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

1 Introduction **1**

2 Molecular dynamics **3**
- 2.1 Time-integration . . . 3
- 2.2 Potentials . . . 3
  - 2.2.1 Lennard-Jones . . . 3
  - 2.2.2 Stillinger-Weber . . . 4
  - 2.2.3 Vashishta . . . 4
- 2.3 Boundary conditions . . . 5
  - 2.3.1 Minimum image convention . . . 6
- 2.4 Thermostats . . . 8
  - 2.4.1 Berendsen . . . 8
  - 2.4.2 Andersen . . . 9
  - 2.4.3 Nosé-Hoover . . . 9
- 2.5 Efficiency improvements . . . 9
  - 2.5.1 Cut-off . . . 9
  - 2.5.2 Cell lists and neighbor lists . . . 10
  - 2.5.3 Parallelization . . . 11

3 LAMMPS **13**
- 3.1 Installation . . . 13
  - 3.1.1 Linux . . . 13
  - 3.1.2 Mac OS X with Homebrew . . . 14
- 3.2 Input scripts . . . 15
  - 3.2.1 System configurations . . . 15
  - 3.2.2 Run-time commands . . . 16
  - 3.2.3 Output . . . 18
- 3.3 Visualization . . . 20
  - 3.3.1 OVITO . . . 20

4 Preparing a molecular dynamics simulation **21**
- 4.1 Silica . . . 21
  - 4.1.1 Unit cell of β-cristobalite . . . 22

4.2 Building a crystal 22
4.3 Verifications 24
4.3.1 Melting point 24
4.4 Shaping the silica 26
4.5 Moving the sphere towards the slab 28

5 This must be sorted in designated chapters 29
5.1 Radial distribution of normal force 29
5.1.1 Radial binning 29

6 Computing the normal force distribution 33
6.1 Creating a custom compute 33
6.1.1 Find a similar compute 34
6.1.2 Creating the class 34
6.2 Least squares regression 40

A Source code 43
A.1 compute_group_group.h 44
A.2 compute_group_group_atom.h 45
A.3 compute_group_group_atom.cpp 46

B Something 51
B.1 LAMMPS units 51

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?

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 a mass, and the interaction between them is described by a force field. Molecular dynamics aid our understanding of macroscopic phenomena by giving us insight in microscopic effects. 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

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

$$V(r) = 4\epsilon \left[\left(\frac{\theta}{r} \right)^{12} - \left(\frac{\theta}{r} \right)^{6} \right], \tag{2.1}$$

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. 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 $r = 2^{1/6}\theta$.

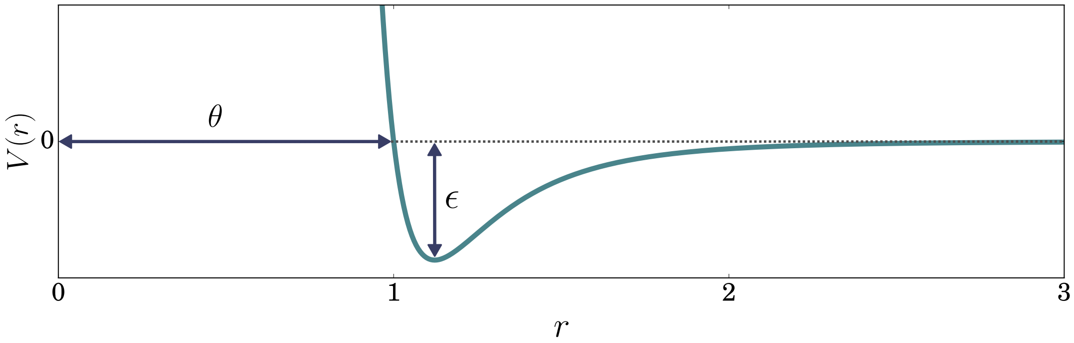


Figure 2.1: Lennard-Jones potential as function of inter-atomic distance, r, in units of θ.

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

$$V = \sum_{i<j} V_{ij}^{(2)}(r_{ij}) + \sum_{i<j<k} V_{ijk}^{(3)}(\mathbf{r}_{ij}, \mathbf{r}_{ik}), \tag{2.2}$$

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

$$V_{ij}^{(2)}(r) = \frac{H_{ij}}{r^{\eta_{ij}}} + \frac{Z_i Z_j}{r} e^{-r/r_{1s}} - \frac{D_{ij}}{2r^4} e^{-r/r_{4s}} - \frac{W_{ij}}{r^6}. \tag{2.3}$$

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

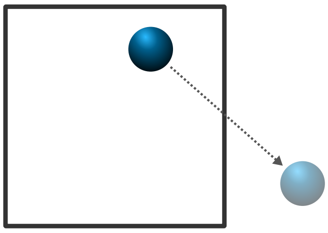
$$V^{(3)}_{ijk}(\mathbf{r}_{ij}, \mathbf{r}_{ik}) = B_{ijk} \exp\left(\frac{\xi}{r_{ij} - r_0} + \frac{\xi}{r_{ik} - r_0}\right) \frac{(\cos\theta_{ijk} - \cos\theta_0)^2}{1 + C_{ijk}(\cos\theta_{ijk} - \cos\theta_0)^2} \tag{2.4}$$

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 (2.9).

