Introductory LAMMPS Hands-on Tutorial

Prepared by:
Kyle Hall (kylewmhall@gmail.com)
&
Axel Kohlmeyer (akohlmey@gmail.com)

1 Introduction

This tutorial provides a brief overview of using LAMMPS to create and simulate atomic and molecular systems. This document will walk you through a series of LAMMPS commands and sample simulations while focusing on two particular systems, an pure argon system and a solvated polyethylene chain. Throughout this document, you will find a series of **Tasks**. These are places for you to expand your knowledge and try running some simple simulations. The tasks build on each other, so it is recommended that you go through the tutorial sequentially. This LAMMPS tutorial requires you to make extensive use of the LAMMPS documentation/manual, which is provided locally in PDF format and can be accessed at via the internet at http://lammps.sandia.gov/doc/Manual.html

You have been provided with a Windows Installer Package and a Linux virtual machine where you have compiled a custom LAMMPS executable from source. The following exercises should work equally well with either executable. For the customized Linux executable, you may need to include additional packages and re-compile. Unpack the archive with the provided inputs and work your way through this document. Inside the archive, you will see two folders: argon and solvatedPE. These folders contain the files that you will need for completing Sections 2-4 of this tutorial. As you complete each task, put its input script in a new folder and run LAMMPS from that folder. That will allow you to keep the input scripts separate and compare their results.

Note: When you run a LAMMPS simulation, LAMMPS overwrites files, so running two simulations in the same folder generally means that you will at least partially overwrite/lose the results of one of the simulations.

You can run LAMMPS from the command line using:

```
lmp_serial -in MyInputFile
```

or

```
lmp_mpi -in MyInputFile
```

or

```
lmp -in MyInputFile
```

where `lmp_serial`, `lmp_mpi`, or `lmp` is the name of the specific LAMMPS executable and `MyInputFile` is the input script that you want LAMMPS to execute.

This tutorial is meant to be completed from the command line, and assumes that you are comfortable modifying text files from the command line using *vi* or *nano* or similar text editors.

2 Constructing and Running a Basic Argon System

2.1 LAMMPS Script Basics & Building a System from Scratch

This subsection introduces you to some fundamental LAMMPS commands in the context of simulating condensed argon. The following walks you through the commands appearing in the input file in.argon in.argon is located in the folder `argon`

2.1.1 Specifying Characteristic of your LAMMPS Simulation

Running simulations involves many choices beyond which system you want to study. Sometimes, these decisions are a matter of preference, and might not dramatically change a simulation's results. For example, you can use many different units systems with LAMMPS (e.g., reduced units, metric, cgs). If you properly convert input files and scripts, you could probably simulate a system using several different units systems; however, one alternative is often easier or more desirable. In this Section, you will simulate argon. You could use reduced units for the simulation, but let's stick to what LAMMPS calls *real* units where distances are measured in Angstrom, time in femtoseconds, energy in kcal/mol, etc. You can set the units for your simulation using the *units* command.

```
units    real
```

This is the first command in in.argon. Look at the LAMMPS documentation both for more information about the *units* command and for details about what is meant by *real* units.

Most choices, if not all, will substantially impact your simulation results, so great care should be taken when making methodological choices at the start of a project. For example, consider boundary conditions (i.e., the properties of the edge of your simulation box). Often we want to simulate bulk (infinite) systems, but can only conduct a very small, finite-size simulation. To overcome this, we can use periodic boundary conditions. This means that space is wrapped and continuous. For each direction, if a particle exits one side of the box (e.g., the right side) then it enters via the opposite side of box (e.g., the left side). The following *boundary* command enforces this behaviour for the x, y and z directions of your soon-to-be LAMMPS simulation cell, and this is the second command in in.argon.

```
boundary    p p p
```

You could also choose to simulate a non-periodic finite system, but that alternative system would have very different behaviour. Moreover, when you use finite boundary conditions, you must be very careful with how you setup and carry out your LAMMPS simulation, or you will encounter errors. This is not a shortcoming of LAMMPS, but reflects the difficulties associated with simulating finite systems. You should carefully consider all methodological details of a simulation before starting any calculations.

You need to tell LAMMPS what types of attributes it should expect, store and print for the particles that will be in your system. You tell LAMMPS this information using the *atom_style* command. In this Section, you will simulate an argon system, so the *atomic* option is reasonable.

```
atom_style    atomic
```

Based on this command, LAMMPS will now store the velocities, coordinates and atom type for the atoms in your system. But you do not currently have a system or atoms…
You now need to construct your argon system.

