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 Hafreager

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?

Chapter 2

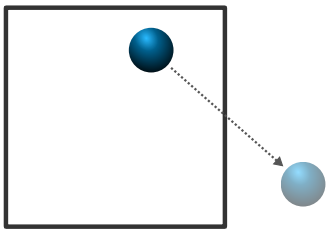
Molecular dynamics

... Molecular dynamics simulations are computationally expensive. And often scientists must balance statistical precision and CPU time. Technological improvements have provided the ability to simulate larger systems and/or at longer time frames.

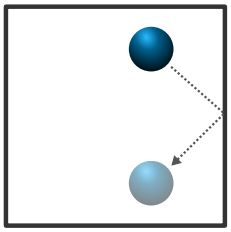
2.1 Boundary conditions

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.

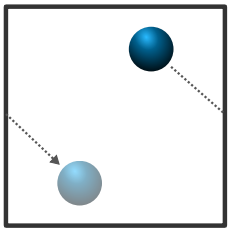
No boundary conditions. Particles are not subject to any special rules at any boundry. The system size may be regarded as infinite. This might be a reasonable choise when studying explosions for instance, or in experiments that is on a really short time scale.



Reflecting boundary conditions, which acts 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 areas of a material or materials that has a periodic structure.



2.1.1 Minimum image convention

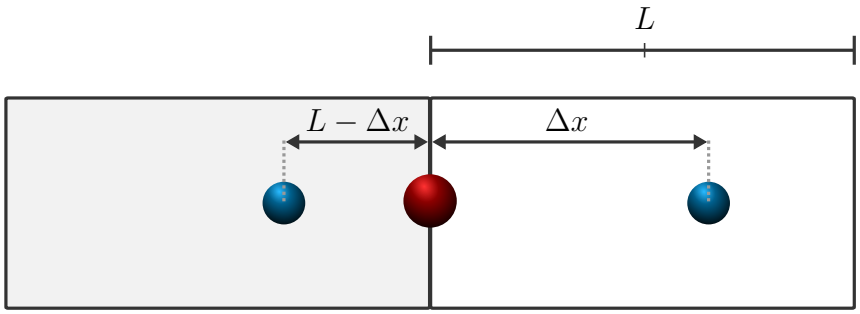


Figure 2.1: Minimum image convention in 1 dimension. If an atom is separated from the reference atom (red) by more than half the system length, $L/2$, the minimum image convention states that the distance between the two is the distance to the periodic copy, which is $L - \Delta x$. The white space indicates the system, while the gray area is periodic replications of the system. The replica on the other side is omitted, because it is not

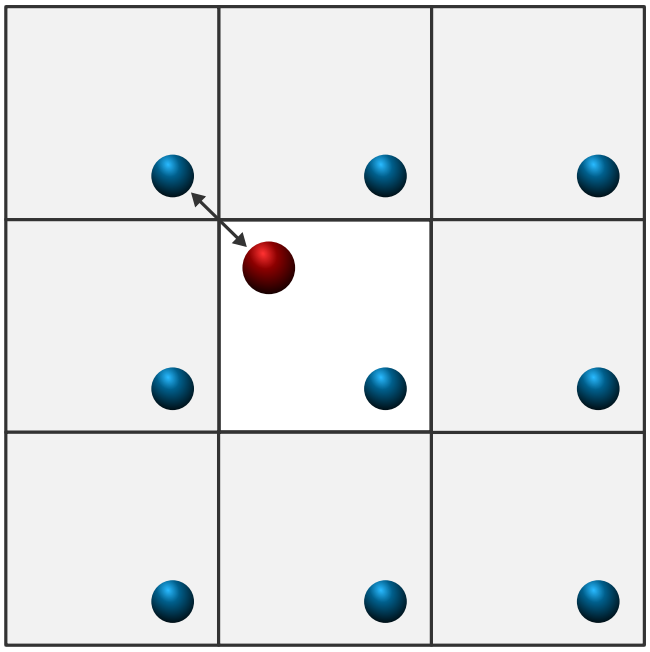


Figure 2.2: Minimum image convention in 2 dimensions. The position of an atom, with respect to the reference atom (red), is the position of the original or any of the replicated images of the other atom that has the shortest distance to the reference atom. The shortest distance in this case is marked with a two-sided 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.

