CHAPTER

EIGHT

HOWTO DISCUSSIONS

These doc pages describe how to perform various tasks with LAMMPS, both for users and developers. The glossary website page also lists MD terminology, with links to corresponding LAMMPS manual pages. The example input scripts included in the examples directory of the LAMMPS source code distribution and highlighted on the *Example scripts* page also show how to set up and run various kinds of simulations.

8.1 General howto

8.1.1 Restart a simulation

There are 3 ways to continue a long LAMMPS simulation. Multiple *run* commands can be used in the same input script. Each run will continue from where the previous run left off. Or binary restart files can be saved to disk using the *restart* command. At a later time, these binary files can be read via a *read_restart* command in a new script. Or they can be converted to text data files using the *-r command-line switch* and read by a *read_data* command in a new script.

Here we give examples of 2 scripts that read either a binary restart file or a converted data file and then issue a new run command to continue where the previous run left off. They illustrate what settings must be made in the new script. Details are discussed in the documentation for the *read_restart* and *read_data* commands.

Look at the *in.chain* input script provided in the *bench* directory of the LAMMPS distribution to see the original script that these 2 scripts are based on. If that script had the line

```
restart         50 tmp.restart
```

added to it, it would produce two binary restart files (tmp.restart.50 and tmp.restart.100) as it ran.

This script could be used to read the first restart file and re-run the last 50 timesteps:

```
read_restart    tmp.restart.50

neighbor        0.4 bin
neigh_modify    every 1 delay 1

fix             1 all nve
fix             2 all langevin 1.0 1.0 10.0 904297

timestep        0.012

run             50
```

Note that the following commands do not need to be repeated because their settings are included in the restart file: units, atom_style, special_bonds, pair_style, bond_style. However, these commands do need to be used, since their settings are not in the restart file: neighbor, fix, timestep.

If you actually use this script to perform a restarted run, you will notice that the thermodynamic data match at step 50 (if you also put a thermo 50 command in the original script), but do not match at step 100. This is because the *fix langevin* command uses random numbers in a way that does not allow for perfect restarts.

As an alternate approach, the restart file could be converted to a data file as follows:

```
lmp_g++ -r tmp.restart.50 tmp.restart.data
```

Then, this script could be used to re-run the last 50 steps:

```
units          lj
atom_style     bond
pair_style     lj/cut 1.12
pair_modify    shift yes
bond_style     fene
special_bonds  0.0 1.0 1.0

read_data      tmp.restart.data

neighbor       0.4 bin
neigh_modify   every 1 delay 1

fix            1 all nve
fix            2 all langevin 1.0 1.0 10.0 904297

timestep       0.012

reset_timestep 50
run            50
```

Note that nearly all the settings specified in the original in.chain script must be repeated, except the pair_coeff and bond_coeff commands, since the new data file lists the force field coefficients. Also, the *reset_timestep* command is used to tell LAMMPS the current timestep. This value is stored in restart files, but not in data files.

8.1.2 Visualize LAMMPS snapshots

Snapshots from LAMMPS simulations can be viewed, visualized, and analyzed in a variety of ways.

LAMMPS snapshots are created by the *dump* command, which can create files in several formats. The native LAMMPS dump format is a text file (see dump atom or dump custom) which can be visualized by several visualization tools for MD simulation trajectories. OVITO and VMD seem to be the most popular choices among them.

The *dump image* and *dump movie* styles can output internally rendered images or convert them to a movie during the MD run. It is also possible to create visualizations from LAMMPS inputs or restart file with the *LAMMPS-GUI*, which uses the *dump image* command internally. The Snapshot Image Viewer can be used to adjust the visualization of the system interactively and then export the corresponding LAMMPS commands to the clipboard to be inserted into input files.

Programs included with LAMMPS as auxiliary tools can convert between LAMMPS format files and other formats. See the *Tools* page for details. These are rarely needed these days.

8.1.3 Run multiple simulations from one input script

This can be done in several ways. See the documentation for individual commands for more details on how these examples work.

If "multiple simulations" means to continue a previous simulation for more timesteps, then you simply use the *run* command multiple times. For example, this script

```
units lj
atom_style atomic
read_data data.lj
run 10000
run 10000
run 10000
run 10000
run 10000
```

would run 5 successive simulations of the same system for a total of 50,000 timesteps.

If you wish to run totally different simulations, one after the other, the *clear* command can be used in between them to re-initialize LAMMPS. For example, this script

```
units lj
atom_style atomic
read_data data.lj
run 10000
clear
units lj
atom_style atomic
read_data data.lj.new
run 10000
```

would run 2 independent simulations, one after the other.

For large numbers of independent simulations, you can use *variables* and the *next* and *jump* commands to loop over the same input script multiple times with different settings. For example, this script, named in.polymer

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
read_data data.polymer
run 10000
shell cd ..
clear
next d
jump in.polymer
```

would run 8 simulations in different directories, using a data.polymer file in each directory. The same concept could be used to run the same system at 8 different temperatures, using a temperature variable and storing the output in different log and dump files, for example

```
variable a loop 8
variable t index 0.8 0.85 0.9 0.95 1.0 1.05 1.1 1.15
log log.$a
read data.polymer
velocity all create $t 352839
```

(continues on next page)

(continued from previous page)

```
fix 1 all nvt $t $t 100.0
dump 1 all atom 1000 dump.$a
run 100000
clear
next t
next a
jump in.polymer
```

All of the above examples work whether you are running on 1 or multiple processors, but assumed you are running LAMMPS on a single partition of processors. LAMMPS can be run on multiple partitions via the *-partition command-line switch*.

In the last 2 examples, if LAMMPS were run on 3 partitions, the same scripts could be used if the index and loop variables were replaced with *universe*-style variables, as described in the *variable* command. Also, the next t and next a commands would need to be replaced with a single next a t command. With these modifications, the 8 simulations of each script would run on the 3 partitions one after the other until all were finished. Initially, 3 simulations would be started simultaneously, one on each partition. When one finished, that partition would then start the fourth simulation, and so forth, until all 8 were completed.

8.1.4 Multi-replica simulations

Several commands in LAMMPS run multi-replica simulations, meaning that multiple instances (replicas) of your simulation are run simultaneously, with small amounts of data exchanged between replicas periodically.

These are the relevant commands:

- *hyper* for bond boost hyperdynamics (HD)
- *neb* for nudged elastic band calculations (NEB)
- *neb_spin* for magnetic nudged elastic band calculations
- *prd* for parallel replica dynamics (PRD)
- *tad* for temperature accelerated dynamics (TAD)
- *temper* for parallel tempering with fixed volume
- *temper/npt* for parallel tempering extended for NPT
- *temper/grem* for parallel tempering with generalized replica exchange (gREM)
- *fix pimd* for path-integral molecular dynamics (PIMD)

NEB is a method for finding transition states and barrier potential energies. HD, PRD, and TAD are methods for performing accelerated dynamics to find and perform infrequent events. Parallel tempering or replica exchange runs different replicas at a series of temperature to facilitate rare-event sampling. PIMD runs different replicas whose individual particles in different replicas are coupled together by springs to model a system of ring-polymers which can represent the quantum nature of atom cores.

These commands can only be used if LAMMPS was built with the REPLICA package. See the *Build package* page for more info.

In all these cases, you must run with one or more processors per replica. The processors assigned to each replica are determined at run-time by using the *-partition command-line switch* to launch LAMMPS on multiple partitions, which in this context are the same as replicas. E.g. these commands:

```
mpirun -np 16 lmp_linux -partition 8x2 -in in.temper
mpirun -np 8 lmp_linux -partition 8x1 -in in.neb
```

would each run 8 replicas, on either 16 or 8 processors. Note the use of the *-in command-line switch* to specify the input script which is required when running in multi-replica mode.

Also note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors. Thus, the above commands could be run on a single-processor (or few-processor) desktop so that you can run a multi-replica simulation on more replicas than you have physical processors. This is useful for testing and debugging, since with most modern processors and MPI libraries, the efficiency of a calculation can severely diminish when oversubscribing processors.

8.1.5 Library interface to LAMMPS

As described on the *Build basics* doc page, LAMMPS can be built as a static or shared library, so that it can be called by another code, used in a *coupled manner* with other codes, or driven through a *Python interface*.

At the core of LAMMPS is the LAMMPS class, which encapsulates the state of the simulation program through the state of the various class instances that it is composed of. So a calculation using LAMMPS requires creating an instance of the LAMMPS class and then send it (text) commands, either individually or from a file, or perform other operations that modify the state stored inside that instance or drive simulations. This is essentially what the src/main.cpp file does as well for the standalone LAMMPS executable, reading commands either from an input file or the standard input.

Creating a LAMMPS instance can be done by using C++ code directly or through a C-style interface library to LAMMPS that is provided in the files src/library.cpp and src/library.h. This *C language API*, can be used from C and C++, and is also the basis for the *Python* and *Fortran* interfaces or the *SWIG based wrappers* included in the LAMMPS source code.

The examples/COUPLE and python/examples directories contain some example programs written in C++, C, Fortran, and Python, which show how a driver code can link to LAMMPS as a library, run LAMMPS on a subset of processors (so the others are available to run some other code concurrently), grab data from LAMMPS, change it, and send it back into LAMMPS.

A detailed documentation of the available APIs and examples of how to use them can be found in the *Programmer Guide* section of this manual.

8.1.6 Coupling LAMMPS to other codes

LAMMPS is designed to support being coupled to other codes. For example, a quantum mechanics code might compute forces on a subset of atoms and pass those forces to LAMMPS. Or a continuum finite element (FE) simulation might use atom positions as boundary conditions on FE nodal points, compute a FE solution, and return interpolated forces on MD atoms.

LAMMPS can be coupled to other codes in at least 4 different ways. Each has advantages and disadvantages, which you will have to think about in the context of your application.

1. Define a new *fix* or *compute* command that calls the other code. In this scenario, LAMMPS is the driver code. During timestepping, the fix or compute is invoked, and can make library calls to the other code, which has been linked to LAMMPS as a library. This is the way the *VORONOI* package, which computes Voronoi tesselations using the Voro++ library, is interfaced to LAMMPS. See the *compute voronoi* command for more details. Also see the *Modify* pages for information on how to add a new fix or compute to LAMMPS.

2. Define a new LAMMPS command that calls the other code. This is conceptually similar to method (1), but in this case LAMMPS and the other code are on a more equal footing. Note that now the other code is not called during the timestepping of a LAMMPS run, but between runs. The LAMMPS input script can be used to alternate LAMMPS runs with calls to the other code, invoked via the new command. The *run* command facilitates this with its *every* option, which makes it easy to run a few steps, invoke the command, run a few steps, invoke the command, etc.

In this scenario, the other code can be called as a library, as in 1., or it could be a stand-alone code, invoked by a system() call made by the command (assuming your parallel machine allows one or more processors to start up another program). In the latter case the stand-alone code could communicate with LAMMPS through files that the command writes and reads.

See the *Modify command* page for information on how to add a new command to LAMMPS.

3. Use LAMMPS as a library called by another code. In this case, the other code is the driver and calls LAMMPS as needed. Alternately, a wrapper code could link and call both LAMMPS and another code as libraries. Again, the *run* command has options that allow it to be invoked with minimal overhead (no setup or clean-up) if you wish to do multiple short runs, driven by another program. Details about using the library interface are given in the *library API* documentation.
4. Couple LAMMPS with another code in a client/server fashion, using the MDI Library developed by the Molecular Sciences Software Institute (MolSSI) to run LAMMPS as either an MDI driver (client) or an MDI engine (server). The MDI driver issues commands to the MDI server to exchange data between them. See the *Using LAMMPS with the MDI library for code coupling* page for more information about how LAMMPS can operate in either of these modes.

8.1.7 Using LAMMPS with the MDI library for code coupling

Client/server coupling of two (or more) codes is where one code is the "client" and sends request messages (data) to one (or more) "server" code(s). A server responds to each request with a reply message (data). This enables two (or more) codes to work in tandem to perform a simulation. In this context, LAMMPS can act as either a client or server code. It does this by using the MolSSI Driver Interface (MDI) library, developed by the Molecular Sciences Software Institute (MolSSI), which is supported by the *MDI* package.

Alternate methods for coupling codes with LAMMPS are described on the *Coupling LAMMPS to other codes* page.

Some advantages of client/server coupling are that the codes can run as stand-alone executables; they need not be linked together. Thus, neither code needs to have a library interface. This also makes it easy to run the two codes on different numbers of processors. If a message protocol (format and content) is defined for a particular kind of simulation, then in principle any code which implements the client-side protocol can be used in tandem with any code which implements the server-side protocol. Neither code needs to know what specific other code it is working with.

In MDI nomenclature, a client code is the "driver", and a server code is an "engine". One driver code can communicate with one or more instances of one or more engine codes. Driver and engine codes can be written in any language: C, C++, Fortran, Python, etc.

In addition to allowing driver and engine(s) to run as stand-alone executables, MDI also enables an engine to be a *plugin* to the client code. In this scenario, server code(s) are compiled as shared libraries, and one (or more) instances of the server are instantiated by the driver code. If the driver code runs in parallel, it can split its MPI communicator into multiple sub-communicators, and launch each plugin engine instance on a sub-communicator. Driver processors within that sub-communicator exchange messages with the corresponding engine instance, and can also send MPI messages to other processors in the driver. The driver code can also destroy engine instances and re-instantiate them. LAMMPS can operate as either a stand-alone or plugin MDI engine. When it operates as a driver, it can use either stand-alone or plugin MDI engines.

The way in which an MDI driver communicates with an MDI engine is by making MDI_Send() and MDI_Recv() calls, which are conceptually similar to MPI_Send() and MPI_Recv() calls. Each send or receive operation uses a string to identify the command name, and optionally some data, which can be a single value or vector of values of any data type. Inside the MDI library, data is exchanged between the driver and engine via MPI calls or sockets. This is a run-time choice by the user.

The *MDI* package provides a *mdi engine* command, which enables LAMMPS to operate as an MDI engine. Its doc page explains the variety of standard and custom MDI commands which the LAMMPS engine recognizes and can

respond to.

The package also provides a *mdi plugin* command, which enables LAMMPS to operate as an MDI driver and load an MDI engine as a plugin library.

The package furthermore includes a *fix mdi/qm* command, in which LAMMPS operates as an MDI driver in conjunction with a quantum mechanics code as an MDI engine. The post_force() method of the fix_mdi_qm.cpp file shows how a driver issues MDI commands to another code. This command can be used to couple to an MDI engine, which is either a stand-alone code or a plugin library.

As explained in the *fix mdi/qm* command documentation, it can be used to perform *ab initio* MD simulations or energy minimizations, or to evaluate the quantum energy and forces for a series of independent systems. The examples/mdi directory has example input scripts for all of these use cases.

The package also has a *fix mdi/qmmm* command in which LAMMPS operates as an MDI driver in conjunction with a quantum mechanics code as an MDI engine to perform QM/MM simulations. The LAMMPS input script partitions the system into QM and MM (molecular mechanics) atoms. As described below the examples/QUANTUM directory has examples for coupling to 3 different quantum codes in this manner.

The examples/mdi directory contains Python scripts and LAMMPS input script which use LAMMPS as either an MDI driver or engine, or both. Currently, 5 example use cases are provided:

- Run ab initio MD (AIMD) using 2 instances of LAMMPS. As a driver, LAMMPS performs the timestepping in either NVE or NPT mode. As an engine, LAMMPS computes forces and is a surrogate for a quantum code.
- LAMMPS runs an MD simulation as a driver. Every N steps it passes the current snapshot to an MDI engine to evaluate the energy, virial, and peratom forces. As the engine, LAMMPS is a surrogate for a quantum code.
- LAMMPS loops over a series of data files and passes the configuration to an MDI engine to evaluate the energy, virial, and peratom forces and thus acts as a simulation driver. As the engine, LAMMPS is used as a surrogate for a quantum code.
- A Python script driver invokes a sequence of unrelated LAMMPS calculations. Calculations can be single-point energy/force evaluations, MD runs, or energy minimizations.
- Run AIMD with a Python driver code and 2 LAMMPS instances as engines. The first LAMMPS instance performs MD timestepping. The second LAMMPS instance acts as a surrogate QM code to compute forces.

Note

In any of these examples where LAMMPS is used as an engine, an actual QM code (provided it has support for MDI) could be used in its place, without modifying the input scripts or launch commands, except to specify the name of the QM code.

The examples/mdi/Run.sh file illustrates how to launch both driver and engine codes so that they communicate using the MDI library via either MPI or sockets, or using the engine as a stand-alone code, or as a plugin library.

As of March 2023, these are quantum codes with MDI support provided via Python wrapper scripts included in the LAMMPS distribution. These can be used with the fix mdi/qm and fix mdi/qmmm commands to perform QM calculations of an entire system (e.g. AIMD) or QM/MM simulations. See the examples/QUANTUM sub-directories for more details:

- LATTE - AIMD only
- PySCF - QM/MM only
- NWChem - AIMD or QM/MM

There are also at least two quantum codes which have direct MDI support, Quantum ESPRESSO (QE) and INQ. There are also several QM codes which have indirect support through QCEngine or i-PI. The former means they require a wrapper program (QCEngine) with MDI support which writes/read files to pass data to the quantum code itself. The list of QCEngine-supported and i-PI-supported quantum codes is on the MDI webpage.

These direct- and indirect-support codes should be usable for full system calculations (e.g. AIMD). Whether they support QM/MM models depends on the individual QM code.

8.1.8 Broken Bonds

Typically, molecular bond interactions persist for the duration of a simulation in LAMMPS. However, some commands break bonds dynamically, including the following:

- *bond_style quartic*
- *fix bond/break*
- *fix bond/react*
- *BPM package* bond styles

A bond can break if it is stretched beyond a user-defined threshold or more generally if other criteria are met.

For the quartic bond style, when a bond is broken its bond type is set to 0 to effectively break it and pairwise forces between the two atoms in the broken bond are "turned on". Angles, dihedrals, etc cannot be defined for a system when *bond_style quartic* is used.

Similarly, bond styles in the BPM package are also incompatible with angles, dihedrals, etc. and when a bond breaks its type is set to zero. However, in the BPM package one can either turn off all pair interactions between bonded particles or leave them on, overlaying pair forces on top of bond forces. To remove pair forces, the special bond list is dynamically updated. More details can be found on the *Howto BPM* page.

The *fix bond/break* and *fix bond/react* commands allow breaking of bonds within a molecular topology with may also define angles, dihedrals, etc. These commands update internal topology data structures to remove broken bonds, as well as the appropriate angle, dihedral, etc interactions which include the bond. They also trigger a rebuild of the neighbor list when this occurs, to turn on the appropriate pairwise forces.

Note that when bonds are dumped to a file via the *dump local* command, bonds with type 0 are not included.

The *delete_bonds* command can be used to query the status of broken bonds with type = 0 or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

The compute *count/type* command tallies the current number of bonds (or angles, etc) for each bond (angle, etc) type. It also tallies broken bonds with type = 0.

The compute *nbond/atom* command tallies the current number of bonds each atom is part of, excluding broken bonds with type = 0.

8.2 Settings howto

8.2.1 2d simulations

You must use the *dimension* command to specify a 2d simulation. The default is 3d.

A 2d simulation box must be periodic in z as set by the *boundary* command. This is the default.

Simulation boxes in LAMMPS can be either orthogonal or triclinic in shape. Orthogonal boxes in 2d are a rectangle with 4 edges that are each perpendicular to either the x or y coordinate axes. Triclinic boxes in 2d are a parallelogram with opposite pairs of faces parallel to each other. LAMMPS supports two forms of triclinic boxes, restricted and general, which for 2d differ in how the box is oriented with respect to the xy coordinate axes. See the *Howto triclinic* for a detailed description of all 3 kinds of simulation boxes.

Here are examples of using the *create_box* command to define the simulation box for a 2d system.