2.1.2 Constructing a system with atoms on a 3D lattice

One of the easiest ways to create a system is to place particles (e.g., atoms or molecules) on a lattice – a repeating ordered pattern in 3D space. This is not appropriate for all systems and research questions, but it is a reasonable starting point for your argon simulation. Creating a system where particles occupy space according to a lattice involves several steps in LAMMPS.

Step 1: Constructing a lattice
The *lattice* command provides a quick way to specify some standard lattice patterns. Here, you will create a face-centered cubic (*fcc*) lattice to get started with a lattice constant of 6.0 Å.

```
lattice    fcc 6.0
```

Because you have already specified that LAMMPS use *real* units, LAMMPS knows to interpret *6.0* in the *lattice* command as 6.0 Å.

Step 2: Constructing the simulation box
The *region* command allows you to specify geometrical regions. Because you have already defined a lattice, you can use the *block* option to make a region that it some multiple of the specified lattice constant.

```
region     boxID block 0.1 10.1  0.1 10.1  0.1 10.1
```

The pairs of numbers indicate the lower and upper limits along the x, y and z axes. For example, in the box command, the extent of the region you are defining is 0.6 Å (0.1*6.0 Å) to 60.6 Å (10.1*6.0 Å) along x given that the lattice constant is 6.0 Å.

Between *region* and *block* is an identifier *boxID*. A number of LAMMPS commands must be assigned identifiers. User-assigned identifiers generally appear after a command's name and before the command's subsequent options/flags. Identifiers allow subsequent commands to the use the lists or data resulting from the labelled command. For example, the *region* command only defines a region of space, and does not construct our simulation box. By passing the identifier *boxID* to *create_box*, *create_box* can create a simulation box consistent the *region* command labelled *boxID*.

```
create_box    1 boxID
```

Unlike *region*, *create_box* does not have an identifier. The *1* after *create_box* instead indicates the number of different atom types that will be present in the *boxID* region.

Step 3: Filling the simulation box with particles
While *region* and *create_box* can create a simulation box (the volume that you will simulate), you still do not have any atoms in the simulation. In LAMMPS, there are many ways to fill a simulation cell with atoms or molecules, and many 3rd-party tools are available for creating more complex systems. One of the simplest commands for filling a simulation cell with particles (either atoms or molecules) is *create_atoms*. Like *create_box*, it lacks an identifier.

```
create_atoms    1 box
```

The number after *create_atoms* is the number of different atom types to be added to the simulation box. Here, the value is *1* because you only have one atom type; you will be simulating pure argon. Specifying the *box* option tells LAMMPS to fill your newly created box according to the lattice that you previously specified. You can also use *create_atoms* create a random distribution of particles or a single particle in a simulation cell.

Note: Nasty problems can arise when using create_atoms to position atoms on a lattice, though you should not encounter them as you go through this tutorial. Take a look at the unit cell for a face-center cubic lattice before proceeding in order to help you see why.

If your box perfectly aligns with lattice's particle positions, then there is an ambiguity about what to do with particles appearing right at the edge of the unit cell. This situation illustrated below using a simple cubic lattice and two unit cells.

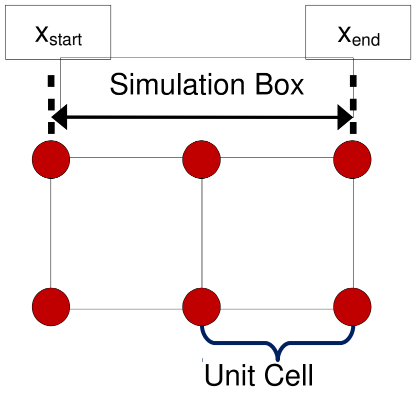

Figure 1. A simulation box defined on a simple cubic lattice that is two unit cells wide, and spans x_{start} to x_{end} along the x axis.