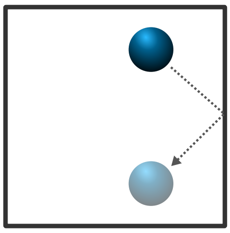
2.3 Boundary conditions

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.

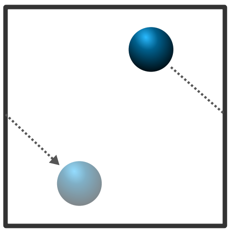
No boundary conditions. Particles are not subject to any special rules at any boundry. The system size may be regarded as infinite. This might be a reasonable choise when studying explosions for instance, or in experiments that is on a really short time scale.



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.

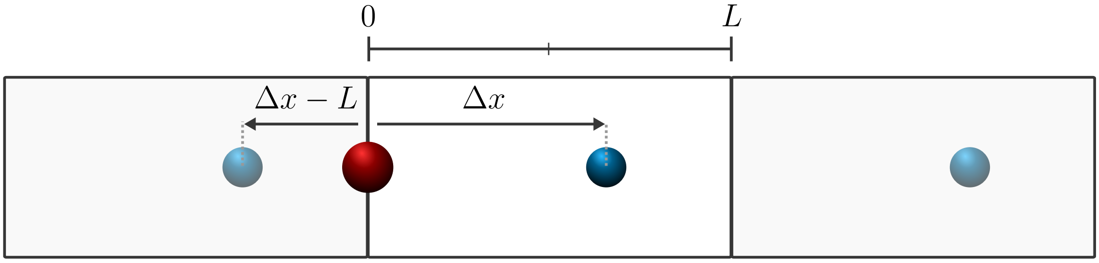


Figure 2.2: Minimum image convention in 1 dimension. If an atom is separated from the reference atom (red) by more than half the system length, $L/2$, the minimum image convention states that the distance between the two is the distance to the periodic copy, which is $|\Delta x - L|$. The white area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

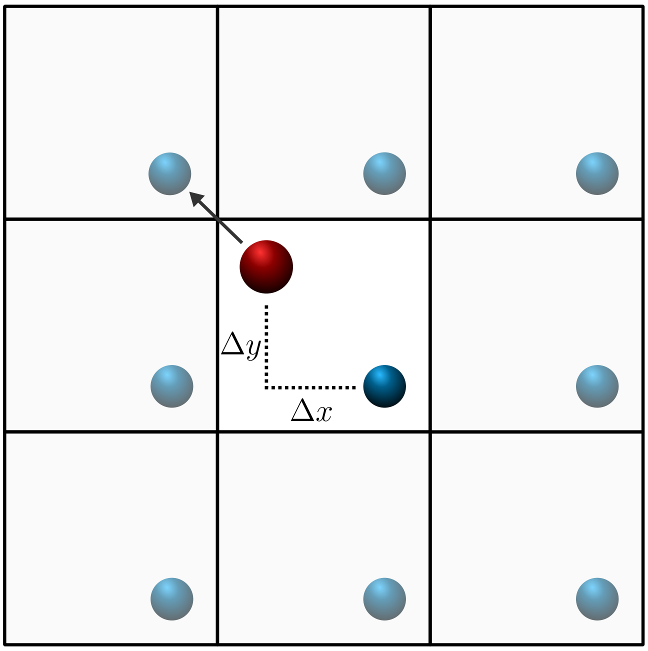


Figure 2.3: Minimum image convention in 2 dimensions. The position of an atom, with respect to the reference atom (red), is the position of the original or any of the replicated images that has the shortest distance to the reference atom. The minimum image is found by decomposing the atoms position, 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 Thermostats

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

$$E_k = \frac{f}{2} N k_b T \tag{2.5}$$

, for a system of N atoms with$f$ degrees of freedom and temperature T. k_b is the Boltzmann constant. If we assume newtonian motion, the total kinetic energy in a system is statistically

$$E_k = \frac{1}{2} \langle m_i \mathbf{v_i}^2 \rangle. \tag{2.6}$$

We can equate these to find the statistical temperature given as

$$T = \frac{\langle m_i \mathbf{v_i}^2 \rangle}{f N k_b}. \tag{2.7}$$

Thus, we can approximate the instantaneous temperature using the atoms velocity. This value will fluctuate about the real value, so time-averaging is often used when measuring.