```
# 2d orthogonal box using a block-style region
region mybox block -10 10 0 10 -0.5 0.5
create_box 1 mybox

# 2d restricted triclinic box using a prism-style region with only xy tilt
region mybox prism 0 10 0 10 -0.5 0.5 2.0 0.0 0.0
create_box 1 mybox

# 2d general triclinic box using a primitive cell for a 2d hex lattice
lattice       custom 1.0 a1 1.0 0.0 0.0 a2 0.5 0.86602540378 0.0 &
              a3 0.0 0.0 1.0 basis 0.0 0.0 0.0 triclinic/general
create_box    1 NULL 0 5 0 5 -0.5 0.5
```

Note that for 2d orthogonal or restricted triclinic boxes, the box has a 3rd dimension which must straddle z = 0.0 in the z dimension. Typically the width of box in the z dimension should be narrow, e.g. -0.5 to 0.5, but that is not required. For a 2d general triclinic box, the *a3* vector defined by the *lattice* command must be (0.0,0.0,1.0), which is its default value. Also the *clo* and *chi* arguments of the *create_box* command must be -0.5 and 0.5.

Here are examples of using the *read_data* command to define the simulation box for a 2d system via keywords in the header section of the data file. These are the same boxes as the examples for the *create_box* command

```
# 2d orthogonal box
-10 10   xlo xhi
0 10     ylo yhi
-0.5 0.5 zlo zhi       # this is the default, so no need to specify

# 2d restricted triclinic box with only xy tilt
-10 10   xlo xhi
0 10     ylo yhi
-0.5 0.5 zlo zhi       # this is the default, so no need to specify
2.0 0.0 0.0 xy xz yz

# 3d general triclinic box using a primitive cell for a 2d hex lattice
5 0 0              avec
2.5 4.3301270189 0 bvec
0 0 1              cvec          # this is the default, so no need to specify
0 0 -0.5           abc origin    # this is the default for 2d, so no need to specify
```

Note that for 2d orthogonal or restricted triclinic boxes, the box has a 3rd dimension specified by the *zlo zhi* values,

which must straddle z = 0.0. Typically the width of box in the z dimension should be narrow, e.g. -0.5 to 0.5, but that is not required. For a 2d general triclinic box, the z component of *avec* and *bvec* must be zero, and *cvec* must be (0,0,1), which is the default. The z component of *abc origin* must also be -0.5, which is the default.

If using the *create_atoms* command to create atoms in the 2d simulation box, all the z coordinates of created atoms will be zero.

If using the *read_data* command to read in a data file of atom coordinates for a 2d system, the z coordinates of all atoms should be zero. A value within epsilon of zero is also allowed in case the data file was generated by another program with finite numeric precision, in which case the z coord for the atom will be set to zero.

Use the *fix enforce2d* command as the last fix defined in the input script. It ensures that the z-components of velocities and forces are zeroed out every timestep. The reason to make it the last fix is so that any forces added by other fixes will also be zeroed out.

Many of the example input scripts included in the examples directory are for 2d models.

Note

Some models in LAMMPS treat particles as finite-size spheres, as opposed to point particles. See the *atom_style sphere* and *fix nve/sphere* commands for details. By default, for 2d simulations, such particles will still be modeled as 3d spheres, not 2d discs (circles), meaning their moment of inertia will be that of a sphere. If you wish to model them as 2d discs, see the *set density/disc* command and the *disc* option for the *fix nve/sphere*, *fix nvt/sphere*, *fix nph/sphere*, *fix npt/sphere* commands.

8.2.2 Type labels

Added in version 15Sep2022.

Each atom in LAMMPS has an associated numeric atom type. Similarly, each bond, angle, dihedral, and improper is assigned a bond type, angle type, and so on. The primary use of these types is to map potential (force field) parameters to the interactions of the atom, bond, angle, dihedral, and improper.

By default, type values are entered as integers from 1 to Ntypes wherever they appear in LAMMPS input or output files. The total number Ntypes for each interaction is "locked in" when the simulation box is created.

A recent addition to LAMMPS is the option to use strings - referred to as type labels - as an alternative. Using type labels instead of numeric types can be advantageous in various scenarios. For example, type labels can make inputs more readable and generic (i.e. usable through the *include command* for different systems with different numerical values assigned to types. This generality also applies to other inputs like data files read by *read_data* or molecule template files read by the *molecule* command. A discussion of the current type label support can be found in *(Gissinger)*. See below for a list of other commands that can use type labels in different ways.

LAMMPS will *internally* continue to use numeric types, which means that many previous restrictions still apply. For example, the total number of types is locked in when creating the simulation box, and potential parameters for each type must be provided even if not used by any interactions.

A collection of type labels for all type-kinds (atom types, bond types, etc.) is stored as a "label map" which is simply a list of numeric types and their associated type labels. Within a type-kind, each type label must be unique. It can be assigned to only one numeric type. To read and write type labels to data files for a given type-kind, *all* associated numeric types need have a type label assigned. Partial maps can be saved with the *labelmap write* command and read back with the *include* command.

Valid type labels can contain most ASCII characters, but cannot start with a number, a '#', or a '*'. Also, labels must not contain whitespace characters. When using the *labelmap command* in the LAMMPS input, if certain characters appear in the type label, such as the single (') or double (") quote or the '#' character, the label must be put in either

double, single, or triple (“””) quotes. Triple quotes allow for the most generic type label strings, but they require to have a leading and trailing blank space. When defining type labels the blanks will be ignored. Example:

```
labelmap angle 1 """ C1'-C2"-C3# """
```

This command will map the string `C1'-C2"-C3#` to the angle type 1.

There are two ways to define label maps. One is via the *labelmap* command. The other is via the *read_data* command. A data file can have sections such as *Atom Type Labels*, *Bond Type Labels*, etc., which assign type labels to numeric types. The label map can be written out to data files by the *write_data* command. This map is also written to and read from restart files, by the *write_restart* and *read_restart* commands.

Use of type labels in LAMMPS input or output

Many LAMMPS input script commands that take a numeric type as an argument can use the associated type label instead. If a type label is not defined for a particular numeric type, only its numeric type can be used.

This example assigns labels to the atom types, and then uses the type labels to redefine the pair coefficients.

```
pair_coeff 1 2 1.0 1.0          # numeric types
labelmap atom 1 C 2 H
pair_coeff C H 1.0 1.0          # type labels
```

Adding support for type labels to various commands is an ongoing project. If an input script command (or a section in a file read by a command) allows substituting a type label for a numeric type argument, it will be explicitly mentioned in that command’s documentation page.

As a temporary measure, input script commands can take advantage of variables and how they can be expanded during processing of the input. The variables can use functions that will translate type label strings to their respective number as defined in the current label map. See the *variable* command for details.

For example, here is how the pair_coeff command could be used with type labels if it did not yet support them, either with an explicit variable command or an implicit variable used in the pair_coeff command.

```
labelmap atom 1 C 2 H
variable atom1 equal label2type(atom,C)
variable atom2 equal label2type(atom,H)
pair_coeff ${atom1} ${atom2} 1.0 1.0
```