You might think that LAMMPS would need to add atoms at both x_{start} and x_{end} to construct the system in Figure 1. However, your simulation is periodic, so x_{start} and x_{end} are actually the same position. In turn, LAMMPS has to sort our whether to put a particle at x_{start} or x_{end}, and make sure that it does not add particles at both positions. For cubic lattices, LAMMPS can generally sort this out, but it becomes harder with other lattice types, particularly custom lattices. If LAMMPS does put two particles in the same position, then there will definitely be problems with your simulation (e.g., exploding simulations, extremely large forces, etc.). To avoid these problems, it is best to specify your simulation box so that lattice particle positions do not coincide with the boundaries of your simulation box (see Figure 2). This is why the *region block* command in `in.argon` specifies each axis range as 0.1 to 10.1, and not 0.0 to 10.0.

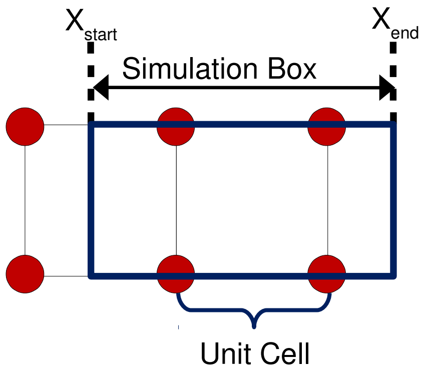

Figure 2. A simulation box defined on a simple cubic lattice that is two unit cells wide. The simulation box is shifted with respect to the underlying lattice.

Step 4: Specify the properties of the atoms
Up to this point you have established atomic positions within your system for atom type *1*, but have not specified any information about atom type *1*, such as its atomic mass or how the atoms of type *1* will interact. LAMMPS needs that information to perform your simulation, and your choices here strongly influence whether the atom type *1* accurately represents argon atoms. Some relevant parameter and interaction specifications are provided for you in `in.argon` on lines 13-16. This information is delimited in the input file by comment lines; LAMMPS interprets lines beginning with # as comments and ignores them.

Take several moments to analyze the meaning of lines 13-16 in in.argon using the LAMMPS documentation. You may find it helpful to annotate a physical copy of in.argon as you do this. This will help you start familiarizing yourself both with the logic and syntax of LAMMPS commands, as well as the LAMMPS documentation; two very important things for people wanting to use LAMMPS.

2.1.3 Running a simulation

Now that you have a simulation box full of atoms, you need to specify how positions and velocities will be updated as your simulation progresses, this can be done using *fix* commands such as…

```
fix ensembleFix all nve
```

In LAMMPS, *fix* commands specify operations that occur for groups of particles within a system as that system temporally evolves (or is energetically minimized). The basic format of a *fix* command is a fix identifier, a label indicating the group of particles on which the fix operates, and the style (i.e., nature) of the fix. In the above example, these are *ensembleFix*, *all* and *nve*, respectively. The group specifier *all* indicates that this fix will apply to all atom types in your system. The style *nve* indicates that particle positions and velocities are to be updated in a way consistent with the microcanonical ensemble (NVE ensemble). More specifically, LAMMPS will use discretized integration of Newton's equations of motion to evolve the system.

Note: Even though you choose the fix nve in your input script, that does not necessarily mean that you are performing a microcanonical (NVE) simulation. If you have additional fixes or operations in your LAMMPS input file such as thermostating, you can perturb the dynamics of your particle away from those of the mircocanonical ensemble.

Now that you have specified how LAMMPS should update particle positions, you now just need to specify how long you want your simulation to run. This is done with the *run* command.

```
run    4000
```

The number after *run* indicates how many times you want the simulation to update the positions of the particles (i.e., the number of timesteps in the simulation).

Now try running your argon simulation by using `in.argon` as your input script, and executing `lmp_serial –in in.argon` from the terminal in the directory containing `in.argon.`

2.2 Argon Simulation Debrief

2.2.1 Your input is underspecified

You initial argon simulation should have run without any errors. You should be concerned.

Your input file `in.argon` lacks a lot of important information that is necessary to conduct a molecular simulation. For example, you specified how many times LAMMPS should update the positions of the particles in the argon system (4000, to be exact). However, to integrate the equations of motion and thus update particle positions, LAMMPS needs to know the simulation's timestep – the window of simulation time between each position update (often one to several femtoseconds for classical Molecular Dynamics simulations). A simulation's timestep is very important. If it is too large, it will be difficult to evolve the system and artifacts are likely to occur since a particle's current position, velocity and forces become unreliable indicators of its future position. Alternatively, if the timestep is too small, then LAMMPS spends a lot of time calculating minor changes, and the productivity of the simulation is reduced. However, you did not have to worry about selecting a timestep for the argon simulation. This is because LAMMPS has a default of 1.0 fs when using *real* units.