Often, we want to control the temperature of our simulated system. This is done by fixing the temperature in equation (2.7) and manipulate the atoms velocities. There are several methods to 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.

2.4.1 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

$$\gamma = \sqrt{1 + \frac{\Delta t}{\tau} \left(\frac{T_{bath}}{T} - 1 \right)}, \tag{2.8}$$

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.

2.4.2 Andersen

The Andersen thermostat simulates (hard) collisions between atoms inside the system and in the heat bath. Atoms which collide will gain a new normally distributed velocity with standard deviation $\sqrt{k_B T_{bath}/m}$. For all atoms, a random uniformly distributed number in the interval $[0, 1]$ is generated. If this number is less than $\Delta t/\tau$, the atom is considered to have collided, and is assigned a new velocity. In this case, τ is treated as a collision time, and should have about the same value as the τ in the Berendsen thermostat ($20\Delta t$). The Andersen thermostat is very useful when equilibrating systems, but disturbs the dynamics of e.g. lattice vibrations.

2.4.3 Nosé-Hoover

2.5 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.

2.5.1 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.

$$V(r) = \begin{cases} V(r) - V(r_c) - \frac{\partial V(r)}{\partial r}\Big|_{r=r_c}(r - r_c) & , r \leq r_c \\ 0 & , r > r_c \end{cases} \tag{2.9}$$

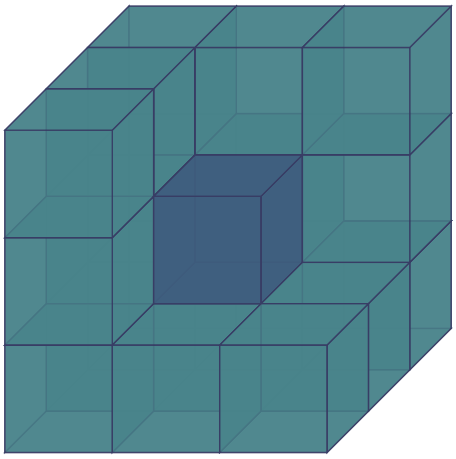


Figure 2.4: Illustrative figure of cells of concern when building the neighbor lists. The lighter cubes are neighboring cells; the darker cube is the cell containing the reference atom. The 7 cells in front of the dark cell are removed from the figure, but are also included.

which implies the force

$$F(r) = \begin{cases} F(r) = F(r) - F(r_c) & , r \leq r_c \\ 0 & , r > r_c \end{cases} \tag{2.10}$$

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.

2.5.2 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 5.4. 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!

---

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

2.5.3 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

$$S = \frac{T_s}{T_p}, \tag{2.11}$$

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 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 × 15 × 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

fix nvt all nvt temp ${T} ${T} 1.0
```

```
run ${N}
```

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

```
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.

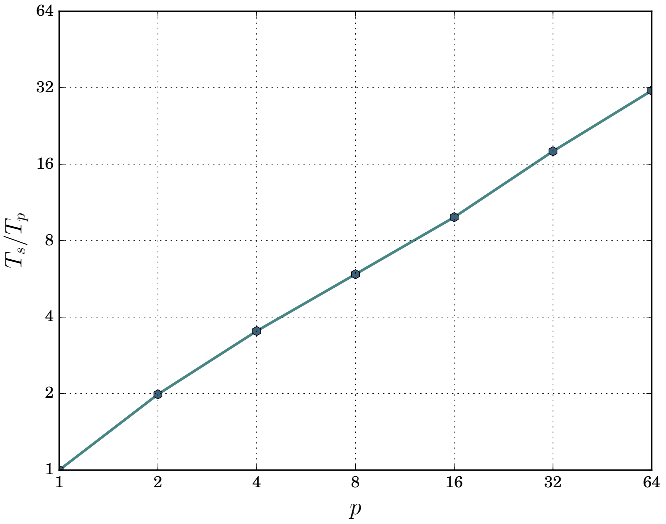


Figure 2.5: Speedup as a function of number of processors.

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

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.

3.1 Installation

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

3.1.1 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 xvf 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.

```
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 makes the resulting executable compatible with parallelization in MPI.

```
1 make serial
2 make 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.

```
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.

3.1.2 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.

```
brew info lammps
```

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

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

---

[2] http://brew.sh/

In the example shown we installed the package manybody.

3.2 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.

3.2.1 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 3.1 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 3.1: 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.

3.2.2 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.