Chapter 3

LAMMPS

LAMMPS stands 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.

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

```
make package-status
```

Installing specific packages is accomplished as shown below.

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

```
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 follows shown below.

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

The executable is now available as `lmp_serial` or `lmp_mpi` from anywhere.

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 if 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 with the following command.

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

In the example shown we installed the package manybody.

[2] http://brew.sh/

3.1.3 Windows

3.2 Running LAMMPS

3.3 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.

3.3.1 Cut-off

Depending on the potential in use, the forces become negligible at certain distances. For instance if one use the Lennard-Jones potential

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{3.1}$$

the contributions are practically zero for atoms positioned at a distance $r \geq 3\sigma$. Therefore, during a simulation we choose to only account for the contributions from atoms closer than this *cut-off* length. The number of contributions will then only depend on the density, which is an intensive[3] property. Thus, the number of computations is reduced to $\propto N$, which is an immense relief in computational expense!

In order to actually do this we must keep track of which atoms are within the cut-off length of each atom. This is achieved using cell lists and neighbor lists.

3.3.2 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 a specific 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. By neighboring cells we mean... This is illustrated in figure 3.1.

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

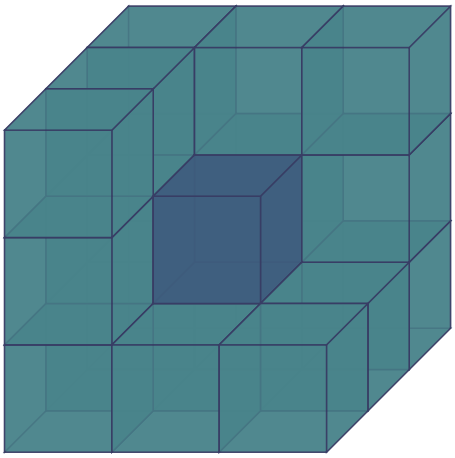


Figure 3.1: Illustrative figure of cells of concern when building the neighbor lists. The lighter cubes are neighboring cells; the darker cube is the cell containing the reference atom. The 7 cells in front of the dark cell are removed from the figure, but are also included.

3.3.3 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

$$S = \frac{T_s}{T_p}, \tag{3.2}$$

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 the author has 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 3.1 from the supercomputer using 1, 2, 4, 8, 16, 32 and 64 processors in the same fashion as shown in Listing 3.2. The resulting speedup of using the respective number of processors is plotted in figure 3.2.

```
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 3.1: 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 3.2: Command used to excecute the input script speedup.in on 8 parallel processors and set the filename variable to speedup.restart.

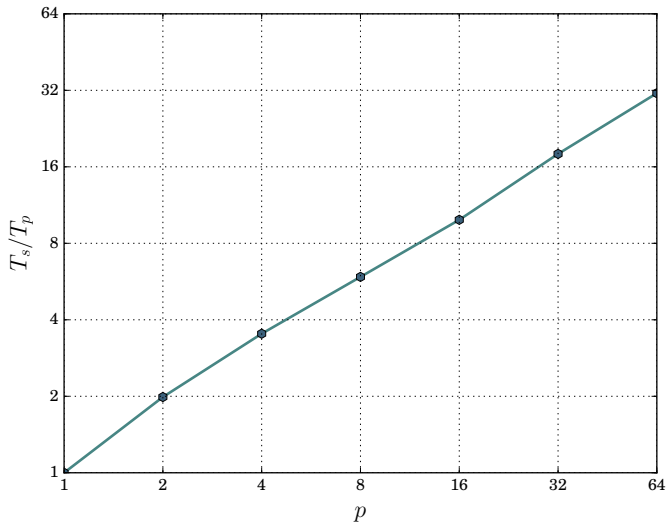


Figure 3.2: Speedup as a function of number of processors.