LAMMPS has many pre-set defaults that you can either use or override. The simulation's timestep is one example. Another example is how the particle positions are temporally evolved. There are different numerical integration schemes that can be used to update particle positions during a simulation, and this can be set in LAMMPS using the *run_style* command. Both the simulation's timestep and its numerical integration scheme are important factors when starting a simulation. The danger with relying on defaults is that you might be leaving important methodological choices to the whims of LAMMPS defaults, and those defaults might be poor choices for your systems.

Task #1: Make a copy of `in.argon`, and call it `in.task-01`. In `in.task-01`, set the simulation's timestep to 2.0 fs using the *timestep* command. Consult the LAMMPS documentation as need. Note that this command needs to appear before the *run* command. Try re-running your simulation with `in.task-01` to make sure that your revised script is working properly.

2.2.2 Why did I not get more information?

Your initial argon simulation yielded a `log.lammps` file. Take a moment to look at this file. Can you see from this file how LAMMPS went sequentially through the lines in your LAMMPS input script?

While the `log.lammps` file provides a small amount of detail, you will need much more information if you want to understand the argon system that you are simulating. You might want system configurations (i.e., how the particles are distributed in your simulation box), thermodynamic information as the system evolves, or even on-the-fly movies or snapshots of the system. LAMMPS can provide all of this and more, but it only does so if you explicitly instruct it to save specific types of data.

Task #2: Copy `in.task-01`, and call it `in.task-02`. Adapt `in.task-02` so that LAMMPS will perform the following operations when you simulate your argon system with `in.task-02`

- Write the system's thermodynamic information (e.g., T and P) every 200.0 fs using the *thermo* command. Look in the LAMMPS documentation at option *one* for the *thermo_style* command if you want to know precisely what will be written out.
- Create snapshots of the simulation's evolution on-the-fly using images 2000.0 fs apart. Hint: you need to replace the *XXX*'s in the following command. Consider how argon*.jpg is treated by LAMMPS since this it is a common strategy for specifying write-outs in LAMMPS.

```
dump        XX XX image XX argon*.jpg type type
```

Task #2 Follow-up: Look at the first and last jpeg images of your simulation. You can view them them in the graphical interface with an image viewer or a web browser. How are the images similar and/or different? Now, look at `log.lammps`, which should now have much more information in it. Are your insights from the images consistent with the system's properties during the simulation?

3 Putting the dynamics in Molecular Dynamics

When you ran your adapted `in.task-02` input script, you should have seen that the argon atoms essentially did not move and that your simulation had a temperature very close to 0 K. You properly created an argon system with particles periodically arranged with no initial velocities, so there really is no reason for them to move. Your results are physically reasonable given the system you created and simulated, but not particularly interesting or physically relevant.

3.1 Adding velocities and seeing things moving

The *velocity* command allows you to assign initial velocities to the particles in your system after you have created a system or read in a system configuration).

Task #3: Copy `in.task-02` to make `in.task-03`, and add the following two lines immediately after the argon parameters portion of the script (i.e., lines 11-17).

```
variable     temp   index   100.0
velocity     all create ${temp} 11235712 mom yes rot yes dist gaussian
```

In the first line, the *variable* command assigns 100.0 to the variable *temp*, which you can then access later in your input script, such as in the following *velocity* command. Here, the *velocity* command is, for *all* particles, generating velocities (hence, the *create* option) consistent with a temperature of *temp* (100 K). The additional options are modifying the behaviour of *velocity*. See the LAMMPS documentation for more details.

By putting *variable* commands at the start of your input script and using them throughout a longer LAMMPS input script to specify information (e.g., temperature), you can quickly modify your scripts as need arises to, for example, do simulations at multiple temperatures, without going through your entire LAMMPS script and doing many low-level, repetitious changes.

Run an argon simulation using your newly created `in.task-03` input script.

Task #3 Follow-up: Look again at the first and last jpeg images of your simulations to confirm that the particles are now moving during your simulation, and then look at the log.lammps file. What is the temperature and pressure at the end of your simulation? Does the system stay at 100 K once you start your simulation?