```
labelmap atom 1 C 2 H
pair_coeff $(label2type(atom,C)) $(label2type(atom,H)) 80.0 1.2
```

Commands that can use label types

Any workflow that involves reading multiple data files, molecule templates or a combination of the two can be streamlined by using type labels instead of numeric types, because types are automatically synced between the files. The creation of simulation-ready reaction templates for *fix bond/react* is much simpler when using type labels, and results in templates that can be used without modification in multiple simulations or different systems.

(Gissinger) J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

8.2.3 Triclinic (non-orthogonal) simulation boxes

By default, LAMMPS uses an orthogonal simulation box to encompass the particles. The orthogonal box has its "origin" at (xlo,ylo,zlo) and extends to (xhi,yhi,zhi). Conceptually it is defined by 3 edge vectors starting from the origin given by **A** = (xhi-xlo,0,0); **B** = (0,yhi-ylo,0); **C** = (0,0,zhi-zlo). The *boundary* command sets the boundary conditions for the 6 faces of the box (periodic, non-periodic, etc). The 6 parameters (xlo,xhi,ylo,yhi,zlo,zhi) are defined at the time the simulation box is created by one of these commands:

- *create_box*
- *read_data*
- *read_restart*
- *read_dump*

Internally, LAMMPS defines box size parameters lx,ly,lz where lx = xhi-xlo, and similarly in the y and z dimensions. The 6 parameters, as well as lx,ly,lz, can be output via the *thermo_style custom* command. See the *Howto 2d* doc page for info on how zlo and zhi are defined for 2d simulations.

Triclinic simulation boxes

LAMMPS also allows simulations to be performed using triclinic (non-orthogonal) simulation boxes shaped as a 3d parallelepiped with triclinic symmetry. For 2d simulations a triclinic simulation box is effectively a parallelogram; see the *Howto 2d* doc page for details.

One use of triclinic simulation boxes is to model solid-state crystals with triclinic symmetry. The *lattice* command can be used with non-orthogonal basis vectors to define a lattice that will tile a triclinic simulation box via the *create_atoms* command.

A second use is to run Parrinello-Rahman dynamics via the *fix npt* command, which will adjust the xy, xz, yz tilt factors to compensate for off-diagonal components of the pressure tensor. The analog for an *energy minimization* is the *fix box/relax* command.

A third use is to shear a bulk solid to study the response of the material. The *fix deform* command can be used for this purpose. It allows dynamic control of the xy, xz, yz tilt factors as a simulation runs. This is discussed in the *Howto NEMD* doc page on non-equilibrium MD (NEMD) simulations.

Conceptually, a triclinic parallelepiped is defined with an "origin" at (xlo,ylo,zhi) and 3 edge vectors **A** = (ax,ay,az), **B** = (bx,by,bz), **C** = (cx,cy,cz) which can be arbitrary vectors, so long as they are non-zero, distinct, and not co-planar. In addition, they must define a right-handed system, such that (**A** cross **B**) points in the direction of **C**. Note that a left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the **A**, **B**, **C** vectors.

The 4 commands listed above for defining orthogonal simulation boxes have triclinic options which allow for specification of the origin and edge vectors **A**, **B**, **C**. For each command, this can be done in one of two ways, for what LAMMPS calls a *general* triclinic box or a *restricted* triclinic box.

A *general* triclinic box is specified by an origin (xlo, ylo, zlo) and arbitrary edge vectors **A** = (ax,ay,az), **B** = (bx,by,bz), and **C** = (cx,cy,cz). So there are 12 parameters in total.

A *restricted* triclinic box also has an origin (xlo,ylo,zlo), but its edge vectors are of the following restricted form: **A** = (xhi-xlo,0,0), **B** = (xy,yhi-ylo,0), **C** = (xz,yz,zhi-zlo). So there are 9 parameters in total. Note that the restricted form requires **A** to be along the x-axis, **B** to be in the xy plane with a y-component in the +y direction, and **C** to have its z-component in the +z direction. Note that a restricted triclinic box is *right-handed* by construction since (**A** cross **B**) points in the direction of **C**.

The *xy,xz,yz* values can be zero or positive or negative. They are called "tilt factors" because they are the amount of displacement applied to edges of faces of an orthogonal box to change it into a restricted triclinic parallelepiped.

Note

Any right-handed general triclinic box (i.e. solid-state crystal basis vectors) can be rotated in 3d around its origin in order to conform to the LAMMPS definition of a restricted triclinic box. See the discussion in the next sub-section about general triclinic simulation boxes in LAMMPS.

Note that the *thermo_style custom* command has keywords for outputting the various parameters that define the size and shape of orthogonal, restricted triclinic, and general triclinic simulation boxes.

For orthogonal boxes there 6 thermo keywords (xlo,ylo,zlo) and (xhi,yhi,zhi).

For restricted triclinic boxes there are 9 thermo keywords for (xlo,ylo,zlo), (xhi,yhi,zhi), and the (xy,xz,yz) tilt factors.

For general triclinic boxes there are 12 thermo keywords for (xlo,ylo,zhi) and the components of the **A**, **B**, **C** edge vectors, namely (avecx,avecy,avecz), (bvecx,bvecy,bvecz), and (cvecx,cvecy,cvecz),

The remainder of this doc page explains (a) how LAMMPS operates with general triclinic simulation boxes, (b) mathematical transformations between general and restricted triclinic boxes which may be useful when creating LAMMPS inputs or interpreting outputs for triclinic simulations, and (c) how LAMMPS uses tilt factors for restricted triclinic simulation boxes.

General triclinic simulation boxes in LAMMPS

LAMMPS allows specification of general triclinic simulation boxes with their atoms as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. Likewise it allows output of dump files, data files, and thermodynamic data (e.g. pressure tensor) in a general triclinic format.

However internally, LAMMPS only uses restricted triclinic simulation boxes. This is for parallel efficiency and to formulate partitioning of the simulation box across processors, neighbor list building, and inter-processor communication of per-atom data with methods similar to those used for orthogonal boxes.

This means 4 things which are important to understand:

- Input of a general triclinic system is immediately converted to a restricted triclinic system.
- If output of per-atom data for a general triclinic system is requested (e.g. for atom coordinates in a dump file), conversion from a restricted to general triclinic system is done at the time of output.
- The conversion of the simulation box and per-atom data from general triclinic to restricted triclinic (and vice versa) is a 3d rotation operation around an origin, which is the lower left corner of the simulation box. This means an input data file for a general triclinic system should specify all per-atom quantities consistent with the general triclinic box and its orientation relative to the standard x,y,z coordinate axes. For example, atom coordinates should be inside the general triclinic simulation box defined by the edge vectors **A**, **B**, **C** and its origin. Likewise per-atom velocities should be in directions consistent with the general triclinic box orientation. E.g. a velocity vector which will be in the +x direction once LAMMPS converts from a general to restricted triclinic box, should be specified in the data file in the direction of the **A** edge vector. See the *read_data* doc page for info on all the per-atom vector quantities to which this rule applies when a data file for a general triclinic box is input.
- If commands such as *write_data* or *dump custom* are used to output general triclinic information, it is effectively the inverse of the operation described in the preceding bullet.

- Other LAMMPS commands such as *region* or *velocity* or *set*, operate on a restricted triclinic system even if a general triclinic system was defined initially.

This is the list of commands which have general triclinic options:

- *create_box* - define a general triclinic box
- *create_atoms* - add atoms to a general triclinic box
- *lattice* - define a custom lattice consistent with the **A**, **B**, **C** edge vectors of a general triclinic box
- *read_data* - read a data file for a general triclinic system
- *write_data* - write a data file for a general triclinic system
- *dump atom, dump custom* - output dump snapshots in general triclinic format
- *dump_modify triclinic/general* - select general triclinic format for dump output
- *thermo_style* - output the pressure tensor in general triclinic format
- *thermo_modify triclinic/general* - select general triclinic format for thermo output
- *read_restart* - read a restart file for a general triclinic system
- *write_restart* - write a restart file for a general triclinic system

---

Transformation from general to restricted triclinic boxes

Let **A**,**B**,**C** be the right-handed edge vectors of a general triclinic simulation box. The equivalent LAMMPS **a**,**b**,**c** for a restricted triclinic box are a 3d rotation of **A**, **B**, and **C** and can be computed as follows:

$$\begin{pmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{pmatrix} = \begin{pmatrix} a_x & b_x & c_x \\ 0 & b_y & c_y \\ 0 & 0 & c_z \end{pmatrix}$$

$$\begin{aligned}
a_x =& A \\
b_x =& \mathbf{B} \cdot \hat{\mathbf{A}} &=& B\cos\gamma \\
b_y =& |\hat{\mathbf{A}} \times \mathbf{B}| &=& B\sin\gamma &=& \sqrt{B^2 - {b_x}^2} \\
c_x =& \mathbf{C} \cdot \hat{\mathbf{A}} &=& C\cos\beta \\
c_y =& \mathbf{C} \cdot \widehat{(\mathbf{A} \times \mathbf{B})} \times \hat{\mathbf{A}} &=& \frac{\mathbf{B}\cdot\mathbf{C} - b_x c_x}{b_y} \\
c_z =& |\mathbf{C} \cdot \widehat{(\mathbf{A} \times \mathbf{B})}| &=& \sqrt{C^2 - {c_x}^2 - {c_y}^2}
\end{aligned}$$

where A = | **A** | indicates the scalar length of **A**. The hat symbol (^) indicates the corresponding unit vector. β and γ are angles between the **A**, **B**, **C** vectors as described below.

For consistency, the same rotation applied to the triclinic box edge vectors can also be applied to atom positions, velocities, and other vector quantities. This can be conveniently achieved by first converting to fractional coordinates in the general triclinic coordinates and then converting to coordinates in the restricted triclinic basis. The transformation is given by the following equation:

$$\mathbf{x} = \begin{pmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{pmatrix} \cdot \frac{1}{V} \begin{pmatrix} \mathbf{B}\times\mathbf{C} \\ \mathbf{C}\times\mathbf{A} \\ \mathbf{A}\times\mathbf{B} \end{pmatrix} \cdot \mathbf{X}$$

where V is the volume of the box (same in either basis), **X** is the fractional vector in the general triclinic basis and **x** is the resulting vector in the restricted triclinic basis.

Crystallographic general triclinic representation of a simulation box

General triclinic crystal structures are often defined using three lattice constants a, b, and c, and three angles α, β, and γ. Note that in this nomenclature, the a, b, and c lattice constants are the scalar lengths of the edge vectors **a**, **b**, and **c** defined above. The relationship between these 6 quantities (a, b, c, α, β, γ) and the LAMMPS restricted triclinic box sizes (lx,ly,lz) = (xhi-xlo,yhi-ylo,zhi-zlo) and tilt factors (xy,xz,yz) is as follows:

$$
\begin{aligned}
a &= \mathrm{lx} \\
b^2 &= \mathrm{ly}^2 + \mathrm{xy}^2 \\
c^2 &= \mathrm{lz}^2 + \mathrm{xz}^2 + \mathrm{yz}^2 \\
\cos\alpha &= \frac{\mathrm{xy} * \mathrm{xz} + \mathrm{ly} * \mathrm{yz}}{b * c} \\
\cos\beta &= \frac{\mathrm{xz}}{c} \\
\cos\gamma &= \frac{\mathrm{xy}}{b}
\end{aligned}
$$

The inverse relationship can be written as follows:

$$
\begin{aligned}
\mathrm{lx} &= a \\
\mathrm{xy} &= b\cos\gamma \\
\mathrm{xz} &= c\cos\beta \\
\mathrm{ly}^2 &= b^2 - \mathrm{xy}^2 \\
\mathrm{yz} &= \frac{b * c\cos\alpha - \mathrm{xy} * \mathrm{xz}}{\mathrm{ly}} \\
\mathrm{lz}^2 &= c^2 - \mathrm{xz}^2 - \mathrm{yz}^2
\end{aligned}
$$

The values of a, b, c, α , β, and γ can be printed out or accessed by computes using the *thermo_style custom* keywords *cella*, *cellb*, *cellc*, *cellalpha*, *cellbeta*, *cellgamma*, respectively.

Output of restricted and general triclinic boxes in a dump file

As discussed on the *dump* command doc page, when the BOX BOUNDS for a snapshot is written to a dump file for a restricted triclinic box, an orthogonal bounding box which encloses the triclinic simulation box is output, along with the 3 tilt factors (xy, xz, yz) of the restricted triclinic box, formatted as follows:

```
ITEM: BOX BOUNDS xy xz yz
xlo_bound xhi_bound xy
ylo_bound yhi_bound xz
zlo_bound zhi_bound yz
```

This bounding box is convenient for many visualization programs and is calculated from the 9 restricted triclinic box parameters (xlo,xhi,ylo,yhi,zlo,zhi,xy,xz,yz) as follows:

$$
\begin{aligned}
\text{xlo\_bound} &= \text{xlo} + \text{MIN}(0.0,\text{xy},\text{xz},\text{xy}+\text{xz}) \\
\text{xhi\_bound} &= \text{xhi} + \text{MAX}(0.0,\text{xy},\text{xz},\text{xy}+\text{xz}) \\
\text{ylo\_bound} &= \text{ylo} + \text{MIN}(0.0,\text{yz}) \\
\text{yhi\_bound} &= \text{yhi} + \text{MAX}(0.0,\text{yz}) \\
\text{zlo\_bound} &= \text{zlo} \\
\text{zhi\_bound} &= \text{zhi}
\end{aligned}
$$

These formulas can be inverted if you need to convert the bounding box back into the restricted triclinic box parameters, e.g. xlo = xlo_bound - MIN(0.0,xy,xz,xy+xz).

Periodicity and tilt factors for triclinic simulation boxes

There is no requirement that a triclinic box be periodic in any dimension, though it typically should be in y or z if you wish to enforce a shift in coordinates due to periodic boundary conditions across the y or z boundaries. See the doc page for the *boundary* command for an explanation of shifted coordinates for restricted triclinic boxes which are periodic.

Some commands that work with triclinic boxes, e.g. the *fix deform* and *fix npt* commands, require periodicity or non-shrink-wrap boundary conditions in specific dimensions. See the command doc pages for details.

A restricted triclinic box can be defined with all 3 tilt factors = 0.0, so that it is initially orthogonal. This is necessary if the box will become non-orthogonal, e.g. due to use of the *fix npt* or *fix deform* commands. Alternatively, you can use the *change_box* command to convert a simulation box from orthogonal to restricted triclinic and vice versa.

Note

Highly tilted restricted triclinic simulation boxes can be computationally inefficient. This is due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

LAMMPS will issue a warning if you define a restricted triclinic box with a tilt factor which skews the box more than half the distance of the parallel box length, which is the first dimension in the tilt factor (e.g. x for xz).

For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor should be between -5 and 5 to avoid the warning. Similarly, both xz and yz should be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that these are not limitations, since if the maximum tilt factor is 5 (as in this example), then simulations boxes and atom configurations with tilt = . . . , -15, -5, 5, 15, 25, . . . are all geometrically equivalent.

If the box tilt exceeds this limit during a dynamics run (e.g. due to the *fix deform* command), then by default the box is "flipped" to an equivalent shape with a tilt factor within the warning bounds, and the run continues. See the *fix deform* page for further details. Box flips that would normally occur using the *fix deform* or *fix npt* commands can be suppressed using the *flip no* option with either of the commands.

One exception to box flipping is if the first dimension in the tilt factor (e.g. x for xy) is non-periodic. In that case, the limits on the tilt factor are not enforced, since flipping the box in that dimension would not change the atom positions due to non-periodicity. In this mode, if the system tilts to large angles, the simulation will simply become inefficient, due to the highly skewed simulation box.

8.2.4 Thermostats

Thermostatting means controlling the temperature of particles in an MD simulation. *Barostatting* means controlling the pressure. Since the pressure includes a kinetic component due to particle velocities, both these operations require calculation of the temperature. Typically a target temperature (T) and/or pressure (P) is specified by the user, and the thermostat or barostat attempts to equilibrate the system to the requested T and/or P.

Thermostatting in LAMMPS is performed by *fixes*, or in one case by a pair style. Several thermostatting fixes are available: Nose-Hoover (nvt), Berendsen, CSVR, Langevin, and direct rescaling (temp/rescale). Dissipative particle dynamics (DPD) thermostatting can be invoked via the *dpd/tstat* pair style:

- *fix nvt*
- *fix nvt/sphere*
- *fix nvt/asphere*
- *fix nvt/sllod*
- *fix temp/berendsen*
- *fix temp/csvr*
- *fix langevin*
- *fix temp/rescale*
- *pair_style dpd/tstat*

Fix nvt only thermostats the translational velocity of particles. *Fix nvt/sllod* also does this, except that it subtracts out a velocity bias due to a deforming box and integrates the SLLOD equations of motion. See the *Howto nemd* page for further details. *Fix nvt/sphere* and *fix nvt/asphere* thermostat not only translation velocities but also rotational velocities for spherical and aspherical particles.

Note

A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

DPD thermostatting alters pairwise interactions in a manner analogous to the per-particle thermostatting of *fix langevin*.

Any of the thermostatting fixes can be instructed to use custom temperature computes that remove bias which has two effects: first, the current calculated temperature, which is compared to the requested target temperature, is calculated with the velocity bias removed; second, the thermostat adjusts only the thermal temperature component of the particle's velocities, which are the velocities with the bias removed. The removed bias is then added back to the adjusted velocities. See the doc pages for the individual fixes and for the *fix_modify* command for instructions on how to assign a temperature compute to a thermostatting fix.

For example, you can apply a thermostat only to atoms in a spatial region by using it in conjunction with *compute temp/region*. Or you can apply a thermostat to only the x and z components of velocity by using it with *compute temp/partial*. Of you could thermostat only the thermal temperature of a streaming flow of particles without affecting the streaming velocity, by using *compute temp/profile*.

Below is a list of custom temperature computes that can be used like that:

- *compute temp/asphere command*
- *compute temp/body command*
- *compute temp/chunk command*

- *compute temp/com command*
- *compute temp/deform command*
- *compute temp/partial command*
- *compute temp/profile command*
- *compute temp/ramp command*
- *compute temp/region command*
- *compute temp/rotate command*
- *compute temp/sphere command*

Note

Only the nvt fixes perform time integration, meaning they update the velocities and positions of particles due to forces and velocities respectively. The other thermostat fixes only adjust velocities; they do NOT perform time integration updates. Thus they should be used in conjunction with a constant NVE integration fix such as these:

- *fix nve*
- *fix nve/sphere*
- *fix nve/asphere*

Thermodynamic output, which can be setup via the *thermo_style* command, often includes temperature values. As explained on the page for the *thermo_style* command, the default temperature is setup by the thermo command itself. It is NOT the temperature associated with any thermostatting fix you have defined or with any compute you have defined that calculates a temperature. The doc pages for the thermostatting fixes explain the ID of the temperature compute they create. Thus if you want to view these temperatures, you need to specify them explicitly via the *thermo_style custom* command. Or you can use the *thermo_modify* command to re-define what temperature compute is used for default thermodynamic output.

(Daivis and Todd) Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

8.2.5 Barostats

Barostatting means controlling the pressure in an MD simulation. *Thermostatting* means controlling the temperature of the particles. Since the pressure includes a kinetic component due to particle velocities, both these operations require calculation of the temperature. Typically a target temperature (T) and/or pressure (P) is specified by the user, and the thermostat or barostat attempts to equilibrate the system to the requested T and/or P.

Barostatting in LAMMPS is performed by *fixes*. Two barostatting methods are currently available: Nose-Hoover (npt and nph) and Berendsen:

- *fix npt*
- *fix npt/sphere*
- *fix npt/asphere*
- *fix nph*
- *fix press/berendsen*

The *fix npt* commands include a Nose-Hoover thermostat and barostat. *Fix nph* is just a Nose/Hoover barostat; it does no thermostatting. Both *fix nph* and *fix press/berendsen* can be used in conjunction with any of the thermostatting fixes.

As with the *thermostats*, *fix npt* and *fix nph* only use translational motion of the particles in computing T and P and performing thermo/barostatting. *Fix npt/sphere* and *fix npt/asphere* thermo/barostat using not only translation velocities but also rotational velocities for spherical and aspherical particles.

All of the barostatting fixes use the *compute pressure* compute to calculate a current pressure. By default, this compute is created with a simple *compute temp* (see the last argument of the *compute pressure* command), which is used to calculated the kinetic component of the pressure. The barostatting fixes can also use temperature computes that remove bias for the purpose of computing the kinetic component which contributes to the current pressure. See the doc pages for the individual fixes and for the *fix_modify* command for instructions on how to assign a temperature or pressure compute to a barostatting fix.

Note

As with the thermostats, the Nose/Hoover methods (*fix npt* and *fix nph*) perform time integration. *Fix press/berendsen* does NOT, so it should be used with one of the constant NVE fixes or with one of the NVT fixes.

Thermodynamic output, which can be setup via the *thermo_style* command, often includes pressure values. As explained on the page for the *thermo_style* command, the default pressure is setup by the thermo command itself. It is NOT the pressure associated with any barostatting fix you have defined or with any compute you have defined that calculates a pressure. The doc pages for the barostatting fixes explain the ID of the pressure compute they create. Thus if you want to view these pressures, you need to specify them explicitly via the *thermo_style custom* command. Or you can use the *thermo_modify* command to re-define what pressure compute is used for default thermodynamic output.

8.2.6 Walls

Walls in an MD simulation are typically used to bound particle motion, i.e. to serve as a boundary condition.

Walls in LAMMPS can be of rough (made of particles) or idealized surfaces. Ideal walls can be smooth, generating forces only in the normal direction, or frictional, generating forces also in the tangential direction.

Rough walls, built of particles, can be created in various ways. The particles themselves can be generated like any other particle, via the *lattice* and *create_atoms* commands, or read in via the *read_data* command.

Their motion can be constrained by many different commands, so that they do not move at all, move together as a group at constant velocity or in response to a net force acting on them, move in a prescribed fashion (e.g. rotate around a point), etc. Note that if a time integration fix like *fix nve* or *fix nvt* is not used with the group that contains wall particles, their positions and velocities will not be updated.

- *fix aveforce* - set force on particles to average value, so they move together
- *fix setforce* - set force on particles to a value, e.g. 0.0
- *fix freeze* - freeze particles for use as granular walls
- *fix nve/noforce* - advect particles by their velocity, but without force
- *fix move* - prescribe motion of particles by a linear velocity, oscillation, rotation, variable

The *fix move* command offers the most generality, since the motion of individual particles can be specified with *variable* formula which depends on time and/or the particle position.

For rough walls, it may be useful to turn off pairwise interactions between wall particles via the *neigh_modify exclude* command.

Rough walls can also be created by specifying frozen particles that do not move and do not interact with mobile particles, and then tethering other particles to the fixed particles, via a *bond*. The bonded particles do interact with other mobile particles.

Idealized walls can be specified via several fix commands. *Fix wall/gran* creates frictional walls for use with granular particles; all the other commands create smooth walls.

- *fix wall/reflect* - reflective flat walls
- *fix wall/lj93* - flat walls, with Lennard-Jones 9/3 potential
- *fix wall/lj126* - flat walls, with Lennard-Jones 12/6 potential
- *fix wall/colloid* - flat walls, with *pair_style colloid* potential
- *fix wall/harmonic* - flat walls, with repulsive harmonic spring potential
- *fix wall/morse* - flat walls, with Morse potential
- *fix wall/region* - use region surface as wall
- *fix wall/gran* - flat or curved walls with *pair_style granular* potential

The *lj93*, *lj126*, *colloid*, *harmonic*, and *morse* styles all allow the flat walls to move with a constant velocity, or oscillate in time. The *fix wall/region* command offers the most generality, since the region surface is treated as a wall, and the geometry of the region can be a simple primitive volume (e.g. a sphere, or cube, or plane), or a complex volume made from the union and intersection of primitive volumes. *Regions* can also specify a volume "interior" or "exterior" to the specified primitive shape or *union* or *intersection*. *Regions* can also be "dynamic" meaning they move with constant velocity, oscillate, or rotate.

The only frictional idealized walls currently in LAMMPS are flat or curved surfaces specified by the *fix wall/gran* command. At some point we plan to allow region surfaces to be used as frictional walls, as well as triangulated surfaces.

8.2.7 NEMD simulations

Non-equilibrium molecular dynamics or NEMD simulations are typically used to measure a fluid's rheological properties such as viscosity. In LAMMPS, such simulations can be performed by first setting up a non-orthogonal simulation box (see the preceding Howto section).

A shear strain can be applied to the simulation box at a desired strain rate by using the *fix deform* command. The *fix nvt/sllod* command can be used to thermostat the sheared fluid and integrate the SLLOD equations of motion for the system. Fix nvt/sllod uses *compute temp/deform* to compute a thermal temperature by subtracting out the streaming velocity of the shearing atoms. The velocity profile or other properties of the fluid can be monitored via the *fix ave/chunk* command.

Note

A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

As discussed in the previous section on non-orthogonal simulation boxes, the amount of tilt or skew that can be applied is limited by LAMMPS for computational efficiency to be 1/2 of the parallel box length. However, *fix deform* can continuously strain a box by an arbitrary amount. As discussed in the *fix deform* command, when the tilt value reaches a limit, the box is flipped to the opposite limit which is an equivalent tiling of periodic space. The strain rate can then continue to change as before. In a long NEMD simulation these box re-shaping events may occur many times.

In a NEMD simulation, the "remap" option of *fix deform* should be set to "remap v", since that is what *fix nvt/sllod* assumes to generate a velocity profile consistent with the applied shear strain rate.

An alternative method for calculating viscosities is provided via the *fix viscosity* command.

NEMD simulations can also be used to measure transport properties of a fluid through a pore or channel. Simulations of steady-state flow can be performed using the *fix flow/gauss* command.

(Daivis and Todd) Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

8.2.8 Long-range dispersion settings

The PPPM method computes interactions by splitting the pair potential into two parts, one of which is computed in a normal pairwise fashion, the so-called real-space part, and one of which is computed using the Fourier transform, the so called reciprocal-space or kspace part. For both parts, the potential is not computed exactly but is approximated. Thus, there is an error in both parts of the computation, the real-space and the kspace error. The just mentioned facts are true both for the PPPM for Coulomb as well as dispersion interactions. The deciding difference - and also the reason why the parameters for pppm/disp have to be selected with more care - is the impact of the errors on the results: The kspace error of the PPPM for Coulomb and dispersion interaction and the real-space error of the PPPM for Coulomb interaction have the character of noise. In contrast, the real-space error of the PPPM for dispersion has a clear physical interpretation: the underprediction of cohesion. As a consequence, the real-space error has a much stronger effect than the kspace error on simulation results for pppm/disp. Parameters must thus be chosen in a way that this error is much smaller than the kspace error.

When using pppm/disp and not making any specifications on the PPPM parameters via the kspace modify command, parameters will be tuned such that the real-space error and the kspace error are equal. This will result in simulations that are either inaccurate or slow, both of which is not desirable. For selecting parameters for the pppm/disp that provide fast and accurate simulations, there are two approaches, which both have their up- and downsides.

The first approach is to set desired real-space an kspace accuracies via the *kspace_modify force/disp/real* and *kspace_modify force/disp/kspace* commands. Note that the accuracies have to be specified in force units and are thus dependent on the chosen unit settings. For real units, 0.0001 and 0.002 seem to provide reasonable accurate and efficient computations for the real-space and kspace accuracies. 0.002 and 0.05 work well for most systems using lj units. PPPM parameters will be generated based on the desired accuracies. The upside of this approach is that it usually provides a good set of parameters and will work for both the *kspace_modify diff ad* and *kspace_modify diff ik* options. The downside of the method is that setting the PPPM parameters will take some time during the initialization of the simulation.

The second approach is to set the parameters for the pppm/disp explicitly using the *kspace_modify mesh/disp*, *kspace_modify order/disp*, and *kspace_modify gewald/disp* commands. This approach requires a more experienced user who understands well the impact of the choice of parameters on the simulation accuracy and performance. This approach provides a fast initialization of the simulation. However, it is sensitive to errors: A combination of parameters that will perform well for one system might result in far-from-optimal conditions for other simulations. For example, parameters that provide accurate and fast computations for all-atomistic force fields can provide insufficient accuracy or united-atomistic force fields (which is related to that the latter typically have larger dispersion coefficients).

To avoid inaccurate or inefficient simulations, the pppm/disp stops simulations with an error message if no action is taken to control the PPPM parameters. If the automatic parameter generation is desired and real-space and kspace accuracies are desired to be equal, this error message can be suppressed using the *kspace_modify disp/auto yes* command.

A reasonable approach that combines the upsides of both methods is to make the first run using the *kspace_modify force/disp/real* and *kspace_modify force/disp/kspace* commands, write down the PPPM parameters from the output, and specify these parameters using the second approach in subsequent runs (which have the same composition, force field, and approximately the same volume).

Concerning the performance of the pppm/disp there are two more things to consider. The first is that when using the pppm/disp, the cutoff parameter does no longer affect the accuracy of the simulation (subject to that gewald/disp is adjusted when changing the cutoff). The performance can thus be increased by examining different values for the cutoff parameter. A lower bound for the cutoff is only set by the truncation error of the repulsive term of pair potentials.

The second is that the mixing rule of the pair style has an impact on the computation time when using the pppm/disp. Fastest computations are achieved when using the geometric mixing rule. Using the arithmetic mixing rule substantially increases the computational cost. The computational overhead can be reduced using the *kspace_modify mix/disp geom* and *kspace_modify splittol* commands. The first command simply enforces geometric mixing of the dispersion coefficients in kspace computations. This introduces some error in the computations but will also significantly speed-up the simulations. The second keyword sets the accuracy with which the dispersion coefficients are approximated using a matrix factorization approach. This may result in better accuracy then using the first command, but will usually also not provide an equally good increase of efficiency.

Finally, pppm/disp can also be used when no mixing rules apply. This can be achieved using the *kspace_modify mix/disp none* command. Note that the code does not check automatically whether any mixing rule is fulfilled. If mixing rules do not apply, the user will have to specify this command explicitly.

8.3 Analysis howto

8.3.1 Output from LAMMPS (thermo, dumps, computes, fixes, variables)

There are four basic forms of LAMMPS output:

- *Thermodynamic output*, which is a list of quantities printed every few timesteps to the screen and logfile.
- *Dump files*, which contain snapshots of atoms and various per-atom values and are written at a specified frequency.
- Certain fixes can output user-specified quantities to files: *fix ave/time* for time averaging, *fix ave/chunk* for spatial or other averaging, and *fix print* for single-line output of *variables*. Fix print can also output to the screen.
- *Restart files*.

A simulation prints one set of thermodynamic output and (optionally) restart files. It can generate any number of dump files and fix output files, depending on what *dump* and *fix* commands you specify.

As discussed below, LAMMPS gives you a variety of ways to determine what quantities are calculated and printed when the thermodynamics, dump, or fix commands listed above perform output. Throughout this discussion, note that users can also *add their own computes and fixes to LAMMPS* which can generate values that can then be output with these commands.

The following subsections discuss different LAMMPS commands related to output and the kind of data they operate on and produce:

- *Global/per-atom/local/per-grid data*
- *Scalar/vector/array data*
- *Disambiguation*
- *Thermodynamic output*
- *Dump file output*
- *Fixes that write output files*
- *Computes that process output quantities*
- *Fixes that process output quantities*

- *Computes that generate values to output*
- *Fixes that generate values to output*
- *Variables that generate values to output*
- *Summary table of output options and data flow between commands*

Global/per-atom/local/per-grid data

Various output-related commands work with four different "styles" of data: global, per-atom, local, and per-grid. A global datum is one or more system-wide values, e.g. the temperature of the system. A per-atom datum is one or more values per atom, e.g. the kinetic energy of each atom. Local datums are calculated by each processor based on the atoms it owns, and there may be zero or more per atom, e.g. a list of bond distances.

A per-grid datum is one or more values per grid cell, for a grid which overlays the simulation domain. Similar to atoms and per-atom data, the grid cells and the data they store are distributed across processors; each processor owns the grid cells whose center points fall within its subdomain.

Scalar/vector/array data

Global, per-atom, local, and per-grid datums can come in three "kinds": a single scalar value, a vector of values, or a 2d array of values. More specifically these are the valid kinds for each style:

- global scalar
- global vector
- global array
- per-atom vector
- per-atom array
- local vector
- local array
- per-grid vector
- per-grid array

A per-atom vector means a single value per atom; the "vector" is the length of the number of atoms. A per-atom array means multiple values per atom. Similarly a local vector or array means one or multiple values per entity (e.g. per bond in the system). And a per-grid vector or array means one or multiple values per grid cell.

The doc page for a compute or fix or variable that generates data will specify both the styles and kinds of data it produces, e.g. a per-atom vector. Note that a compute or fix may generate multiple styles and kinds of output. However, for per-atom data only a vector or array is output, never both. Likewise for per-local and per-grid data. An example of a fix which generates multiple styles and kinds of data is the *fix mdi/qm* command. It outputs a global scalar, global vector, and per-atom array for the quantum mechanical energy and virial of the system and forces on each atom.

By contrast, different variable styles generate only a single kind of data: a global scalar for an equal-style variable, global vector for a vector-style variable, and a per-atom vector for an atom-style variable.

When data is accessed by another command, as in many of the output commands discussed below, it can be referenced via the following bracket notation, where ID in this case is the ID of a compute. The leading "c_" would be replaced by "f_" for a fix, or "v_" for a variable (and ID would be the name of the variable):

c_ID	entire scalar, vector, or array
c_ID[I]	one element of vector, one column of array
c_ID[I][J]	one element of array

Note that using one bracket reduces the dimension of the data once (vector -> scalar, array -> vector). Using two brackets reduces the dimension twice (array -> scalar). Thus a command that uses scalar values as input can also conceptually operate on an element of a vector or array.

Per-grid vectors or arrays are accessed similarly, except that the ID for the compute or fix includes a grid name and a data name. This is because a fix or compute can create multiple grids (of different sizes) and multiple sets of data (for each grid). The fix or compute defines names for each grid and for each data set, so that all of them can be accessed by other commands. See the *Howto grid* doc page for more details.

Disambiguation

When a compute or fix produces data in multiple styles, e.g. global and per-atom, a reference to the data can sometimes be ambiguous. Usually the context in which the input script references the data determines which style is meant.

For example, if a compute outputs a global vector and a per-atom array, an element of the global vector will be accessed by using c_ID[I] in *thermodynamic output*, while a column of the per-atom array will be accessed by using c_ID[I] in a *dump custom* command.

However, if a *atom-style variable* references c_ID[I], then it could be intended to refer to a single element of the global vector or a column of the per-atom array. The doc page for any command that has a potential ambiguity (variables are the most common) will explain how to resolve the ambiguity.

In this case, an atom-style variables references per-atom data if it exists. If access to an element of a global vector is needed (as in this example), an equal-style variable which references the value can be defined and used in the atom-style variable formula instead.

Similarly, *thermodynamic output* can only reference global data from a compute or fix. But you can indirectly access per-atom data as follows. The reference c_ID[245][2] for the ID of a *compute displace/atom* command, refers to the y-component of displacement for the atom with ID 245. While you cannot use that reference directly in the *thermo_style* command, you can use it an equal-style variable formula, and then reference the variable in thermodynamic output.

Thermodynamic output

The frequency and format of thermodynamic output is set by the *thermo*, *thermo_style*, and *thermo_modify* commands. The *thermo_style* command also specifies what values are calculated and written out. Pre-defined keywords can be specified (e.g. press, etotal, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute* or *fix* or *variable* provides the value to be output. In each case, the compute, fix, or variable must generate global values for input to the *thermo_style custom* command.

Note that thermodynamic output values can be "extensive" or "intensive". The former scale with the number of atoms in the system (e.g. total energy), the latter do not (e.g. temperature). The setting for *thermo_modify norm* determines whether extensive quantities are normalized or not. Computes and fixes produce either extensive or intensive values; see their individual doc pages for details. *Equal-style variables* produce only intensive values; you can include a division by "natoms" in the formula if desired, to make an extensive calculation produce an intensive result.

Dump file output

Dump file output is specified by the *dump* and *dump_modify* commands. There are several pre-defined formats (dump atom, dump xtc, etc).

There is also a *dump custom* format where the user specifies what values are output with each atom. Pre-defined atom attributes can be specified (id, x, fx, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute* or *fix* or *variable* provides the values to be output. In each case, the compute, fix, or variable must generate per-atom values for input to the *dump custom* command.

There is also a *dump local* format where the user specifies what local values to output. A pre-defined index keyword can be specified to enumerate the local values. Two additional kinds of keywords can also be specified (c_ID, f_ID), where a *compute* or *fix* or *variable* provides the values to be output. In each case, the compute or fix must generate local values for input to the *dump local* command.

There is also a *dump grid* format where the user specifies what per-grid values to output from computes or fixes that generate per-grid data.

Fixes that write output files

Several fixes take various quantities as input and can write output files: *fix ave/time*, *fix ave/chunk*, *fix ave/histo*, *fix ave/correlate*, and *fix print*.

The *fix ave/time* command enables direct output to a file and/or time-averaging of global scalars or vectors. The user specifies one or more quantities as input. These can be global *compute* values, global *fix* values, or *variables* of any style except the atom style which produces per-atom values. Since a variable can refer to keywords used by the *thermo_style custom* command (like temp or press) and individual per-atom values, a wide variety of quantities can be time averaged and/or output in this way. If the inputs are one or more scalar values, then the fix generate a global scalar or vector of output. If the inputs are one or more vector values, then the fix generates a global vector or array of output. The time-averaged output of this fix can also be used as input to other output commands.

The *fix ave/chunk* command enables direct output to a file of chunk-averaged per-atom quantities like those output in dump files. Chunks can represent spatial bins or other collections of atoms, e.g. individual molecules. The per-atom quantities can be atom density (mass or number) or atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The chunk-averaged output of this fix is global and can also be used as input to other output commands.

Note that the *fix ave/grid* command can also average the same per-atom quantities within spatial bins, but it does this for a distributed grid whose grid cells are owned by different processors. It outputs per-grid data, not global data, so it is more efficient for large numbers of averaging bins.

The *fix ave/histo* command enables direct output to a file of histogrammed quantities, which can be global or per-atom or local quantities. The histogram output of this fix can also be used as input to other output commands.

The *fix ave/correlate* command enables direct output to a file of time-correlated quantities, which can be global values. The correlation matrix output of this fix can also be used as input to other output commands.

The *fix print* command can generate a line of output written to the screen and log file or to a separate file, periodically during a running simulation. The line can contain one or more *variable* values for any style variable except the vector or atom styles). As explained above, variables themselves can contain references to global values generated by *thermodynamic keywords*, *computes*, *fixes*, or other *variables*, or to per-atom values for a specific atom. Thus the *fix print* command is a means to output a wide variety of quantities separate from normal thermodynamic or dump file output.

Computes that process output quantities

The *compute reduce* and *compute reduce/region* commands take one or more per-atom or local vector quantities as inputs and "reduce" them (sum, min, max, ave) to scalar quantities. These are produced as output values which can be used as input to other output commands.

The *compute slice* command take one or more global vector or array quantities as inputs and extracts a subset of their values to create a new vector or array. These are produced as output values which can be used as input to other output commands.

The *compute property/atom* command takes a list of one or more pre-defined atom attributes (id, x, fx, etc) and stores the values in a per-atom vector or array. These are produced as output values which can be used as input to other output commands. The list of atom attributes is the same as for the *dump custom* command.

The *compute property/local* command takes a list of one or more pre-defined local attributes (bond info, angle info, etc) and stores the values in a local vector or array. These are produced as output values which can be used as input to other output commands.

The *compute property/grid* command takes a list of one or more pre-defined per-grid attributes (id, grid cell coords, etc) and stores the values in a per-grid vector or array. These are produced as output values which can be used as input to the *dump grid* command.

The *compute property/chunk* command takes a list of one or more pre-defined chunk attributes (id, count, coords for spatial bins) and stores the values in a global vector or array. These are produced as output values which can be used as input to other output commands.

Fixes that process output quantities

The *fix vector* command can create global vectors as output from global scalars as input, accumulating them one element at a time.

The *fix ave/atom* command performs time-averaging of per-atom vectors. The per-atom quantities can be atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The time-averaged per-atom output of this fix can be used as input to other output commands.

The *fix store/state* command can archive one or more per-atom attributes at a particular time, so that the old values can be used in a future calculation or output. The list of atom attributes is the same as for the *dump custom* command, including per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The output of this fix can be used as input to other output commands.

The *fix ave/grid* command performs time-averaging of either per-atom or per-grid data.

For per-atom data it performs averaging for the atoms within each grid cell, similar to the *fix ave/chunk* command when its chunks are defined as regular 2d or 3d bins. The per-atom quantities can be atom density (mass or number) or atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*.

The chief difference between the *fix ave/grid* and *fix ave/chunk* commands when used in this context is that the former uses a distributed grid, while the latter uses a global grid. Distributed means that each processor owns the subset of grid cells within its subdomain. Global means that each processor owns a copy of the entire grid. The *fix ave/grid* command is thus more efficient for large grids.

For per-grid data, the *fix ave/grid* command takes inputs for grid data produced by other computes or fixes and averages the values for each grid point over time.

Computes that generate values to output

Every *compute* in LAMMPS produces either global or per-atom or local or per-grid values. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The page for each compute command describes what it produces. Computes that produce per-atom or local or per-grid values have the word "atom" or "local" or "grid as the last word in their style name. Computes without the word "atom" or "local" or "grid" produce global values.

Fixes that generate values to output

Some *fixes* in LAMMPS produces either global or per-atom or local or per-grid values which can be accessed by other commands. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The page for each fix command tells whether it produces any output quantities and describes them.

Variables that generate values to output

Variables defined in an input script can store one or more strings. But equal-style, vector-style, and atom-style or atomfile-style variables generate a global scalar value, global vector or values, or a per-atom vector, respectively, when accessed. The formulas used to define these variables can contain references to the thermodynamic keywords and to global and per-atom data generated by computes, fixes, and other variables. The values generated by variables can be used as input to and thus output by the other commands described in this section.

Per-grid variables have not (yet) been implemented.

Summary table of output options and data flow between commands

This table summarizes the various commands that can be used for generating output from LAMMPS. Each command produces output data of some kind and/or writes data to a file. Most of the commands can take data from other commands as input. Thus you can link many of these commands together in pipeline form, where data produced by one command is used as input to another command and eventually written to the screen or to a file. Note that to hook two commands together the output and input data types must match, e.g. global/per-atom/local data and scalar/vector/array data.

Also note that, as described above, when a command takes a scalar as input, that could also be an element of a vector or array. Likewise a vector input could be a column of an array.

Command	Input	Output
thermo_style custom	global scalars	screen, log file
dump custom	per-atom vectors	dump file
dump local	local vectors	dump file
dump grid	per-grid vectors	dump file
fix print	global scalar from variable	screen, file
print	global scalar from variable	screen
computes	N/A	global/per-atom/local/per-grid scalar/vector/array
fixes	N/A	global/per-atom/local/per-grid scalar/vector/array
variables	global scalars and vectors, per-atom vectors	global scalar and vector, per-atom vector
compute reduce	per-atom/local vectors	global scalar/vector
compute slice	global vectors/arrays	global vector/array
compute property/atom	N/A	per-atom vector/array
compute property/local	N/A	local vector/array
compute property/grid	N/A	per-grid vector/array
compute property/chunk	N/A	global vector/array
fix vector	global scalars	global vector
fix ave/atom	per-atom vectors	per-atom vector/array
fix ave/time	global scalars/vectors	global scalar/vector/array, file
fix ave/chunk	per-atom vectors	global array, file
fix ave/grid	per-atom vectors or per-grid vectors	per-grid vector/array
fix ave/histo	global/per-atom/local scalars and vectors	global array, file
fix ave/correlate	global scalars	global array, file
fix store/state	per-atom vectors	per-atom vector/array

8.3.2 Use chunks to calculate system properties

In LAMMPS, "chunks" are collections of atoms, as defined by the *compute chunk/atom* command, which assigns each atom to a chunk ID (or to no chunk at all). The number of chunks and the assignment of chunk IDs to atoms can be static or change over time. Examples of "chunks" are molecules or spatial bins or atoms with similar values (e.g. coordination number or potential energy).

The per-atom chunk IDs can be used as input to two other kinds of commands, to calculate various properties of a system:

- *fix ave/chunk*
- any of the *compute */chunk* commands

Here a brief overview for each of the 4 kinds of chunk-related commands is provided. Then some examples are given of how to compute different properties with chunk commands.

Compute chunk/atom command:

This compute can assign atoms to chunks of various styles. Only atoms in the specified group and optional specified region are assigned to a chunk. Here are some possible chunk definitions:

atoms in same molecule	chunk ID = molecule ID
atoms of same atom type	chunk ID = atom type
all atoms with same atom property (charge, radius, etc)	chunk ID = output of compute property/atom
atoms in same cluster	chunk ID = output of *compute cluster/atom* command
atoms in same spatial bin	chunk ID = bin ID
atoms in same rigid body	chunk ID = molecule ID used to define rigid bodies
atoms with similar potential energy	chunk ID = output of *compute pe/atom*
atoms with same local defect structure	chunk ID = output of *compute centro/atom* or *compute coord/atom* command

Note that chunk IDs are integer values, so for atom properties or computes that produce a floating point value, they will be truncated to an integer. You could also use the compute in a variable that scales the floating point value to spread it across multiple integers.

Spatial bins can be of various kinds, e.g. 1d bins = slabs, 2d bins = pencils, 3d bins = boxes, spherical bins, cylindrical bins.

This compute also calculates the number of chunks *Nchunk*, which is used by other commands to tally per-chunk data. *Nchunk* can be a static value or change over time (e.g. the number of clusters). The chunk ID for an individual atom can also be static (e.g. a molecule ID), or dynamic (e.g. what spatial bin an atom is in as it moves).

Note that this compute allows the per-atom output of other *computes*, *fixes*, and *variables* to be used to define chunk IDs for each atom. This means you can write your own compute or fix to output a per-atom quantity to use as chunk ID. See the *Modify* doc pages for info on how to do this. You can also define a *per-atom variable* in the input script that uses a formula to generate a chunk ID for each atom.

Fix ave/chunk command:

This fix takes the ID of a *compute chunk/atom* command as input. For each chunk, it then sums one or more specified per-atom values over the atoms in each chunk. The per-atom values can be any atom property, such as velocity, force, charge, potential energy, kinetic energy, stress, etc. Additional keywords are defined for per-chunk properties like density and temperature. More generally any per-atom value generated by other *computes*, *fixes*, and *per-atom variables*, can be summed over atoms in each chunk.

Similar to other averaging fixes, this fix allows the summed per-chunk values to be time-averaged in various ways, and output to a file. The fix produces a global array as output with one row of values per chunk.

Compute */chunk commands:

The following computes operate on chunks of atoms to produce per-chunk values. Any compute whose style name ends in "/chunk" is in this category:

- *compute com/chunk*
- *compute gyration/chunk*
- *compute inertia/chunk*
- *compute msd/chunk*

- *compute property/chunk*
- *compute temp/chunk*
- *compute torque/chunk*
- *compute vcm/chunk*

They each take the ID of a *compute chunk/atom* command as input. As their names indicate, they calculate the center-of-mass, radius of gyration, moments of inertia, mean-squared displacement, temperature, torque, and velocity of center-of-mass for each chunk of atoms. The *compute property/chunk* command can tally the count of atoms in each chunk and extract other per-chunk properties.

The reason these various calculations are not part of the *fix ave/chunk command*, is that each requires a more complicated operation than simply summing and averaging over per-atom values in each chunk. For example, many of them require calculation of a center of mass, which requires summing mass*position over the atoms and then dividing by summed mass.

All of these computes produce a global vector or global array as output, with one or more values per chunk. The output can be used in various ways:

- As input to the *fix ave/time* command, which can write the values to a file and optionally time average them.
- As input to the *fix ave/histo* command to histogram values across chunks. E.g. a histogram of cluster sizes or molecule diffusion rates.
- As input to special functions of *equal-style variables*, like sum() and max() and ave(). E.g. to find the largest cluster or fastest diffusing molecule or average radius-of-gyration of a set of molecules (chunks).

Other chunk commands:

- *compute chunk/spread/atom*
- *compute reduce/chunk*

The *compute chunk/spread/atom* command spreads per-chunk values to each atom in the chunk, producing per-atom values as its output. This can be useful for outputting per-chunk values to a per-atom *dump file*. Or for using an atom's associated chunk value in an *atom-style variable*. Or as input to the *fix ave/chunk* command to spatially average per-chunk values calculated by a per-chunk compute.

The *compute reduce/chunk* command reduces a peratom value across the atoms in each chunk to produce a value per chunk. When used with the *compute chunk/spread/atom* command it can create peratom values that induce a new set of chunks with a second *compute chunk/atom* command.

Example calculations with chunks

Here are examples using chunk commands to calculate various properties:

1. Average velocity in each of 1000 2d spatial bins:

```
compute cc1 all chunk/atom bin/2d x 0.0 0.1 y lower 0.01 units reduced
fix 1 all ave/chunk 100 10 1000 cc1 vx vy file tmp.out
```

2. Temperature in each spatial bin, after subtracting a flow velocity:

```
compute cc1 all chunk/atom bin/2d x 0.0 0.1 y lower 0.1 units reduced
compute vbias all temp/profile 1 0 0 y 10
fix 1 all ave/chunk 100 10 1000 cc1 temp bias vbias file tmp.out
```

3. Center of mass of each molecule:

```
compute cc1 all chunk/atom molecule
compute myChunk all com/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

4. Total force on each molecule and ave/max across all molecules:

```
compute cc1 all chunk/atom molecule
fix 1 all ave/chunk 1000 1 1000 cc1 fx fy fz file tmp.out
variable xave equal ave(f_1[2])
variable xmax equal max(f_1[2])
thermo 1000
thermo_style custom step temp v_xave v_xmax
```

5. Histogram of cluster sizes:

```
compute cluster all cluster/atom 1.0
compute cc1 all chunk/atom c_cluster compress yes
compute size all property/chunk cc1 count
fix 1 all ave/histo 100 1 100 0 20 20 c_size mode vector ave running beyond ignore file tmp.histo
```

6. An example for using a per-chunk value to apply per-atom forces to compress individual polymer chains (molecules) in a mixture, is explained on the *compute chunk/spread/atom* command doc page.

7. An example for using one set of per-chunk values for molecule chunks, to create a second set of micelle-scale chunks (clustered molecules, due to hydrophobicity), is explained on the *compute reduce/chunk* command doc page.

8. An example for using one set of per-chunk values (dipole moment vectors) for molecule chunks, spreading the values to each atom in each chunk, then defining a second set of chunks as spatial bins, and using the *fix ave/chunk* command to calculate an average dipole moment vector for each bin. This example is explained on the *compute chunk/spread/atom* command doc page.

8.3.3 Using distributed grids

Added in version 22Dec2022.

LAMMPS has internal capabilities to create uniformly spaced grids which overlay the simulation domain. For 2d and 3d simulations these are 2d and 3d grids respectively. Conceptually a grid can be thought of as a collection of grid cells. Each grid cell can store one or more values (data).

The grid cells and data they store are distributed across processors. Each processor owns the grid cells (and data) whose center points lie within the spatial subdomain of the processor. If needed for its computations, a processor may also store ghost grid cells with their data.

Distributed grids can overlay orthogonal or triclinic simulation boxes; see the *Howto triclinic* doc page for an explanation of the latter. For a triclinic box, the grid cell shape conforms to the shape of the simulation domain, e.g. parallelograms instead of rectangles in 2d.

If the box size or shape changes during a simulation, the grid changes with it, so that it always overlays the entire simulation domain. For non-periodic dimensions, the grid size in that dimension matches the box size, as set by the *boundary* command for fixed or shrink-wrapped boundaries.