```
variable A equal 1000
variable B equal step/${A}
variable C equal ${B}
variable D equal v_B
```

Listing 3.2: 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 4.3. 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:

```
c_computeID
c_computeID[1]
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 3.2.3 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]
```

3.2.3 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.

```
dump dumpID1 all atom 100 name.dump
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..

Fixes

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

```
fix timeAvgID all ave/time 100 5 1000 c_comID file name.avg
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.

```
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.

```
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.

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

3.3 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.

3.3.1 OVITO

Chapter 4

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.

4.1 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

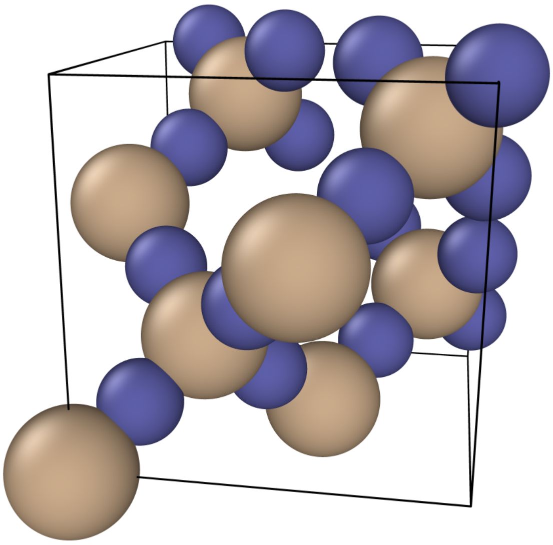


Figure 4.1: Unit cell of b-cristobalite. Tan and blue spheres represent silicon and oxygen atoms respectively. The unit cell is cubical, with edges of length 7.12Å.

4.1.1 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.

4.2 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 4.1. 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 4.1: 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

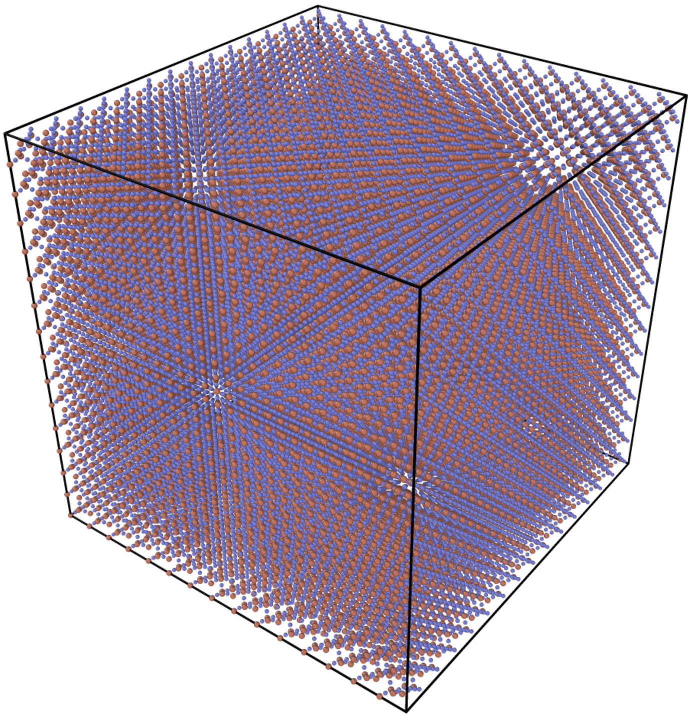


Figure 4.2: System built from 15 × 15 × 15 unit cells of b-cristobalite.

have a crystal of the desired size. For our purpose, we generate a large cube of 15 × 15 × 15 unit cells. This is done as follows.

4.3 Verifications

4.3.1 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 designated temperature. 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:

$$\langle r^2(t)\rangle = \frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{r}_i(t) - \mathbf{r}_i(0)\right)^2 . \tag{4.1}$$

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 ${N}
    compute msdID all msd
    fix msdDumpID all ave/time 1 1 3 c_msdID[4] file msd_N${N}_T${T}.txt
    run ${N}
    uncompute msdID
    unfix nvtID
    unfix msdDumpID
    variable T equal ${T}+50
next i
jump SELF meltingLoop
```

Listing 4.2: Main blalbabla.