The result indicate that the speedup is in fact not simply $S = p$. Anyhow, we clearly see the advantage of using 64 processors 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 4

Setting up the system

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

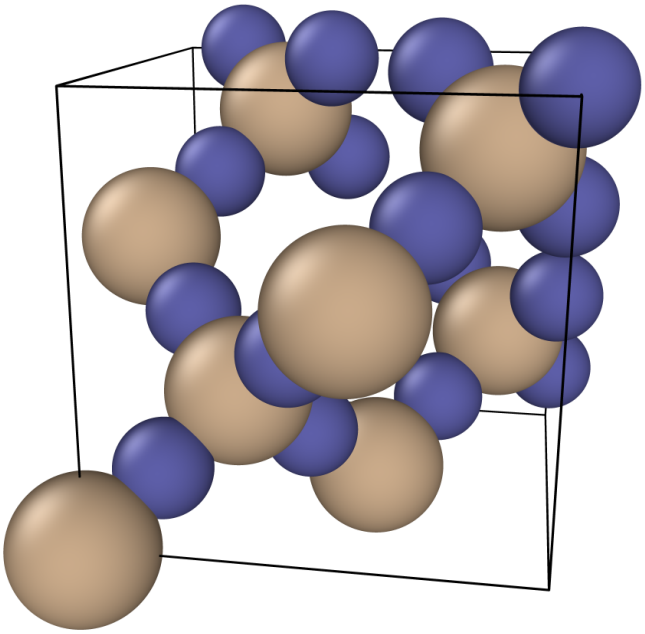


Figure 4.1: Unit cell of b-cristobalite. Tan and blue spheres represent silicon and oxygen atoms respectively. The unit cell is cubical, with edges of length 7.12Å.

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 consisting of 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.

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

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[3].

```
# 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.

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

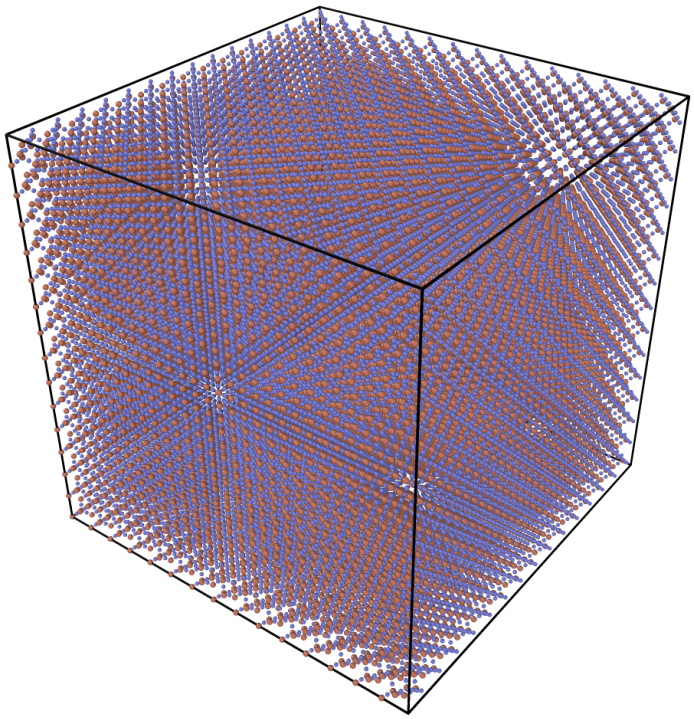


Figure 4.2: System built from $15 \times 15 \times 15$ unit cells of b-cristobalite.

We use the unit cell as building blocks, placing them concurrently until we have a crystal of the desired size. For our purpose, we generate a large cube of $15 \times 15 \times 15$ 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*[4]. 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 the melting point is know to be We expect the melting point to be about 1900K, so we start out at 1500K, compute the msd for N time steps, increase the temperature, equilibrate at this temperature, and repeat.

```
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.

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 by 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.

First, we define 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`.