If load-balancing is invoked by the *balance* or *fix balance* commands, then the subdomain owned by a processor can change which may also change which grid cells they own.

Post-processing and visualization of grid cell data can be enabled by the *dump grid*, *dump grid/vtk*, and *dump image* commands. The latter has an optional *grid* keyword. The OVITO visualization tool also plans (as of Nov 2022) to add support for visualizing grid cell data (along with atoms) using *dump grid* output files as input.

Note

For developers, distributed grids are implemented within the code via two classes: Grid2d and Grid3d. These partition the grid across processors and have methods which allow forward and reverse communication of ghost grid data as well as load balancing. If you write a new compute or fix which needs a distributed grid, these are the classes to look at. A new pair style could use a distributed grid by having a fix define it. Please see the section on *using distributed grids within style classes* for a detailed description.

These are the commands which currently define or use distributed grids:

- *fix ttm/grid* - store electron temperature on grid
- *fix ave/grid* - time average per-atom or per-grid values
- *compute property/grid* - generate grid IDs and coords
- *dump grid* - output per-grid values in LAMMPS format
- *dump grid/vtk* - output per-grid values in VTK format
- *dump image grid* - include colored grid in output images
- *pair_style amoeba* - FFT grids
- *kspace_style pppm* (and variants) - FFT grids
- *kspace_style msm* (and variants) - MSM grids

The grids used by the *kspace_style* can not be referenced by an input script. However the grids and data created and used by the other commands can be.

A compute or fix command may create one or more grids (of different sizes). Each grid can store one or more data fields. A data field can be a single value per grid point (per-grid vector) or multiple values per grid point (per-grid array). See the *Howto output* doc page for an explanation of how per-grid data can be generated by some commands and used by other commands.

A command accesses grid data from a compute or fix using a *grid reference* with the following syntax:

- c_ID:gname:dname
- c_ID:gname:dname[I]
- f_ID:gname:dname
- f_ID:gname:dname[I]

The prefix "c_" or "f_" refers to the ID of the compute or fix; gname is the name of the grid, which is assigned by the compute or fix; dname is the name of the data field, which is also assigned by the compute or fix.

If the data field is a per-grid vector (one value per grid point), then no brackets are used to access the values. If the data field is a per-grid array (multiple values per grid point), then brackets are used to specify the column I of the array. I ranges from 1 to Ncol inclusive, where Ncol is the number of columns in the array and is defined by the compute or fix.

Currently, there are no per-grid variables implemented in LAMMPS. We may add this feature at some point.

8.3.4 Calculate temperature

Temperature is computed as kinetic energy divided by some number of degrees of freedom (and the Boltzmann constant). Since kinetic energy is a function of particle velocity, there is often a need to distinguish between a particle's advection velocity (due to some aggregate motion of particles) and its thermal velocity. The sum of the two is the particle's total velocity, but the latter is often what is wanted to compute a temperature.

LAMMPS has several options for computing temperatures, any of which can be used in *thermostatting* and *barostatting*. These *compute commands* calculate temperature:

- *compute temp*
- *compute temp/sphere*
- *compute temp/asphere*
- *compute temp/com*
- *compute temp/deform*
- *compute temp/partial*
- *compute temp/profile*
- *compute temp/ramp*
- *compute temp/region*

All but the first 3 calculate velocity biases directly (e.g. advection velocities) that are removed when computing the thermal temperature. *Compute temp/sphere* and *compute temp/asphere* compute kinetic energy for finite-size particles that includes rotational degrees of freedom. They both allow for velocity biases indirectly, via an optional extra argument which is another temperature compute that subtracts a velocity bias. This allows the translational velocity of spherical or aspherical particles to be adjusted in prescribed ways.

8.3.5 Calculate elastic constants

Elastic constants characterize the stiffness of a material. The formal definition is provided by the linear relation that holds between the stress and strain tensors in the limit of infinitesimal deformation. In tensor notation, this is expressed as

$$s_{ij} = C_{ijkl} e_{kl}$$

where the repeated indices imply summation. s_{ij} are the elements of the symmetric stress tensor. e_{kl} are the elements of the symmetric strain tensor. C_{ijkl} are the elements of the fourth rank tensor of elastic constants. In three dimensions, this tensor has 3^4 = 81 elements. Using Voigt notation, the tensor can be written as a 6x6 matrix, where C_{ij} is now the derivative of s_i w.r.t. e_j. Because s_i is itself a derivative w.r.t. e_i, it follows that C_{ij} is also symmetric, with at most $\frac{7\times6}{2}$ = 21 distinct elements.

At zero temperature, it is easy to estimate these derivatives by deforming the simulation box in one of the six directions using the *change_box* command and measuring the change in the stress tensor. A general-purpose script that does this is given in the examples/ELASTIC directory described on the *Examples* doc page.

Calculating elastic constants at finite temperature is more challenging, because it is necessary to run a simulation that performs time averages of differential properties. There are at least 3 ways to do this in LAMMPS. The most reliable way to do this is by exploiting the relationship between elastic constants, stress fluctuations, and the Born matrix, the second derivatives of energy w.r.t. strain *(Ray)*. The Born matrix calculation has been enabled by the *compute born/matrix* command, which works for any bonded or non-bonded potential in LAMMPS. The most expensive part of the calculation is the sampling of the stress fluctuations. Several examples of this method are provided in the examples/ELASTIC_T/BORN_MATRIX directory described on the *Examples* doc page.

A second way is to measure the change in average stress tensor in an NVT simulations when the cell volume undergoes a finite deformation. In order to balance the systematic and statistical errors in this method, the magnitude of the deformation must be chosen judiciously, and care must be taken to fully equilibrate the deformed cell before sampling the stress tensor. An example of this method is provided in the examples/ELASTIC_T/DEFORMATION directory described on the *Examples* doc page.

Another approach is to sample the triclinic cell fluctuations that occur in an NPT simulation. This method can also be slow to converge and requires careful post-processing *(Shinoda)*. We do not provide an example of this method.

A nice review of the advantages and disadvantages of all of these methods is provided in the paper by Clavier et al. *(Clavier)*.

(Ray) J. R. Ray and A. Rahman, J Chem Phys, 80, 4423 (1984).

(Shinoda) Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

(Clavier) G. Clavier, N. Desbiens, E. Bourasseau, V. Lachet, N. Brusselle-Dupend and B. Rousseau, Mol Sim, 43, 1413 (2017).

8.3.6 Calculate thermal conductivity

The thermal conductivity κ of a material can be measured in at least 4 ways using various options in LAMMPS. See the examples/KAPPA directory for scripts that implement the 4 methods discussed here for a simple Lennard-Jones fluid model. Also, see the *Howto viscosity* page for an analogous discussion for viscosity.

The thermal conductivity tensor κ is a measure of the propensity of a material to transmit heat energy in a diffusive manner as given by Fourier's law

$$J = -\kappa \cdot \text{grad}(T)$$

where J is the heat flux in units of energy per area per time and grad(T) is the spatial gradient of temperature. The thermal conductivity thus has units of energy per distance per time per degree K and is often approximated as an isotropic quantity, i.e. as a scalar.

The first method is to setup two thermostatted regions at opposite ends of a simulation box, or one in the middle and one at the end of a periodic box. By holding the two regions at different temperatures with a *thermostatting fix*, the energy added to the hot region should equal the energy subtracted from the cold region and be proportional to the heat flux moving between the regions. See the papers by *Ikeshoji and Hafskjold* and *Wirnsberger et al* for details of this idea. Note that thermostatting fixes such as *fix nvt*, *fix langevin*, and *fix temp/rescale* store the cumulative energy they add/subtract.

Alternatively, as a second method, the *fix heat* or *fix ehex* commands can be used in place of thermostats on each of two regions to add/subtract specified amounts of energy to both regions. In both cases, the resulting temperatures of the two regions can be monitored with the "compute temp/region" command and the temperature profile of the intermediate region can be monitored with the *fix ave/chunk* and *compute ke/atom* commands.

The third method is to perform a reverse non-equilibrium MD simulation using the *fix thermal/conductivity* command which implements the rNEMD algorithm of Muller-Plathe. Kinetic energy is swapped between atoms in two different layers of the simulation box. This induces a temperature gradient between the two layers which can be monitored with the *fix ave/chunk* and *compute ke/atom* commands. The fix tallies the cumulative energy transfer that it performs. See the *fix thermal/conductivity* command for details.

The fourth method is based on the Green-Kubo (GK) formula which relates the ensemble average of the auto-correlation of the heat flux to κ. The heat flux can be calculated from the fluctuations of per-atom potential and kinetic energies and per-atom stress tensor in a steady-state equilibrated simulation. This is in contrast to the two preceding non-equilibrium methods, where energy flows continuously between hot and cold regions of the simulation box.

The *compute heat/flux* command can calculate the needed heat flux and describes how to implement the Green_Kubo formalism using additional LAMMPS commands, such as the *fix ave/correlate* command to calculate the needed auto-correlation. See the page for the *compute heat/flux* command for an example input script that calculates the thermal conductivity of solid Ar via the GK formalism.

(Ikeshoji) Ikeshoji and Hafskjold, Molecular Physics, 81, 251-261 (1994).

(Wirnsberger) Wirnsberger, Frenkel, and Dellago, J Chem Phys, 143, 124104 (2015).

8.3.7 Calculate viscosity

The shear viscosity η of a fluid can be measured in at least 6 ways using various options in LAMMPS. See the examples/VISCOSITY directory for scripts that implement the 5 methods discussed here for a simple Lennard-Jones fluid model and 1 method for SPC/E water model. Also, see the *page on calculating thermal conductivity* for an analogous discussion for thermal conductivity.

η is a measure of the propensity of a fluid to transmit momentum in a direction perpendicular to the direction of velocity or momentum flow. Alternatively it is the resistance the fluid has to being sheared. It is given by

$$J = -\eta \cdot \text{grad}(V_{\text{stream}})$$

where J is the momentum flux in units of momentum per area per time. and $\text{grad}(V_{\text{stream}})$ is the spatial gradient of the velocity of the fluid moving in another direction, normal to the area through which the momentum flows. Viscosity thus has units of pressure-time.

The first method is to perform a non-equilibrium MD (NEMD) simulation by shearing the simulation box via the *fix deform* command, and using the *fix nvt/sllod* command to thermostat the fluid via the SLLOD equations of motion. Alternatively, as a second method, one or more moving walls can be used to shear the fluid in between them, again with some kind of thermostat that modifies only the thermal (non-shearing) components of velocity to prevent the fluid from heating up.

> **Note**
>
> A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

In both cases, the velocity profile setup in the fluid by this procedure can be monitored by the *fix ave/chunk* command, which determines $\text{grad}(V_{\text{stream}})$ in the equation above. E.g. the derivative in the y-direction of the V_x component of fluid motion or $\text{grad}(V_{\text{stream}}) = \frac{\text{d}V_x}{\text{dy}}$. The P_{xy} off-diagonal component of the pressure or stress tensor, as calculated by the *compute pressure* command, can also be monitored, which is the J term in the equation above. See the *Howto nemd* page for details on NEMD simulations.

The third method is to perform a reverse non-equilibrium MD simulation using the *fix viscosity* command which implements the rNEMD algorithm of Muller-Plathe. Momentum in one dimension is swapped between atoms in two different layers of the simulation box in a different dimension. This induces a velocity gradient which can be monitored with the *fix ave/chunk* command. The fix tallies the cumulative momentum transfer that it performs. See the *fix viscosity* command for details.

The fourth method is based on the Green-Kubo (GK) formula which relates the ensemble average of the auto-correlation of the stress/pressure tensor to η. This can be done in a fully equilibrated simulation which is in contrast to the two preceding non-equilibrium methods, where momentum flows continuously through the simulation box.

Here is an example input script that calculates the viscosity of liquid Ar via the GK formalism:

```
# Sample LAMMPS input script for viscosity of liquid Ar

units       real
variable    T equal 200.0       # run temperature
variable    Tinit equal 250.0   # equilibration temperature
variable    V equal vol
variable    dt equal 4.0
variable    p equal 400     # correlation length
variable    s equal 5       # sample interval
variable    d equal $p*$s   # dump interval

# convert from LAMMPS real units to SI

variable    kB equal 1.3806504e-23    # [J/K] Boltzmann
variable    atm2Pa equal 101325.0
variable    A2m equal 1.0e-10
variable    fs2s equal 1.0e-15
variable    convert equal ${atm2Pa}*${atm2Pa}*${fs2s}*${A2m}*${A2m}*${A2m}

# setup problem

dimension    3
boundary     p p p
lattice      fcc 5.376 orient x 1 0 0 orient y 0 1 0 orient z 0 0 1
region       box block 0 4 0 4 0 4
create_box   1 box
create_atoms 1 box
mass         1 39.948
pair_style   lj/cut 13.0
pair_coeff   * * 0.2381 3.405
timestep     ${dt}
thermo       $d

# equilibration and thermalization

velocity     all create ${Tinit} 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp ${Tinit} ${Tinit} 10 drag 0.2
run          8000

# viscosity calculation, switch to NVE if desired

velocity     all create $T 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp $T $T 10 drag 0.2
#unfix       NVT
#fix         NVE all nve

reset_timestep 0
variable     pxy equal pxy
variable     pxz equal pxz
variable     pyz equal pyz
fix          SS all ave/correlate $s $p $d &
             v_pxy v_pxz v_pyz type auto file S0St.dat ave running
variable     scale equal ${convert}/(${kB}*$T)*$V*$s*${dt}
```

(continues on next page)

(continued from previous page)

```
variable     v11 equal trap(f_SS[3])*${scale}
variable     v22 equal trap(f_SS[4])*${scale}
variable     v33 equal trap(f_SS[5])*${scale}
thermo_style custom step temp press v_pxy v_pxz v_pyz v_v11 v_v22 v_v33
run          100000
variable     v equal (v_v11+v_v22+v_v33)/3.0
variable     ndens equal count(all)/vol
print        "average viscosity: $v [Pa.s] @ $T K, ${ndens} atoms/A^3"
```

The fifth method is related to the above Green-Kubo method, but uses the Einstein formulation, analogous to the Einstein mean-square-displacement formulation for self-diffusivity. The time-integrated momentum fluxes play the role of Cartesian coordinates, whose mean-square displacement increases linearly with time at sufficiently long times.

The sixth is the periodic perturbation method, which is also a non-equilibrium MD method. However, instead of measuring the momentum flux in response to an applied velocity gradient, it measures the velocity profile in response to applied stress. A cosine-shaped periodic acceleration is added to the system via the *fix accelerate/cos* command, and the *compute viscosity/cos* command is used to monitor the generated velocity profile and remove the velocity bias before thermostatting.

Note

An article by *(Hess)* discussed the accuracy and efficiency of these methods.

(Daivis and Todd) Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

(Hess) Hess, B. The Journal of Chemical Physics 2002, 116 (1), 209-217.

8.3.8 Calculate diffusion coefficients

The diffusion coefficient *D* of a material can be measured in at least 2 ways using various options in LAMMPS. See the examples/DIFFUSE directory for scripts that implement the 2 methods discussed here for a simple Lennard-Jones fluid model.

The first method is to measure the mean-squared displacement (MSD) of the system, via the *compute msd* command. The slope of the MSD versus time is proportional to the diffusion coefficient. The instantaneous MSD values can be accumulated in a vector via the *fix vector* command, and a line fit to the vector to compute its slope via the *variable slope* function, and thus extract *D*.

The second method is to measure the velocity auto-correlation function (VACF) of the system, via the *compute vacf* command. The time-integral of the VACF is proportional to the diffusion coefficient. The instantaneous VACF values can be accumulated in a vector via the *fix vector* command, and time integrated via the *variable trap* function, and thus extract *D*.

8.3.9 Output structured data from LAMMPS

LAMMPS can output structured data with the *print* and *fix print* command. This gives you flexibility since you can build custom data formats that contain system properties, thermo data, and variables values. This output can be directed to the screen and/or to a file for post processing.