---

[5] http://lammps.sandia.gov/doc/compute_msd.html

[6] https://en.wikipedia.org/wiki/Cristobalite

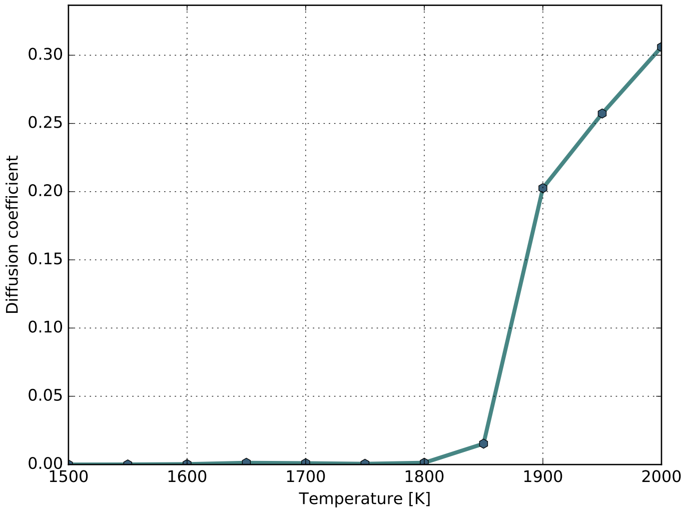


Figure 4.3: Mean square displacement as a function of temperature. The face transition occurs where we see a rapid change in the behavior.

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.

4.4 Shaping the silica

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 4.3, 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 4.3: Defining regions to keep or delete from a system of dimensions 106.8 × 106.8 × 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 4.3 on the system is shown in figure 4.2 is shown in figure 4.4, where our perspective is along the y-axis.

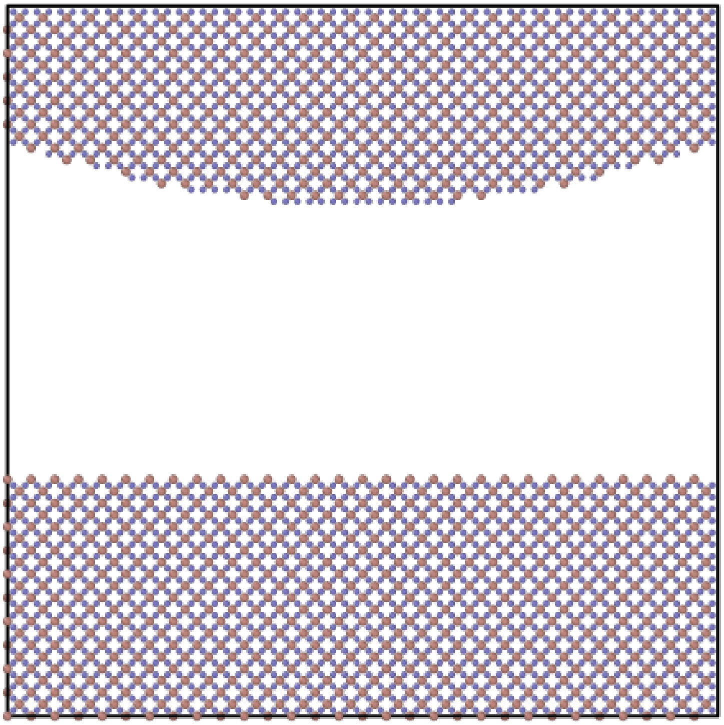


Figure 4.4: xz-perspective on a system built from 15 × 15 × 15 unit cells of b-cristobalite, with certain regions carves out. This is a result from applying Listing 4.3 to the system shown in figure 4.2. The top shape is a sphere cap, while the bottom is a slab.

4.5 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 probably a lot of smart ways to do this. The author has 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. 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. Then, 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.

```
fix freezeID groupID setforce 0 0 0
velocity groupID set 0 0 0
```

Listing 4.4: LAMMPS commands for hard coding the forces and velocities of atoms within a specific group. Effectively freezing them.

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

Listing 4.5: LAMMPS command for changing the size of the simulation box.

Chapter 5

This must be sorted in designated chapters

5.1 Radial distribution of normal force

In order to find a radial distribution of the normal force, F_N, we partition the system into a grid in the xy-plane. We then use the command

```
1   compute chunkID all chunk/atom bin/2d x 0 7.12 y 0 7.12
2   compute stressID all stress/atom NULL
3   fix fixChunkID all ave/chunk 1 1 10 chunkID
        c_stressID[3] file forcesInChunks.txt
```

to compute the stress of every chunk in the z-direction, σ_{zz} (sum of every individual atom stress in the chunk).

Line 1 establishes the grid, with bin width 7.12Å.

Line 2 creates a compute of the stress

Line 3 stores the sum of individual stresses in each chunk to the file `forcesInChunks.txt`. This is done every 10 time steps in order to reduce correlation effects.

The data in stored from each time step can easily be averaged to produce a result as shown in figure X.

We can then find the radial distribution simply by binning this matrix in radial bins, and average the normal forces of the chunks within the bins.