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

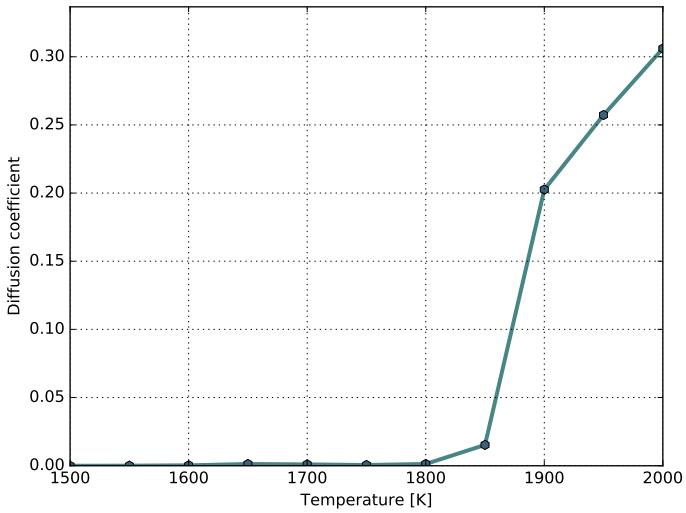


Figure 4.3: Mean square displacement as a function of temperature. The face transition occurs where we see a rapid change in the behavior.

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 bothRegion
```

Listing 4.3: 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 is redundant. However, at a later stage we will utilize them and this is an appropriate place for them to be defined.

The appliance of the script in Listing 4.3 on the system shown in figure 4.2 is shown in figure 4.4, where our perspective is looking along the y-axis.

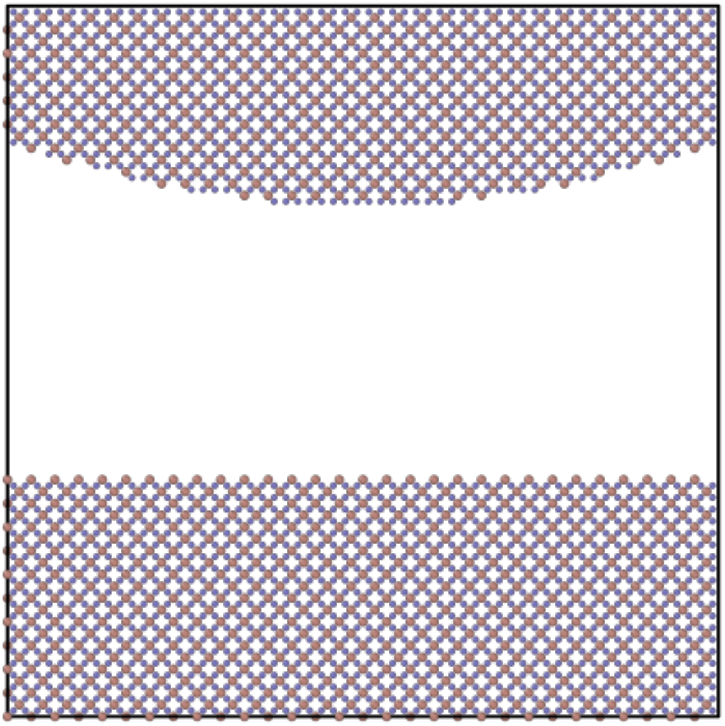


Figure 4.4: xz-perspective on a system built from $15 \times 15 \times 15$ unit cells of b-cristobalite, with certain regions carves out. This is a result from applying Listing 4.3 to the system shown in figure 4.2. The top shape is a sphere cap, while the bottom is a slab.

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.