Writing the current system state, thermo data, variable values

Use the *print* command to output the current system state, which can include system properties, thermo data and variable values.

YAML

```
print """---
timestep: $(step)
pe: $(pe)
ke: $(ke)
..."""" file current_state.yaml screen no
```

Listing 1: current_state.yaml

```
---
timestep: 250
pe: -4.7774327356321810711
ke: 2.4962152903997174569
```

JSON

```
print """{
  "timestep": $(step),
  "pe": $(pe),
  "ke": $(ke)
}""" file current_state.json screen no
```

Listing 2: current_state.json

```
{
  "timestep": 250,
  "pe": -4.7774327356321810711,
  "ke": 2.4962152903997174569
}
```

YAML format thermo_style or dump_style output

Extracting data from log file

Added in version 24Mar2022.

LAMMPS supports the thermo style "yaml" and for "custom" style thermodynamic output the format can be changed to YAML with *thermo_modify line yaml*. This will produce a block of output in a compact YAML format - one "document" per run - of the following style:

```
---
keywords: ['Step', 'Temp', 'E_pair', 'E_mol', 'TotEng', 'Press', ]
data:
  - [100, 0.757453103239935, -5.7585054860159, 0, -4.62236133677021, 0.207261053624721, ]
  - [110, 0.759322359337036, -5.7614668389562, 0, -4.62251889318624, 0.194314975399602, ]
  - [120, 0.759372342462676, -5.76149365656489, 0, -4.62247073844943, 0.191600048851267, ]
  - [130, 0.756833027516501, -5.75777334823494, 0, -4.62255928350835, 0.208792327853067, ]
...
```

This data can be extracted and parsed from a log file using python with:

```
import re, yaml
try:
    from yaml import CSafeLoader as Loader
except ImportError:
    from yaml import SafeLoader as Loader

docs = ""
with open("log.lammps") as f:
    for line in f:
        m = re.search(r"^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)", line)
        if m: docs += m.group(0) + '\n'

thermo = list(yaml.load_all(docs, Loader=Loader))

print("Number of runs: ", len(thermo))
print(thermo[1]['keywords'][4], ' = ', thermo[1]['data'][2][4])
```

After loading the YAML data, *thermo* is a list containing a dictionary for each "run" where the tag "keywords" maps to the list of thermo header strings and the tag "data" has a list of lists where the outer list represents the lines of output and the inner list the values of the columns matching the header keywords for that step. The second print() command for example will print the header string for the fifth keyword of the second run and the corresponding value for the third output line of that run:

```
Number of runs:  2
TotEng  =  -4.62140097780047
```

Extracting data from dump file

Added in version 4May2022.

YAML format output has been added to multiple commands in LAMMPS, for example *dump yaml* or *fix ave/time* Depending on the kind of data being written, organization of the data or the specific syntax used may change, but the principles are very similar and all files should be readable with a suitable YAML parser. A simple example for this is given below:

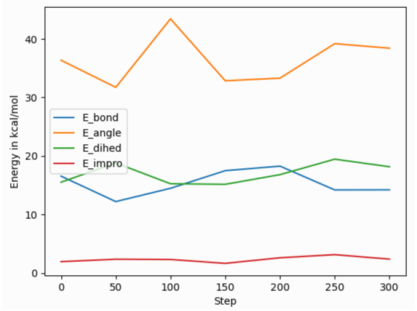
```
import yaml
try:
   from yaml import CSafeLoader as YamlLoader
except ImportError:
   from yaml import SafeLoader as YamlLoader

timesteps = []
with open("dump.yaml", "r") as f:
   data = yaml.load_all(f, Loader=YamlLoader)

   for d in data:
      print('Processing timestep %d' % d['timestep'])
      timesteps.append(d)

print('Read %d timesteps from yaml dump' % len(timesteps))
print('Second timestep: ', timesteps[1]['timestep'])
print('Box info: x: ' , timesteps[1]['box'][0], ' y:', timesteps[1]['box'][1], ' z:',timesteps[1]['box'][2])
print('First 5 per-atom columns: ', timesteps[1]['keywords'][0:5])
print('Corresponding 10th atom data: ', timesteps[1]['data'][9][0:5])
```

The corresponding output for a YAML dump command added to the "melt" example is:

```
Processing timestep 0
Processing timestep 50
Processing timestep 100
Processing timestep 150
Processing timestep 200
Processing timestep 250
Read 6 timesteps from yaml dump
Second timestep:  50
Box info: x:  [0, 16.795961913825074]  y: [0, 16.795961913825074]  z: [0, 16.795961913825074]
First 5 per-atom columns:  ['id', 'type', 'x', 'y', 'z']
Corresponding 10th atom data:  [10, 1, 4.43828, 0.968481, 0.108555]
```

Processing scalar data with Python

After reading and parsing the YAML format data, it can be easily imported for further processing and visualization with the pandas and matplotlib Python modules. Because of the organization of the data in the YAML format thermo output, it needs to be told to process only the 'data' part of the imported data to create a pandas data frame, and one needs to set the column names from the 'keywords' entry. The following Python script code example demonstrates this, and creates the image shown on the right of a simple plot of various bonded energy contributions versus the timestep from a run of the 'peptide' example input after changing the *thermo style* to 'yaml'. The properties to be used for x and y values can be conveniently selected through the keywords. Please note that those keywords can be changed to custom strings with the *thermo_modify colname* command.




```
import re, yaml
import pandas as pd
import matplotlib.pyplot as plt

try:
    from yaml import CSafeLoader as Loader
except ImportError:
    from yaml import SafeLoader as Loader

docs = ""
with open("log.lammps") as f:
    for line in f:
        m = re.search(r"^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)", line)
        if m: docs += m.group(0) + '\n'

thermo = list(yaml.load_all(docs, Loader=Loader))

df = pd.DataFrame(data=thermo[0]['data'], columns=thermo[0]['keywords'])
fig = df.plot(x='Step', y=['E_bond', 'E_angle', 'E_dihed', 'E_impro'], ylabel='Energy in kcal/mol')
plt.savefig('thermo_bondeng.png')
```

Processing vector data with Python

Global *vector* data as produced by *fix ave/time* uses a slightly different organization of the data. You still have the dictionary keys 'keywords' and 'data' for the column headers and the data. But the data is a dictionary indexed by the time step and for each step there are multiple rows of values each with a list of the averaged properties. This requires a slightly different processing, since the entire data cannot be directly imported into a single pandas DataFrame class instance. The following Python script example demonstrates how to read such data. The result will combine the data for the different steps into one large "multi-index" table. The pandas IndexSlice class can then be used to select data from this combined data frame.

```
import yaml
import pandas as pd

try:
    from yaml import CSafeLoader as Loader
except ImportError:
```

(continues on next page)

(continued from previous page)

```
    from yaml import SafeLoader as Loader

with open("ave.yaml") as f:
    ave = yaml.load(f, Loader=Loader)

keys = ave['keywords']
df = {}
for k in ave['data'].keys():
    df[k] = pd.DataFrame(data=ave['data'][k], columns=keys)

# create multi-index data frame
df = pd.concat(df)

# output only the first 3 value for steps 200 to 300 of the column Pressure
idx = pd.IndexSlice
print(df['Pressure'].loc[idx[200:300, 0:2]])
```

Processing scalar data with Perl

The ease of processing YAML data is not limited to Python. Here is an example for extracting and processing a LAMMPS log file with Perl instead.

```
use YAML::XS;

open(LOG, "log.lammps") or die("could not open log.lammps: $!");
my $file = "";
while(my $line = <LOG>) {
    if ($line =~ /^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)/) {
        $file .= $line;
    }
}
close(LOG);

# convert YAML to perl as nested hash and array references
my $thermo = Load $file;

# convert references to real arrays
my @keywords = @{$thermo->{'keywords'}};
my @data = @{$thermo->{'data'}};

# print first two columns
print("$keywords[0] $keywords[1]\n");
foreach (@data) {
    print("${$_}[0]  ${$_}[1]\n");
}
```

Writing continuous data during a simulation

The *fix print* command allows you to output an arbitrary string at defined times during a simulation run.

YAML

```
fix extra all print 50 """
- timestep: $(step)
  pe: $(pe)
  ke: $(ke)""" file output.yaml screen no
```

Listing 3: output.yaml

```
# Fix print output for fix extra
- timestep: 0
  pe: -6.77336805325924729
  ke: 4.4988750000000026219

- timestep: 50
  pe: -4.8082494418323200591
  ke: 2.5257981827119797558

- timestep: 100
  pe: -4.7875608875581505686
  ke: 2.5062598821985102582

- timestep: 150
  pe: -4.7471033686005483787
  ke: 2.466095925545450207

- timestep: 200
  pe: -4.7509052858544134068
  ke: 2.4701136792591693592

- timestep: 250
  pe: -4.7774327356321810711
  ke: 2.4962152903997174569
```

Post-processing of YAML files can be easily be done with Python and other scripting languages. In case of Python the *yaml* package allows you to load the data files and obtain a list of dictionaries.

```
import yaml

with open("output.yaml") as f:
  data = yaml.load(f, Loader=yaml.FullLoader)

print(data)
```

```
[{'timestep': 0, 'pe': -6.773368053259247, 'ke': 4.498875000000003},
 {'timestep': 50, 'pe': -4.80824944183232, 'ke': 2.5257981827119798},
 {'timestep': 100, 'pe': -4.787560887558151, 'ke': 2.5062598821985103},
 {'timestep': 150, 'pe': -4.747103368600548, 'ke': 2.46609592554545},
```

(continues on next page)

(continued from previous page)

```
{'timestep': 200, 'pe': -4.750905285854413, 'ke': 2.4701136792591694},
{'timestep': 250, 'pe': -4.777432735632181, 'ke': 2.4962152903997175}]
```

Line Delimited JSON (LD-JSON)

The JSON format itself is very strict when it comes to delimiters. For continuous output/streaming data it is beneficial use the *line delimited JSON* format. Each line represents one JSON object.

```
fix extra all print 50 """{"timestep": $(step), "pe": $(pe), "ke": $(ke)}""" &
   title "" file output.json screen no
```

Listing 4: output.json

```
{"timestep": 0, "pe": -6.77336805325924729, "ke": 4.4988750000000026219}
{"timestep": 50, "pe": -4.8082494418323200591, "ke": 2.5257981827119797558}
{"timestep": 100, "pe": -4.7875608875581505686, "ke": 2.5062598821985102582}
{"timestep": 150, "pe": -4.7471033686005483787, "ke": 2.466095925545450207}
{"timestep": 200, "pe": -4.7509052858544134068, "ke": 2.4701136792591693592}
{"timestep": 250, "pe": -4.7774327356321810711, "ke": 2.4962152903997174569}
```

One simple way to load this data into a Python script is to use the *pandas* package. It can directly load these files into a data frame:

```
import pandas as pd

data = pd.read_json('output.json', lines=True)
print(data)
```

```
   timestep       pe       ke
0         0 -6.773368  4.498875
1        50 -4.808249  2.525798
2       100 -4.787561  2.506260
3       150 -4.747103  2.466096
4       200 -4.750905  2.470114
5       250 -4.777433  2.496215
```

8.4 Force fields howto

8.4.1 CHARMM, AMBER, COMPASS, and DREIDING force fields

A compact summary of the concepts, definitions, and properties of force fields with explicit bonded interactions (like the ones discussed in this HowTo) is given in *(Gissinger)*.

A force field has 2 parts: the formulas that define it and the coefficients used for a particular system. Here we only discuss formulas implemented in LAMMPS that correspond to formulas commonly used in the CHARMM, AMBER, COMPASS, and DREIDING force fields. Setting coefficients is done either from special sections in an input data file via the *read_data* command or in the input script with commands like *pair_coeff* or *bond_coeff* and so on. See the *Tools* doc page for additional tools that can use CHARMM, AMBER, or Materials Studio generated files to assign force field coefficients and convert their output into LAMMPS input. LAMMPS input scripts can also be generated by charmm-gui.org.

CHARMM and AMBER

The CHARMM force field *(MacKerell)* and AMBER force field *(Cornell)* have potential energy function of the form

$$V = \sum_{bonds} E_b + \sum_{angles} E_a + \overbrace{\sum_{dihedral} E_d}^{\substack{\text{charmm}\\ \text{charmmfsw}}} + \sum_{impropers} E_i$$

$$+ \underbrace{\sum_{pairs} (E_{LJ} + E_{coul})}_{\substack{\text{lj/charmm/coul/charmm}\\ \text{lj/charmm/coul/charmm/implicit}\\ \text{lj/charmm/coul/long}\\ \text{lj/charmm/coul/msm}\\ \text{lj/charmmfsw/coul/charmmfsh}\\ \text{lj/charmmfsw/coul/long}}} + \sum_{special} E_s + \sum_{residues} \text{CMAP}(\phi,\psi)$$

The terms are computed by bond styles (relationship between two atoms), angle styles (between 3 atoms) , dihedral/improper styles (between 4 atoms), pair styles (non-covalently bonded pair interactions) and special bonds. The CMAP term (see *fix cmap* command for details) corrects for pairs of dihedral angles ("Correction MAP") to significantly improve the structural and dynamic properties of proteins in crystalline and solution environments *(Brooks)*. The AMBER force field does not include the CMAP term.

The interaction styles listed below compute force field formulas that are consistent with common options in CHARMM or AMBER. See each command's documentation for the formula it computes.

- *bond_style* harmonic
- *angle_style* charmm
- *dihedral_style* charmmfsh
- *dihedral_style* charmm
- *pair_style* lj/charmmfsw/coul/charmmfsh
- *pair_style* lj/charmmfsw/coul/long
- *pair_style* lj/charmm/coul/charmm
- *pair_style* lj/charmm/coul/charmm/implicit
- *pair_style* lj/charmm/coul/long
- *special_bonds* charmm
- *special_bonds* amber

The pair styles compute Lennard Jones (LJ) and Coulombic interactions with additional switching or shifting functions that ramp the energy and/or force smoothly to zero between an inner (a) and outer (b) cutoff. The older styles with *charmm* (not *charmmfsw* or *charmmfsh*) in their name compute the LJ and Coulombic interactions with an energy switching function (esw) $S(r)$ which ramps the energy smoothly to zero between the inner and outer cutoff. This can cause irregularities in pairwise forces (due to the discontinuous second derivative of energy at the boundaries of the switching region), which in some cases can result in complications in energy minimization and detectable artifacts in MD simulations.