5.1.1 Radial binning

Our system is partitioned into a grid. Each cell in the grid holds an averaged value of the normal force in that cell. The radial distribution of normal force should express the averaged value of the normal force at a given radial distance from the center of the sphere. 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 5.1. In many applications where the bin size can be large compared to the length

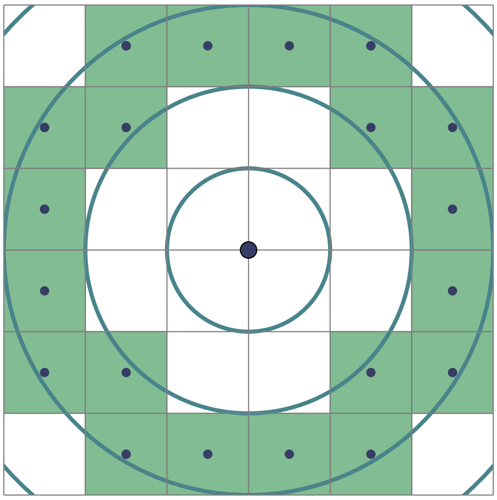


Figure 5.1: Coarse radial binning. The value appointed to the radial bin is the average of the weights of the cells who's center is within the bin. The cells with center within the third radial bin are colored, and their centers drawn.

of the cells, this method might suffice. However, the current radii of the contact area between the sphere and the slab is only about 10 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 weight 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 that is within the bin. This may be regarded as a smoothing of our coarse force distribution, and will probably give a more correct result than the coarse binning method already described. An illustration of this binning method is shown in figure 5.2. The figure clearly show that some cells have a larger area within the bin than others, and thus contribute more.

Computing the area of the cell within the bin should be described here!

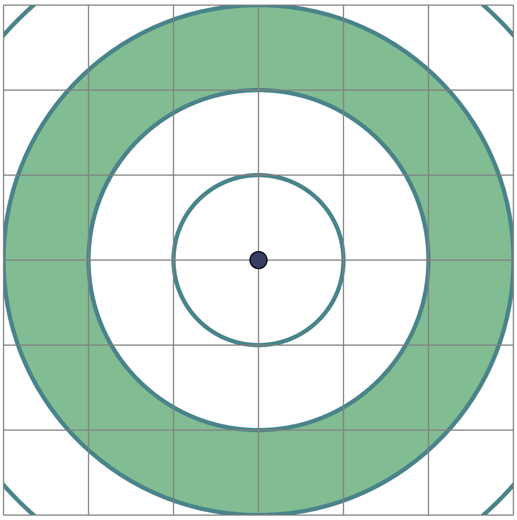


Figure 5.2: Radial binning based on weighted contributions of intersecting cells. The third radial bin is colored, and its value will be the average of the intersecting cells weight times the fraction of the cells area that intersects the bin.

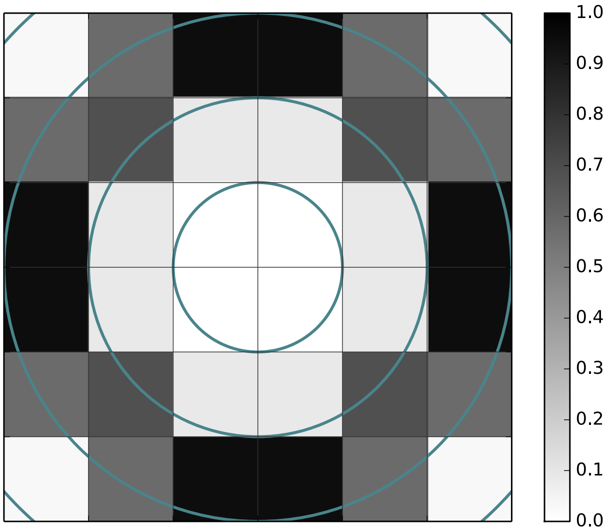


Figure 5.3: Fraction of cell's area that are within the third radial bin.

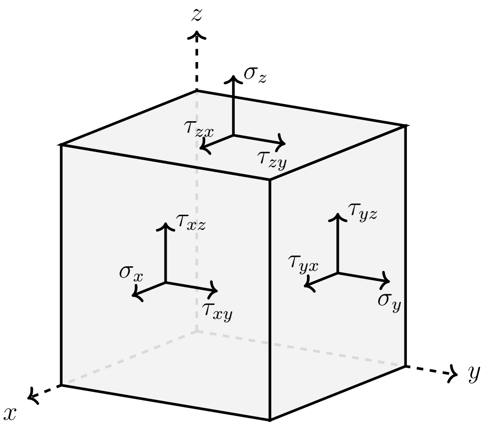

z
σ_z
τ_{zx}
τ_{zy}
τ_{xz}
σ_x
τ_{xy}
τ_{yz}
τ_{yx}
σ_y
y
x

Figure 5.4: Stress

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. The details of that procedure will be described.