```
1 fix freezeID groupID setforce 0 0 0
```

```
2 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.

```
1 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

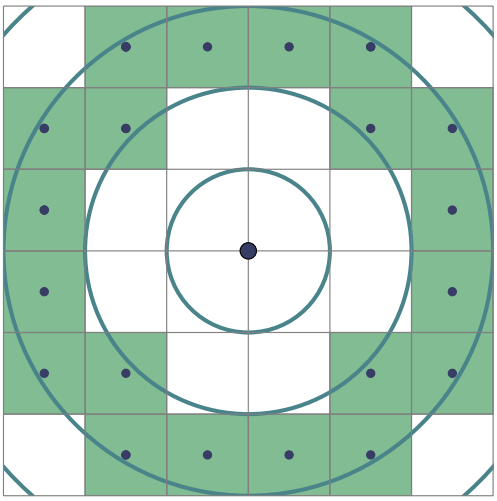


Figure 5.1: Coarse radial binning. The value appointed to the radial bin is the average of the weights of the cells who's center is within the bin. The cells with center within the third radial bin are colored, and their centers drawn.

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 the cells that actually are withing the bin, multiplied by the weight associated with the cell, and average these contributions. 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 and 5.3. The figure clearly shows 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!

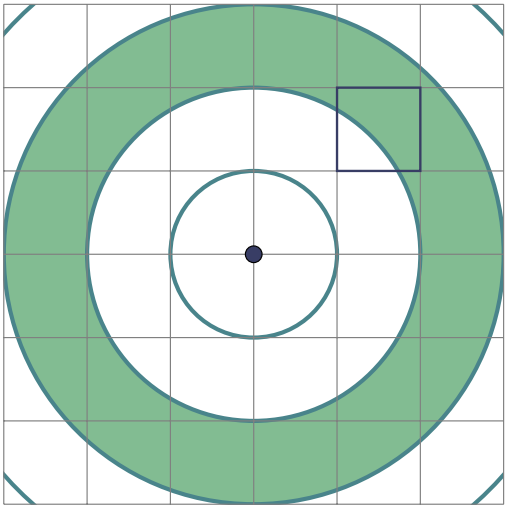


Figure 5.2: Radial binning based on weighted contributions of intersecting cells. The third radial bin is colored, and its value will be the average of the intersecting cells weight times the fraction of the cells area that intersects the bin. A close-up of the outlined cell, is shown in figure 5.3.

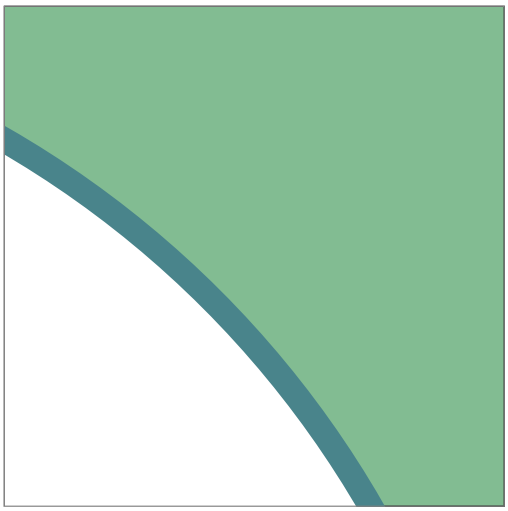


Figure 5.3: Close-up of outlined cell in figure 5.2. The contribution from this cell will be it weight multiplied by the fraction of its total area that is colored. I might use this figure to explain the procedure.

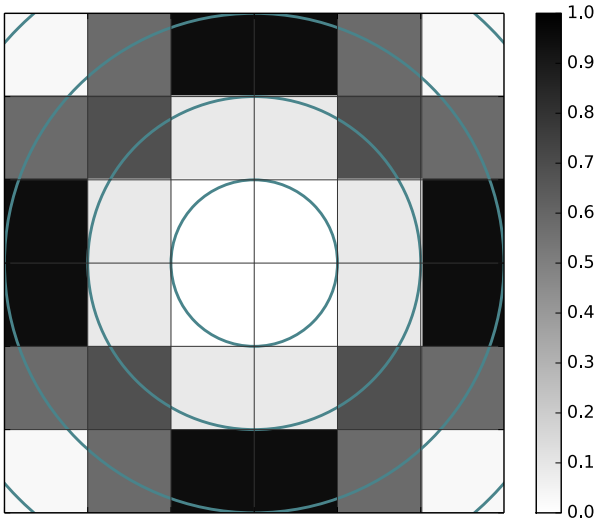


Figure 5.4: Fraction of cell's area that are within the third radial bin.

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 already, and chances are they 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.

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.

