilist = list->ilist;
86      numneigh = list->numneigh;
87      firstneigh = list->firstneigh;
88
89      // loop over neighbors of my atoms
90      // skip if I,J are not in 2 groups
91
92
```

```
93      for (ii = 0; ii < inum; ii++) {
94          i = ilist[ii];
95
96          array_atom[i][0] = 0;
97          array_atom[i][1] = 0;
98          array_atom[i][2] = 0;
99          array_atom[i][3] = 0;
100     }
101
102     for (ii = 0; ii < inum; ii++) {
103         i = ilist[ii];
104
105         // skip if atom I is not in either group
106         if (!(mask[i] & groupbit || mask[i] & jgroupbit))
                continue;
107
108         xtmp = x[i][0];
109         ytmp = x[i][1];
110         ztmp = x[i][2];
111         itype = type[i];
112         jlist = firstneigh[i];
113         jnum = numneigh[i];
114
115         for (jj = 0; jj < jnum; jj++) {
116             j = jlist[jj];
117             factor_lj = special_lj[sbmask(j)];
118             factor_coul = special_coul[sbmask(j)];
119             j &= NEIGHMASK;
120
121             // skip if atom J is not in either group
122             if (!(mask[j] & groupbit || mask[j] &
                    jgroupbit)) continue;
123
124             int ij_flag = 0;
125             int ji_flag = 0;
126             if (mask[i] & groupbit && mask[j] & jgroupbit)
                    ij_flag = 1;
127             if (mask[j] & groupbit && mask[i] & jgroupbit)
                    ji_flag = 1;
128
129             // skip if atoms I,J are only in the same group
130             if (!ij_flag && !ji_flag) continue;
131
132             delx = xtmp - x[j][0];
133             dely = ytmp - x[j][1];
134             delz = ztmp - x[j][2];
135             rsq = delx*delx + dely*dely + delz*delz;
136             jtype = type[j];
137
138             if (rsq < cutsq[itype][jtype]) {
139                 eng = pair->single(i, j, itype, jtype,
```

```
                        rsq, factor_coul, factor_lj, fpair);
140
141                     // energy only computed once so tally full
                            amount
142                     // force tally is jgroup acting on igroup
143
144                     if (newton_pair || j < nlocal) {
145                         array_atom[i][0] += eng;
146                         if (ij_flag) {
147                             array_atom[i][1] += delx*fpair;
148                             array_atom[i][2] += dely*fpair;
149                             array_atom[i][3] += delz*fpair;
150                         }
151                         if (ji_flag) {
152                             array_atom[j][1] -= delx*fpair;
153                             array_atom[j][2] -= dely*fpair;
154                             array_atom[j][3] -= delz*fpair;
155                         }
156
157                         // energy computed twice so tally half
                                amount
158                         // only tally force if I own igroup
                                atom
159                     }
160                     else {
161                         array_atom[i][0] += 0.5*eng;
162                         if (ij_flag) {
163                             array_atom[i][1] += delx*fpair;
164                             array_atom[i][2] += dely*fpair;
165                             array_atom[i][3] += delz*fpair;
166                         }
167                     }
168                 }
169             }
170         }
171     }
```

Appendix B

Something

B.1 LAMMPS units

Table B.1: Unit convention of the *metal* unit style in LAMMPS.

Mass	grams/mole
Distance	Angstroms
Time	picoseconds
Energy	eV
Velocity	Angstroms/picosecond
Force	eV/Angstrom
Torque	eV
Temperature	Kelvin
Pressure	bars
Dynamic viscosity	Poise
Charge	multiple of electron charge
Dipole	charge · Angstroms
Electric field	volts / Angstrom
Density	gram / cmˆdim

Bibliography

[1] "Friction." Merriam-Webster.com. Merriam-Webster, n.d. Web. 25 Feb. 2017.

[2] Ian M. Hutchings. Leonardo da Vinci's studies of friction. *Wear*, 360-361:51–66, 2016.

[3] J. L. Lebowitz, J. K. Percus, and L. Verlet. Ensemble dependence of fluctuations with application to machine computations. *Physical Review*, 153(1):250–254, 1967.

[4] Knut Mørken. Numerical Algorithms and Digital Representation, 2016.

[5] Bo N. J. Persson. *Sliding Friction*. NanoScience and Technology. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[6] W Voight. Lehrbuch der Kristallphysik. *Teubner, Leipzig*, 962:1928, 1928.