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

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.

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 purpose of the class was to save the forces acting on atoms in one group from atoms of another group. In this section we will try do give brief instructions on how this was done.

[MAYBE AN ILLUSTRATIVE FIGURE HERE]

6.1.1 Find a similar compute

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!

For instance there is a compute named *group/group*[1] 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 total force acting on all atoms from atoms of other groups. It should also work with the Vashishta potential.

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.

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 complete header-file is shown in Listing 6.2 and explained in detail below.

```
#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;
```

---

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

```
  double **carray;

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

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

`ComputeStyle` defines the command to be used in the LAMMPS 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` × 3.

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

```
#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"

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);
}


void ComputeGroupGroupAtom::compute_peratom()
{
    // grow array if necessary
    if (atom->nmax > nmax) {
        memory->destroy(carray);
        nmax = atom->nmax;
        memory->create(carray, nmax, size_peratom_cols, "group/group/atom:carray");
        array_atom = carray;
    }

    if (pairflag) pair_contribution();
    if (kspaceflag) kspace_contribution(); // This doesn't happen though. See compute_group_group.cpp constructor.
}


void ComputeGroupGroupAtom::pair_contribution()
{
    int i,j,ii,jj,inum,jnum,itype,jtype;
    double xtmp,ytmp,ztmp,delx,dely,delz;
    double rsq,eng,fpair,factor_coul,factor_lj;
    int *ilist,*jlist,*numneigh,**firstneigh;

    double **x = atom->x;
    int *type = atom->type;
    int *mask = atom->mask;
    int nlocal = atom->nlocal;
    double *special_coul = force->special_coul;
    double *special_lj = force->special_lj;
    int newton_pair = force->newton_pair;
    double *columns;

    // invoke half neighbor list (will copy or build if necessary)

    neighbor->build_one(list);

    inum = list->inum;
    ilist = list->ilist;
    numneigh = list->numneigh;
    firstneigh = list->firstneigh;

    // loop over neighbors of my atoms
    // skip if I,J are not in 2 groups


    for (ii = 0; ii < inum; ii++) {
        i = ilist[ii];

        // skip if atom I is not in either group
        if (!(mask[i] & groupbit || mask[i] & jgroupbit)) continue;

        xtmp = x[i][0];
        ytmp = x[i][1];
        ztmp = x[i][2];
        itype = type[i];
        jlist = firstneigh[i];
        jnum = numneigh[i];

        for (jj = 0; jj < jnum; jj++) {
            j = jlist[jj];
            factor_lj = special_lj[sbmask(j)];
            factor_coul = special_coul[sbmask(j)];
            j &= NEIGHMASK;

            // skip if atom J is not in either group
            if (!(mask[j] & groupbit || mask[j] & jgroupbit)) continue;

            int ij_flag = 0;
            int ji_flag = 0;
            if (mask[i] & groupbit && mask[j] & jgroupbit) ij_flag = 1;
            if (mask[j] & groupbit && mask[i] & jgroupbit) ji_flag = 1;

            // skip if atoms I,J are only in the same group
            if (!ij_flag && !ji_flag) continue;

            delx = xtmp - x[j][0];
            dely = ytmp - x[j][1];
            delz = ztmp - x[j][2];
            rsq = delx*delx + dely*dely + delz*delz;
            jtype = type[j];

            if (rsq < cutsq[itype][jtype]) {
                eng = pair->single(i, j, itype, jtype,
                   rsq, factor_coul, factor_lj, fpair);

                // energy only computed once so tally full
                   amount
                // force tally is jgroup acting on igroup

                if (newton_pair || j < nlocal) {
                    array_atom[i][0] += eng;
                    if (ij_flag) {
                        array_atom[i][1] += delx*fpair;
                        array_atom[i][2] += dely*fpair;
                        array_atom[i][3] += delz*fpair;
                    }
                    if (ji_flag) {
                        array_atom[j][1] -= delx*fpair;
                        array_atom[j][2] -= dely*fpair;
                        array_atom[j][3] -= delz*fpair;
                    }

                    // energy computed twice so tally half
                       amount
                    // only tally force if I own igroup
                       atom
                }
                else {
                    array_atom[i][0] += 0.5*eng;
                    if (ij_flag) {
                        array_atom[i][1] += delx*fpair;
                        array_atom[i][2] += dely*fpair;
                        array_atom[i][3] += delz*fpair;
                    }
                }
            }
        }
    }
}
```

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

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 date for each atom. `size_peratom_cols` defines the number of data values to store for each atom. Also, we set the `scalar_flag` and `vector_flag` to `False`, since we do not wish to return a vector or scalar value.

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()` will resize the array to the number of atoms of concern, `nmax`. It does this using LAMMPS internal functions, which we will not care to describe here. Finally it calls upon functionsI ENDED HERE LAST TIME!

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

6.2 Least squares 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:

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

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

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

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.