```
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 {
14  public:
15   ComputeGroupGroupAtom(class LAMMPS *, int, char **);
16   ~ComputeGroupGroupAtom();
17   void compute_peratom() override;
18   int nmax;
```

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

```
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`.

`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` $\times$ 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 using namespace LAMMPS_NS;
21 using namespace MathConst;
22
23 #define SMALL 0.00001
24
25 ComputeGroupGroupAtom::ComputeGroupGroupAtom(LAMMPS *lmp,
       int narg, char **arg) :
26     ComputeGroupGroup(lmp, narg, arg),
27     carray(NULL),
28     nmax(0)
```

```
29 {
30     if (narg < 4) error->all(FLERR,"Illegal compute
           group/group command");
31
32     peratom_flag      = 1;  // Indicating a peratom compute
33     size_peratom_cols = 4;  // # of Columns per atom.
34     extarray          = 0;  // 0/1 if global array is all
           intensive/extensive
35     scalar_flag       = 0;
36     vector_flag       = 0;
37 }
38
39
40 ComputeGroupGroupAtom::~ComputeGroupGroupAtom()
41 {
42     memory->destroy(carray);
43 }
44
45
46 void ComputeGroupGroupAtom::compute_peratom()
47 {
48     // grow array if necessary
49     if (atom->nmax > nmax) {
50         memory->destroy(carray);
51         nmax = atom->nmax;
52         memory->create(carray, nmax, size_peratom_cols,
               "group/group/atom:carray");
53         array_atom = carray;
54     }
55
56     if (pairflag) pair_contribution();
57     if (kspaceflag) kspace_contribution(); // This doesn't
           happen though. See compute_group_group.cpp
           constructor.
58 }
59
60
61 void ComputeGroupGroupAtom::pair_contribution()
62 {
63     int i,j,ii,jj,inum,jnum,itype,jtype;
64     double xtmp,ytmp,ztmp,delx,dely,delz;
65     double rsq,eng,fpair,factor_coul,factor_lj;
66     int *ilist,*jlist,*numneigh,**firstneigh;
67
68     double **x = atom->x;
69     int *type = atom->type;
70     int *mask = atom->mask;
71     int nlocal = atom->nlocal;
72     double *special_coul = force->special_coul;
73     double *special_lj = force->special_lj;
74     int newton_pair = force->newton_pair;
```

```
75      double *columns;
76
77      // invoke half neighbor list (will copy or build if
            necessary)
78
79      neighbor->build_one(list);
80
81      inum = list->inum;
82      ilist = list->ilist;
83      numneigh = list->numneigh;
84      firstneigh = list->firstneigh;
85
86      // loop over neighbors of my atoms
87      // skip if I,J are not in 2 groups
88
89
90      for (ii = 0; ii < inum; ii++) {
91          i = ilist[ii];
92
93          // skip if atom I is not in either group
94          if (!(mask[i] & groupbit || mask[i] & jgroupbit))
                continue;
95
96          xtmp = x[i][0];
97          ytmp = x[i][1];
98          ztmp = x[i][2];
99          itype = type[i];
100         jlist = firstneigh[i];
101         jnum = numneigh[i];
102
103         for (jj = 0; jj < jnum; jj++) {
104             j = jlist[jj];
105             factor_lj = special_lj[sbmask(j)];
106             factor_coul = special_coul[sbmask(j)];
107             j &= NEIGHMASK;
108
109             // skip if atom J is not in either group
110             if (!(mask[j] & groupbit || mask[j] &
                    jgroupbit)) continue;
111
112             int ij_flag = 0;
113             int ji_flag = 0;
114             if (mask[i] & groupbit && mask[j] & jgroupbit)
                    ij_flag = 1;
115             if (mask[j] & groupbit && mask[i] & jgroupbit)
                    ji_flag = 1;
116
117             // skip if atoms I,J are only in the same group
118             if (!ij_flag && !ji_flag) continue;
119
120             delx = xtmp - x[j][0];
```