3.2 Thermostats and Barostats

The argon particles in your system are now moving, but the temperature of the system does not stay at 100 K, but decreases. The system's pressure also corresponds to a large negative value (hundreds of atm of tension). Your are performing an NVE simulation so the total energy of the system should be conserved and not temperature. As well, conducting a NVE simulation means that the volume of the box is fixed, so our system's density and pressure will depend on how you initially set up the system. A system's temperature and pressure can be regulated using thermostats and barostats, respectively. You will now add commands to control temperature and pressure in your argon simulation.

3.2.1 Thermostating - Langevin

Using a Langevin thermostat is a reasonable way to thermally equilibrate a system. Briefly, a Langevin thermostat operates by modifying the forces used to calculate particle displacements so that they now include: **1)** a drag term that works to decelerate particles, and **2)** a random force term emulating a particle's interactions with an artificial, nonexistent bath of particles at the desired temperature. The interactions between particles within the simulation are still calculated; there are just now additional forces at play. Through the balance between drag and random force effects, the system will eventually arrive at the desired temperature. You can add a Langevin thermostat to a LAMMPS simulation using the *fix langevin* command.

Task #4: Make a copy of `in.task-03`, and call it `in.task-04`. Add a Langevin thermostat to your argon simulation by adding the following *fix langevin* command to `in.task-04` after your *fix nve* command. Use the LAMMPS documentation to decide what to replace the *XXX*'s with, and remember that you are trying to simulate your argon system at 100.0 K. You have already defined the variable *temp* to represent 100.0, so you should use it to complete the *fix langevin* command.

```
fix    langevinFix XXX langevin XXX XXX  100.0 1111111 zero yes
```

Run `in.task-04` when you are ready.

Task #4 Follow-up: Look at log.lammps to see the temperature of your argon simulation as a function of time. How does the instantaneous temperature of your simulation compare to your set point of 100.0 K? What does this reveal about Langevin thermostats? Do you understand the meaning of the end portion of the *fix langevin* command that you added to `in.task-04` (specifically, *100.0 1111111 zero yes*)?

Notice how you added a Langevin thermostat using a *fix langevin* while still relying on *fix nve*. This does not mean that you are performing a microcanonical (NVE) simulation with a thermostat. Recall that *fix nve* dictates how LAMMPS integrates the system's equation of motion such that in the absence of additional commands (e.g., some other fixes) your simulation would be a microcanonical (NVE). *fix langevin* achieves an constant temperature (isothermal) simulation by adding additional forces to the particles in your system that are not due to interactions between your system's particles, and thus necessarily perturbs the system away from being a microcanonical simulation. However, *fix langevin* does not change how the LAMMPS integrates the equations of motions.

3.2.2 Thermostating - Nosé-Hoover

Another popular way to achieve isothermal simulations is by using a Nosé-Hoover thermostat. When used in conjunction with constant volume, your simulation will be operating in the NVT (canonical) ensemble, assuming that you do not have additional commands that perturb the system from canonical sampling. The Nosé-Hoover thermostat operates by adding additional degrees of freedom to a system, and altering the particle equations of motion to achieve a constant temperature simulation. In turn, you can no longer use the equations of motions invoked by the *fix nve* command that you have been using so far.

```
fix    ensembleFix all nve
```

You must replace the above command with another *fix* command that will invoke the NVT Nosé-Hoover equations of motion.

```
fix    ensembleFixNVT all nvt temp ${temp} ${temp} 200.0
```

Task #5: Make another copy of `in.task-03`, and call it `in.task-05`. Substitute the above *nvt fix* command into `in.task-05`, and rerun your simulation. How does the system's behavior change? How does the evolution of the system's temperature compare to that of your Langevin simulation during the first few hundred timesteps? Which thermostat seems to be a better choice for initially equilibrating your argon system?

3.2.3 Adding a barostat: Bersendsen Barostat

You have managed to control the temperature of your argon using two different thermostats. Now, you will control the system's pressure using a Berendsen barostat. A Berendsen barostat controls pressure by rescaling your simulation cell and the particles within it so that the pressure of your system relaxes toward your specified set-point. A Berendsen barostat does not change how the particles' equations of motion are solved, rather it adjusts the positions of the particles every timestep. Thus, as with the Langevin thermostat, you can use a *fix nve* command to tell LAMMPS how to integrate the equations of motion, and add a *fix press/berendsen* command to invoke Berendsen barostat.