```python
    def getAngle(self, v1, v2):
        'Computes angle between a vector and a line
            parallel to another vector'

        lv1 = np.linalg.norm(v1)
        lv2 = np.linalg.norm(v2)

        angle = np.arccos(np.dot(v1,v2)/(lv1*lv2))
        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.

Appendix A

Source code

A.1 compute_group_group.h

```cpp
#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"

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);
}


```

```
void ComputeGroupGroupAtom::compute_peratom()
{
    // grow array if necessary
    if (atom->nmax > nmax) {
        memory->destroy(carray);
        nmax = atom->nmax;
        memory->create(carray, nmax, size_peratom_cols, "group/group/atom:carray");
        array_atom = carray;
    }

    if (pairflag) pair_contribution();
    if (kspaceflag) kspace_contribution(); // This doesn't happen though. See compute_group_group.cpp constructor.
}


void ComputeGroupGroupAtom::pair_contribution()
{
    int i,j,ii,jj,inum,jnum,itype,jtype;
    double xtmp,ytmp,ztmp,delx,dely,delz;
    double rsq,eng,fpair,factor_coul,factor_lj;
    int *ilist,*jlist,*numneigh,**firstneigh;

    double **x = atom->x;
    int *type = atom->type;
    int *mask = atom->mask;
    int nlocal = atom->nlocal;
    double *special_coul = force->special_coul;
    double *special_lj = force->special_lj;
    int newton_pair = force->newton_pair;
    double *columns;

    // invoke half neighbor list (will copy or build if necessary)

    neighbor->build_one(list);

    inum = list->inum;
    ilist = list->ilist;
    numneigh = list->numneigh;
    firstneigh = list->firstneigh;

    // loop over neighbors of my atoms
    // skip if I,J are not in 2 groups


    for (ii = 0; ii < inum; ii++) {
        i = ilist[ii];

        // skip if atom I is not in either group
        if (!(mask[i] & groupbit || mask[i] & jgroupbit)) continue;

        xtmp = x[i][0];
        ytmp = x[i][1];
        ztmp = x[i][2];
        itype = type[i];
        jlist = firstneigh[i];
        jnum = numneigh[i];

        for (jj = 0; jj < jnum; jj++) {
            j = jlist[jj];
            factor_lj = special_lj[sbmask(j)];
            factor_coul = special_coul[sbmask(j)];
            j &= NEIGHMASK;

            // skip if atom J is not in either group
            if (!(mask[j] & groupbit || mask[j] & jgroupbit)) continue;

            int ij_flag = 0;
            int ji_flag = 0;
            if (mask[i] & groupbit && mask[j] & jgroupbit) ij_flag = 1;
            if (mask[j] & groupbit && mask[i] & jgroupbit) ji_flag = 1;

            // skip if atoms I,J are only in the same group
            if (!ij_flag && !ji_flag) continue;

            delx = xtmp - x[j][0];
            dely = ytmp - x[j][1];
            delz = ztmp - x[j][2];
            rsq = delx*delx + dely*dely + delz*delz;
            jtype = type[j];

            if (rsq < cutsq[itype][jtype]) {
                eng = pair->single(i, j, itype, jtype, rsq, factor_coul, factor_lj, fpair);

                // energy only computed once so tally full amount
                // force tally is jgroup acting on igroup

                if (newton_pair || j < nlocal) {
                    array_atom[i][0] += eng;
                    if (ij_flag) {
                        array_atom[i][1] += delx*fpair;
                        array_atom[i][2] += dely*fpair;
                        array_atom[i][3] += delz*fpair;
                    }
                    if (ji_flag) {
                        array_atom[j][1] -= delx*fpair;
                        array_atom[j][2] -= dely*fpair;
                        array_atom[j][3] -= delz*fpair;
                    }

                    // energy computed twice so tally half amount
                    // only tally force if I own igroup atom
                }
                else {
                    array_atom[i][0] += 0.5*eng;
                    if (ij_flag) {
                        array_atom[i][1] += delx*fpair;
                        array_atom[i][2] += dely*fpair;
                        array_atom[i][3] += delz*fpair;
                    }
                }
            }
        }
    }
}
```

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 / cm^dim