```
121                 dely = ytmp - x[j][1];
122                 delz = ztmp - x[j][2];
123                 rsq = delx*delx + dely*dely + delz*delz;
124                 jtype = type[j];
125
126                 if (rsq < cutsq[itype][jtype]) {
127                     eng = pair->single(i, j, itype, jtype,
                            rsq, factor_coul, factor_lj, fpair);
128
129                     // energy only computed once so tally full
                            amount
130                     // force tally is jgroup acting on igroup
131
132                     if (newton_pair || j < nlocal) {
133                         array_atom[i][0] += eng;
134                         if (ij_flag) {
135                             array_atom[i][1] += delx*fpair;
136                             array_atom[i][2] += dely*fpair;
137                             array_atom[i][3] += delz*fpair;
138                         }
139                         if (ji_flag) {
140                             array_atom[j][1] -= delx*fpair;
141                             array_atom[j][2] -= dely*fpair;
142                             array_atom[j][3] -= delz*fpair;
143                         }
144
145                         // energy computed twice so tally half
                                amount
146                         // only tally force if I own igroup
                                atom
147                     }
148                     else {
149                         array_atom[i][0] += 0.5*eng;
150                         if (ij_flag) {
151                             array_atom[i][1] += delx*fpair;
152                             array_atom[i][2] += dely*fpair;
153                             array_atom[i][3] += delz*fpair;
154                         }
155                     }
156                 }
157             }
158         }
159 }
```

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:

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

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

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

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 $\mathbf{n} = [1, a, b]$. 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.

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"

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 void ComputeGroupGroupAtom::compute_peratom()
47 {
48     // grow array if necessary
49     if (atom->nmax > nmax) {
50         memory->destroy(carray);
51         nmax = atom->nmax;
52         memory->create(carray, nmax, size_peratom_cols,
               "group/group/atom:carray");
53         array_atom = carray;
54     }
55
56     if (pairflag) pair_contribution();
57     if (kspaceflag) kspace_contribution(); // This doesn't
           happen though. See compute_group_group.cpp
           constructor.
58 }
59
60
61 void ComputeGroupGroupAtom::pair_contribution()
62 {
63     int i,j,ii,jj,inum,jnum,itype,jtype;
64     double xtmp,ytmp,ztmp,delx,dely,delz;
65     double rsq,eng,fpair,factor_coul,factor_lj;
66     int *ilist,*jlist,*numneigh,**firstneigh;
67
68     double **x = atom->x;
69     int *type = atom->type;
70     int *mask = atom->mask;
71     int nlocal = atom->nlocal;
72     double *special_coul = force->special_coul;
73     double *special_lj = force->special_lj;
74     int newton_pair = force->newton_pair;
75     double *columns;
76
77     // invoke half neighbor list (will copy or build if
           necessary)
78
79     neighbor->build_one(list);
80
81     inum = list->inum;
82     ilist = list->ilist;
83     numneigh = list->numneigh;
84     firstneigh = list->firstneigh;
85
86     // loop over neighbors of my atoms
87     // skip if I,J are not in 2 groups
88
89
90     for (ii = 0; ii < inum; ii++) {
91         i = ilist[ii];
92
```

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

```
138                         }
139                         if (ji_flag) {
140                             array_atom[j][1] -= delx*fpair;
141                             array_atom[j][2] -= dely*fpair;
142                             array_atom[j][3] -= delz*fpair;
143                         }
144
145                         // energy computed twice so tally half
                               amount
146                         // only tally force if I own igroup
                               atom
147                     }
148                     else {
149                         array_atom[i][0] += 0.5*eng;
150                         if (ij_flag) {
151                             array_atom[i][1] += delx*fpair;
152                             array_atom[i][2] += dely*fpair;
153                             array_atom[i][3] += delz*fpair;
154                         }
155                     }
156                 }
157             }
158         }
159 }
```