Task #6: Make a copy of `in.task-04` (your script from Task #4), and call it `in.task-06`. You will now try to pressurize your argon system at 1.0 atm. Complete the following *fix press/berendsen* command, and add it to `in.task-06` after your *fix langevin* command. Make sure that you understand what each option means. When you are starting your own simulations, you will probably have to change the *fix press/berendsen* options, including default settings that are not covered in this tutorial.

```
fix    XXX XXX press/berendsen iso XXX XXX 2000.0
```

Task #6 Follow-up: Look at your log.lammps file. If you have properly added the Berendsen barostat command, the pressure of your argon system should rapidly increase at the start of your simulation from a very large negative pressure to around 1.0 atm by the end of the simulation. Why is the pressure oscillating? How do the pressure oscillations compare to the system's volume oscillations?

While the Beresend barostat is useful for rapidly equilibrating a system to a desired pressure, it does not provide correct NPT sampling (i.e., constant pressure, constant temperature sampling for a system containing a constant number of particles). To obtain proper NPT sampling, you should consider using a Nosé-Hoover thermostat <u>and</u> a Nosé-Hoover barostat, which can be accessed using the *fix npt* command (instead of *fix nve* or *fix nvt*).

Optional Task: To test your knowledge and gain more familiarity with LAMMPS, convert `in.task-06` to an NPT simulation that uses the *fix npt* command. Using a Nosé-Hoover barostat is not necessarily a good way to equilibrate a system; however, knowing how to conduct a NPT simulation with *fix npt* will be useful when you want to perform equilibrium simulations. Hint: As with the Berendsen barostat, you should probably using an <u>isotropic</u> barostat for your simulation. This was indicated by the *iso* option in the *fix press/berendsen* command.

4 Simulating a Molecular System

So far, you have been simulating an atomic system. Now, you will start simulating a molecular system, specifically a single polyethylene chain in a solvent. This section will focus on understanding and modifying the LAMMPS script entitled `in.PE_Simulation`. `in.PE_Simulation` is located in `solvatedPE` along with the other files needed for this Section (`data.SolvatedChain` and `param.PE`). When running your revised versions of `in.PE_Simulation`, you will need to move or copy `data.SolvatedChain` and `param.PE` into the folder where you are running your updated script.

4.1 Input/Output Revisited

Instead of generating the system's initial configuration from scratch, you have been provided with a LAMMPS *data* file, `data.SolvatedChain`. A LAMMPS *data* file contains the positions of the particles in your system and necessary connectivity specifications (e.g., bonds and angles). *data* files can also contain particle velocities and force field information (e.g., Lennard-Jones ε and σ parameters, or bond stretching potentials) depending on how the *data* file was created. You can read in a *data* file using the *read_data* command. Below is the *read_data* command that appears in `in.PE_Simulation` under the "Loading System Info" comment block. Recall that LAMMPS interprets script lines beginning with # as comments.

```
read_data    data.SolvatedChain
```

In this example, `data.SolvatedChain` does **not** contain any force field information; this information is stored at the top of in.PE_Simulation and in param.SolvatedPE. At the top of `in.PE_Simulation`, lines 5 to 9 specify the characteristics of the forcefield that will be used for your polyethylene simulation. For example, this script using a different option for *atom_style*.

```
atom_style    full
```

Recall that for the argon system you had used the *atomic* option. Now, that you are going to simulate a molecular system you will need to store more information than what *atomic* allows. The *full* option enables LAMMPS to store everything that you would need for most molecular simulations (e.g., bonds, angles, charges, etc.), so it is a safe option for many simulations. In case of your polyethylene simulation, you can use the *full* option even though polyethylene will be represented as a set of connected uncharged particles. LAMMPS does not care if you tell it to store and use charges, but then use a force field that does not include charges. Lines 6-9 specify different aspects of how interaction energies, and thus forces, will be calculated in your polyethylene system. Moreover, these lines tell LAMMPS what to expect in terms of force field inputs. Take a moment to review these commands.

You will notice that `in.PE_Simulation` lacks explicit force field input specification commands (e.g., commands like the *pair_coeff* command in the scripts for your argon simulations); however, the input script does contain an *include* command.

```
include    param.pe
```

An *include* command instructs LAMMPS to open the specified file (`param.pe`) and to read in that file as LAMMPS commands. As systems become increasingly large and complex, having all of the force field specification commands in your input script becomes increasingly tedious and further reduces the readability of your LAMMPS scripts. *include* commands enable you to place such commands in a separate file, and have your input scripts focus on the simulation execution details rather than data input.

For example, `param.pe` contains the force field input specification commands for your polyethylene simulation, and some related comments.

Just as LAMMPS offers many options for reading in information, it also has many output options. However, if you were to run `in.PE_Simulation` as is, LAMMPS would only output some thermodynamic information to the `log.lammps` file.

Task #7: Make a copy of `in.PE_Simulation` called `in.task-07`. You will now modify `in.task-07` so that LAMMPS will produce a movie of the polyethylene chain in your system as your polyethylene simulation proceeds. To do so, insert the following two commands into "Data Output Commands" section of `in.task-07`. **Please note**, that this dump style will as such **only** work with the Windows LAMMPS package, as it requires an external program, FFmpeg, which is not included in the virtual machine.

```
group   chain type 1 2
dump    chainMovie chain movie 10 chain.mpg type type box yes 0.01
```

The *group* command creates a selection called *chain* consisting of all particles within the simulation that are either *type 1* or *2*. In your polyethylene simulation and `param.pe`, particle types 1 and 2 correspond to the end and center particles composing the polyethylene chain while type 3 is the solvent. Thus, the above group command selects out the particles composing the polyethylene chain. *group* commands enable you to perform a variety of group specific calculations including: only saving information for a specifc group of particles (as you are doing here), integrating equations of motion for a subset of your system (particularly, useful if you want to treat part of a system as rigid and unmoving), and perform analysis operations for a specific group of particles (see the next Section 4.3).

The *dump movie* command instructs LAMMPS to create an mpeg movie called *chain.mpg* consisting of snapshots of every 10th configuration during the simulation (hence the option *10*), but only for the particles composing the *chain* group. The particles will be sized and colored according to their type (hence the two type options), and the simulation box will be shown (*box yes*).

While jpeg images and movies are good for getting a sense of what is going on in your simulation, saving atomic coordinates enables you to subsequently analyze your trajectory, so it is generally more useful to save text-based instantaneous configurations using a *dump* command rather than just images or a movie. Further modify `in.task-07` so that LAMMPS will save an instantaneous configuration every 2000 timesteps that contains each particle's atomic index, molecular index, particle type, and its position (preferably, its unscaled, unwrapped position). You can do this by completing the following *dump* command, and adding it to the "Data Output Commands" section of `in.task-07`. Look at the LAMMPS documentation to determine what must go at the end of the command where the ellipse is located.

```
dump   dumpConfig all custom 2000 config_*.lammpstrj …
```

When you are ready, run your polyethylene simulation.

Task #7 Follow-up: Take a look at the movie that LAMMPS created. ***Note: You will need to watch it outside your virtual machine in Windows.*** How does the conformation of the polyethylene change with time? How does the movie illustrate periodic boundary conditions. If you are familiar with VMD or

OVITO, you can also install VMD or OVITO and try visualizing one of the lammpstrj files resulting from your simulation.

4.2 Restarts

In the previous section, you started your polyethylene simulation using a configuration that you read in from a LAMMPS *data* file. You can have LAMMPS prepare such *data* files by adding *write_data* commands to your LAMMPS script. For example, …

```
write_data data.PE
```

Here, `data.PE` corresponds to the name of the file that LAMMPS will generate. Note that *write_data* has several options about what force field information will be stored in your *data* file, and your choices here might change how you can subsequently use your *data* file. It is a good idea to include at least one *write_data* command after your simulation's *run* command in order to store your system's final state. You can then use the resulting *data* file to start subsequent simulations.

Occasionally, your simulations will end prematurely, perhaps due to power outages or other problems. In these situations, you would probably like to just restart the simulation (pick up from where it left off), rather then start the simulation all over again. While you could try using *data* files to get around this restart problem, LAMMPS offers a binary restart file format for simulation restarts. You can instruct LAMMPS to write restart files using the *restart* command.

```
restart   ${restartFreq} restart.*
```

Here, *restartFreq* is a variable that indicates how often LAMMPS will save *restart* files while *restart.** is the filename template for your restart files. In *restart.**, the * will be replace by the simulation's timestep number each time LAMMPS writes a restart file. Restart files can be read in using the *read_restart* command. Only certain types of information are saved in *restart* files, and not everything in your initial LAMMPS script will be recovered when you restart your simulation from a *restart* file. Consult the LAMMPS documentation for more information about reading, writing and using both *data* files and *restart* files.

4.3 Computes

One of the powerful features of LAMMPS is its ability to perform on-the-fly analysis. Often, such analysis is done by using *compute* commands, which tell LAMMPS to perform specific computations for groups of particles. In this section, you will focus on modifying your polyethylene simulation so that LAMMPS prints out the mean-squared displacement (MSD) of the center of mass of your polymer chain.

Task #8: To do this, you first need to select the polyethylene chain using the *compute chunk/atom* command. The *compute chunk/atom* command assigns the atoms within a system to different groups according to, for example, *type, position, molecule ID* etc. It is thus possible to select individual molecules within your system, and determine their individual properties as a function of time.

Make a copy of `in.task-07`, and call it `in.task-08`. Add the below compute atom/chunk command to the "Data Output Commands" block. The *molecule* option will tell LAMMPS that you want it to create atom sets based on molecular indices. You have to decide what to replace *XXX* with. Hint: *all* will not work, but you have already defined an appropriate *group*.

```
compute chainChunk XXX chunk/atom molecule
```

Below your *compute chunk/atom* line, add the following *compute msd/chunk* command, replacing *XXX*.

```
compute chainMSD XXX msd/chunk chainChunk
```

compute msd/chunk returns the MSD of each chunk that you pass to it (the chunk's MSD along each axis and its total MSD), and stores this information in an array where the number of rows in the array corresponds to the number of chunks that you passed to the command. Here, you are passing it those chunks defined by the *compute atom/chunk* command labelled *chainChunk*, so there should be one chunk corresponding to your polyethylene chain.

Now, the above two commands will result in LAMMPS calculating the mean squared displacement of your polyethylene chain, but will not cause LAMMPS to print out a file with this information. *compute* commands do not result in read/write operations. You can use a *fix ave/time* command to save the chain's MSD information.

```
fix fixMSDPrint chain ave/time 1000 1 1000 c_chainMSD[4] &
    file chain-COM-MSD.dat mode vector
```

This *compute ave/time* command illustrates several important LAMMPS concepts. For clarity, let's run through all of the options. *fixChainMSD* is simply the ID for this *compute ave/time* command while *chain* specifies that the compute should operate on the particles in the *chain* group. The three numbers *1000 1 1000* indicate in number of timesteps:

1. how frequently LAMMPS should sample the quantities used by the *fix* command (so the MSD in this instance),
2. how many of these samples LAMMPS should average (*1* implies no averaging, so LAMMPS just prints instantaneous values), and
3. how often LAMMPS should write the averaged quantities.

Here, LAMMPS will print instantaneous quantities MSD values every 1000 timesteps.

c_chainMSD[4] is specifying what quantity *fix ave/time* should average and write. *c_* tells LAMMPS that the quantity is the result of a *compute* command, and the string after it is the ID for *compute* command of interest (*chainMSD*). Recall that *compute msd/chunk* returns an array of MSD values. *[4]* instructs *fix ave/time* to use the fourth column of that array, which correspond to the total MSD of the chunks.

Notice how the *fix ave/time* is so long, that it spills onto a second line. *&* tells LAMMPS that the command continues on the second line.

file tells LAMMPS that you want the *fix ave/time* command to save its results to a file, and the string after *file* tells LAMMPS the name of the file it is supposed to create (`chain-COM-MSD.dat`, in this instance).

Finally, *vector mode* tells LAMMPS that the fix ave/time command is operating on arrays, and not isolated scalar quanitites.

Add the *fix ave/time* command after your *compute msd/chunk* command, and run in.task-08.

Task #8 Follow-up: Look at the `chain-COM-MSD.dat` file that your polyethylene simulation produced. If everything worked correctly, the start of your file should look something like the following.

```
# Time-averaged data for fix fixMSDPrint
# TimeStep Number-of-rows
# Row c_chainMSD[4]
0 1
1 0
1000 1
1 3.74613
```

Check that you understand how LAMMPS wrote out the MSD data for your polyethylene chain. For example, on line 4, what does 0 and 1 represent? On line 5, what does 1 and 0 represent?

Optional Task: Modify `in.task-08` so that LAMMPS will print out the radius of gyration of the polyethylene chain every 1000 timesteps. Check the LAMMPS documentation to find out how your can calculate the radius of gyration of a *chunk*.