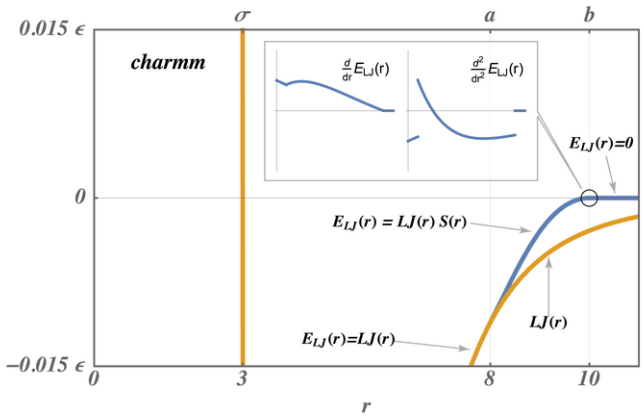
$$LJ(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

$$C(r) = \frac{C q_i q_j}{\varepsilon r}$$

$$S(r) = \frac{\left(b^2 - r^2\right)^2 \left(b^2 + 2r^2 - 3a^2\right)}{\left(b^2 - a^2\right)^3}$$

$$E_{LJ}(r) = \begin{cases} LJ(r), & r \le a \\ LJ(r)S(r), & a < r \le b \\ 0, & r > b \end{cases}$$

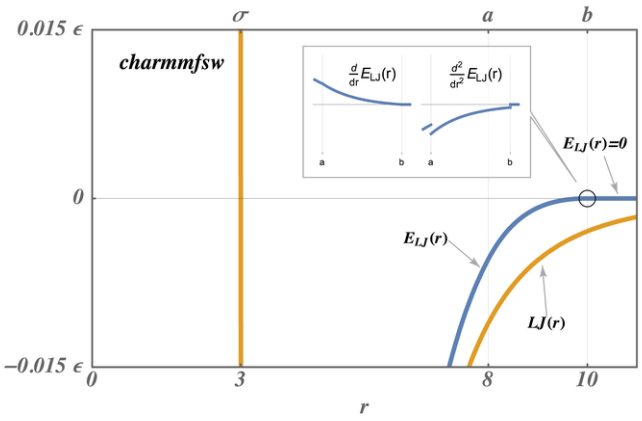
$$E_{coul}(r) = \begin{cases} C(r), & r \le a \\ C(r)S(r), & a < r \le b \\ 0, & r > b \end{cases}$$



The newer styles with *charmmfsw* or *charmmfsh* in their name replace energy switching with force switching (fsw) for LJ interactions and force shifting (fsh) functions for Coulombic interactions *(Steinbach)*

$$E_{LJ}(r) = \begin{cases} 4\varepsilon\sigma^6 \left(\dfrac{\sigma^6 - r^6}{r^{12}} - \dfrac{\sigma^6}{a^6 b^6} + \dfrac{1}{a^3 b^3} \right) & r \le a \\ \dfrac{4\varepsilon\sigma^6 \left(\sigma^6 \left(b^6 - r^6\right)^2 - b^3 r^6 \left(a^3 + b^3\right) \left(b^3 - r^3\right)^2 \right)}{b^6 r^{12} \left(b^6 - a^6\right)} & a < r \le b \\ 0, & r > b \end{cases}$$

$$E_{coul}(r) = \begin{cases} C(r)\dfrac{(b-r)^2}{rb^2}, & r \le b \\ 0, & r > b \end{cases}$$



These styles are used by LAMMPS input scripts generated by https://charmm-gui.org/ *(Brooks)*.

Note

For CHARMM, newer *charmmfsw* or *charmmfsh* styles were released in March 2017. We recommend they be used instead of the older *charmm* styles. See discussion of the differences on the *pair charmm* and *dihedral charmm* doc pages.

Note

The TIP3P water model is strongly recommended for use with the CHARMM force field. In fact, "using the SPC model with CHARMM parameters is a bad idea" and "to enable TIP4P style water in CHARMM, you would have to write a new pair style" . LAMMPS input scripts generated by Solution Builder on https://charmm-gui.org use TIP3P molecules for solvation. Any other water model can and probably will lead to false conclusions.

COMPASS

COMPASS is a general force field for atomistic simulation of common organic molecules, inorganic small molecules, and polymers which was developed using ab initio and empirical parameterization techniques *(Sun)*. See the *Tools* page for the msi2lmp tool for creating LAMMPS template input and data files from BIOVIA's Materials Studio files. Please note that the msi2lmp tool is very old and largely unmaintained, so it does not support all features of Materials Studio provided force field files, especially additions during the last decade. You should watch the output carefully and compare results, where possible. See *(Sun)* for a description of the COMPASS force field.

These interaction styles listed below compute force field formulas that are consistent with the COMPASS force field. See each command's documentation for the formula it computes.

- *bond_style* class2
- *angle_style* class2
- *dihedral_style* class2
- *improper_style* class2
- *pair_style* lj/class2
- *pair_style* lj/class2/coul/cut
- *pair_style* lj/class2/coul/long

- *special_bonds* lj/coul 0 0 1

DREIDING

DREIDING is a generic force field developed by the Goddard group at Caltech and is useful for predicting structures and dynamics of organic, biological and main-group inorganic molecules. The philosophy in DREIDING is to use general force constants and geometry parameters based on simple hybridization considerations, rather than individual force constants and geometric parameters that depend on the particular combinations of atoms involved in the bond, angle, or torsion terms. DREIDING has an *explicit hydrogen bond term* to describe interactions involving a hydrogen atom on very electronegative atoms (N, O, F). Unlike CHARMM or AMBER, the DREIDING force field has not been parameterized for considering solvents (like water).

See *(Mayo)* for a description of the DREIDING force field

The interaction styles listed below compute force field formulas that are consistent with the DREIDING force field. See each command's documentation for the formula it computes.

- *bond_style* harmonic
- *bond_style* morse
- *angle_style* cosine/squared
- *angle_style* harmonic
- *angle_style* cosine
- *angle_style* cosine/periodic
- *dihedral_style* charmm
- *improper_style* umbrella
- *pair_style* buck
- *pair_style* buck/coul/cut
- *pair_style* buck/coul/long
- *pair_style* lj/cut
- *pair_style* lj/cut/coul/cut
- *pair_style* lj/cut/coul/long
- *pair_style* hbond/dreiding/lj
- *pair_style* hbond/dreiding/morse
- *special_bonds* dreiding

(Gissinger) J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

(MacKerell) MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al (1998). J Phys Chem, 102, 3586 . https://doi.org/10.1021/jp973084f

(Cornell) Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman (1995). JACS 117, 5179-5197. https://doi.org/10.1021/ja00124a002

(Steinbach) Steinbach, Brooks (1994). J Comput Chem, 15, 667. https://doi.org/10.1002/jcc.540150702

(Brooks) Brooks, et al (2009). J Comput Chem, 30, 1545. https://onlinelibrary.wiley.com/doi/10.1002/jcc.21287

(Sun) Sun (1998). J. Phys. Chem. B, 102, 7338-7364. https://doi.org/10.1021/jp980939v

(Mayo) Mayo, Olfason, Goddard III (1990). J Phys Chem, 94, 8897-8909. https://doi.org/10.1021/j100389a010

8.4.2 AMOEBA and HIPPO force fields

The AMOEBA and HIPPO polarizable force fields were developed by Jay Ponder's group at the U Washington at St Louis. The LAMMPS implementation is based on Fortran 90 code provided by the Ponder group in their Tinker MD software.

The current implementation (July 2022) of AMOEBA in LAMMPS matches the version discussed in *(Ponder)*, *(Ren)*, and *(Shi)*. Likewise the current implementation of HIPPO in LAMMPS matches the version discussed in *(Rackers)*.

These force fields can be used when polarization effects are desired in simulations of water, organic molecules, and biomolecules including proteins, provided that parameterizations (Tinker PRM force field files) are available for the systems you are interested in. Files in the LAMMPS potentials directory with a "amoeba" or "hippo" suffix can be used. The Tinker distribution and website have additional force field files as well: https://github.com/TinkerTools/tinker/tree/release/params.

Note that currently, HIPPO can only be used for water systems, but HIPPO files for a variety of small organic and biomolecules are in preparation by the Ponder group. Those force field files will be included in the LAMMPS distribution when available.

To use the AMOEBA or HIPPO force fields, a simulation must be 3d, and fully periodic or fully non-periodic, and use an orthogonal (not triclinic) simulation box.

The AMOEBA and HIPPO force fields contain the following terms in their energy (U) computation. Further details for AMOEBA equations are in *(Ponder)*, further details for the HIPPO equations are in *(Rackers)*.

$$
\begin{aligned}
U &= U_{intermolecular} + U_{intramolecular} \\
U_{intermolecular} &= U_{hal} + U_{repulsion} + U_{dispersion} + U_{multipole} + U_{polar} + U_{qxfer} \\
U_{intramolecular} &= U_{bond} + U_{angle} + U_{torsion} + U_{oop} + U_{b\theta} + U_{UB} + U_{pitorsion} + U_{bitorsion}
\end{aligned}
$$

For intermolecular terms, the AMOEBA force field includes only the U_{hal}, $U_{multipole}$, U_{polar} terms. The HIPPO force field includes all but the U_{hal} term. In LAMMPS, these are all computed by the *pair_style amoeba or hippo* command. Note that the $U_{multipole}$ and U_{polar} terms in this formula are not the same for the AMOEBA and HIPPO force fields.

For intramolecular terms, the U_{bond}, U_{angle}, $U_{torsion}$, U_{oop} terms are computed by the *bond_style class2 angle_style amoeba*, *dihedral_style fourier*, and *improper_style amoeba* commands respectively. The *angle_style amoeba* command includes the $U_{b\theta}$ bond-angle cross term, and the U_{UB} term for a Urey-Bradley bond contribution between the I,K atoms in the IJK angle.

The $U_{pitorsion}$ term is computed by the *fix amoeba/pitorsion* command. It computes 6-body interaction between a pair of bonded atoms which each have 2 additional bond partners.

The $U_{bitorsion}$ term is computed by the *fix amoeba/bitorsion* command. It computes 5-body interaction between two 4-body torsions (dihedrals) which overlap, having 3 atoms in common.

These command doc pages have additional details on the terms they compute:

- *pair_style amoeba or hippo*
- *bond_style class2*
- *angle_style amoeba*
- *dihedral_style fourier*
- *improper_style amoeba*

- *fix amoeba/pitorsion*
- *fix amoeba/bitorsion*

To use the AMOEBA or HIPPO force fields in LAMMPS, use commands like the following appropriately in your input script. The only change needed for AMOEBA vs HIPPO simulation is for the *pair_style* and *pair_coeff* commands, as shown below. See examples/amoeba for example input scripts for both AMOEBA and HIPPO.

```
units           real                    # required
atom_style      amoeba
bond_style      class2                  # CLASS2 package
angle_style     amoeba
dihedral_style  fourier                 # EXTRA-MOLECULE package
improper_style  amoeba
                                        # required per-atom data
fix             amtype all property/atom i_amtype ghost yes
fix             extra all property/atom &
                i_amgroup i_ired i_xaxis i_yaxis i_zaxis d_pval ghost yes
fix             polaxe all property/atom i_polaxe

fix             pit all amoeba/pitorsion    # PiTorsion terms in FF
fix_modify      pit energy yes
                                        # Bitorsion terms in FF
fix             bit all amoeba/bitorsion bitorsion.ubiquitin.data
fix_modify      bit energy yes

read_data       data.ubiquitin fix amtype NULL "Tinker Types" &
                fix pit "pitorsion types" "PiTorsion Coeffs" &
                fix pit pitorsions PiTorsions &
                fix bit bitorsions BiTorsions

pair_style      amoeba                  # AMOEBA FF
pair_coeff      * * amoeba_ubiquitin.prm amoeba_ubiquitin.key

pair_style      hippo                   # HIPPO FF
pair_coeff      * * hippo_water.prm hippo_water.key

special_bonds   lj/coul 0.5 0.5 0.5 one/five yes    # 1-5 neighbors
```

The data file read by the *read_data* command should be created by the tools/tinker/tinker2lmp.py conversion program described below. It will create a section in the data file with the header "Tinker Types". A *fix property/atom* command for the data must be specified before the read_data command. In the example above the fix ID is *amtype*.

Similarly, if the system you are simulating defines AMOEBA/HIPPO pitorsion or bitorsion interactions, there will be entries in the data file for those interactions. They require a *fix amoeba/pitortion* and *fix amoeba/bitorsion* command be defined. In the example above, the IDs for these two fixes are *pit* and *bit*.

Of course, if the system being modeled does not have one or more of the following – bond, angle, dihedral, improper, pitorsion, bitorsion interactions – then the corresponding style and fix commands above do not need to be used. See the example scripts in examples/amoeba for water systems as examples; they are simpler than what is listed above.

The two *fix property/atom* commands with IDs (in the example above) *extra* and *polaxe* are also needed to define internal per-atom quantities used by the AMOEBA and HIPPO force fields.

The *pair_coeff* command used for either the AMOEBA or HIPPO force field takes two arguments for Tinker force field files, namely a PRM and KEY file. The keyfile can be specified as NULL and default values for a various settings will

be used. Note that these 2 files are meant to allow use of native Tinker files as-is. However LAMMPS does not support all the options which can be included in a Tinker PRM or KEY file. See specifics below.

A *special_bonds* command with the *one/five* option is required, since the AMOEBA/HIPPO force fields define weighting factors for not only 1-2, 1-3, 1-4 interactions, but also 1-5 interactions. This command will trigger a per-atom list of 1-5 neighbors to be generated. The AMOEBA and HIPPO force fields define their own custom weighting factors for all the 1-2, 1-3, 1-4, 1-5 terms which in the Tinker PRM and KEY files; they can be different for different terms in the force field.

In addition to the list above, these command doc pages have additional details:

- *atom_style amoeba*
- *fix property/atom*
- *special_bonds*

Tinker PRM and KEY files

A Tinker PRM file is composed of sections, each of which has multiple lines. This is the list of PRM sections LAMMPS knows how to parse and use. Any other sections are skipped:

- Angle Bending Parameters
- Atom Type Definitions
- Atomic Multipole Parameters
- Bond Stretching Parameters
- Charge Penetration Parameters
- Charge Transfer Parameters
- Dipole Polarizability Parameters
- Dispersion Parameters
- Force Field Definition
- Literature References
- Out-of-Plane Bend Parameters
- Pauli Repulsion Parameters
- Pi-Torsion Parameters
- Stretch-Bend Parameters
- Torsion-Torsion Parameters
- Torsional Parameters
- Urey-Bradley Parameters
- Van der Waals Pair Parameters
- Van der Waals Parameters

A Tinker KEY file is composed of lines, each of which has a keyword followed by zero or more parameters. This is the list of keywords LAMMPS knows how to parse and use in the same manner Tinker does. Any other keywords are skipped. The value in parenthesis is the default value for the keyword if it is not specified, or if the keyfile in the *pair_coeff* command is specified as NULL:

- a-axis (0.0)

- b-axis (0.0)
- c-axis (0.0)
- ctrn-cutoff (6.0)
- ctrn-taper (0.9 * ctrn-cutoff)
- cutoff
- delta-halgren (0.07)
- dewald (no long-range dispersion unless specified)
- dewald-alpha (0.4)
- dewald-cutoff (7.0)
- dispersion-cutoff (9.0)
- dispersion-taper (9.0 * dispersion-cutoff)
- dpme-grid
- dpme-order (4)
- ewald (no long-range electrostatics unless specified)
- ewald-alpha (0.4)
- ewald-cutoff (7.0)
- gamma-halgren (0.12)
- mpole-cutoff (9.0)
- mpole-taper (0.65 * mpole-cutoff)
- pcg-guess (enabled by default)
- pcg-noguess (disable pcg-guess if specified)
- pcg-noprecond (disable pcg-precond if specified)
- pcg-peek (1.0)
- pcg-precond (enabled by default)
- pewald-alpha (0.4)
- pme-grid
- pme-order (5)
- polar-eps (1.0e-6)
- polar-iter (100)
- polar-predict (no prediction operation unless specified)
- ppme-order (5)
- repulsion-cutoff (6.0)
- repulsion-taper (0.9 * repulsion-cutoff)
- taper
- usolve-cutoff (4.5)
- usolve-diag (2.0)

- vdw-cutoff (9.0)
- vdw-taper (0.9 * vdw-cutoff)

Tinker2lmp.py tool

This conversion tool is found in the tools/tinker directory. As shown in examples/amoeba/README, these commands produce the data files found in examples/amoeba, and also illustrate all the options available to use with the tinker2lmp.py script:

```
python tinker2lmp.py -xyz water_dimer.xyz -amoeba amoeba_water.prm -data data.water_dimer.
↪amoeba                # AMOEBA non-periodic system
python tinker2lmp.py -xyz water_dimer.xyz -hippo hippo_water.prm -data data.water_dimer.hippo
↪             # HIPPO non-periodic system
python tinker2lmp.py -xyz water_box.xyz -amoeba amoeba_water.prm -data data.water_box.amoeba -
↪pbc 18.643 18.643 18.643    # AMOEBA periodic system
python tinker2lmp.py -xyz water_box.xyz -hippo hippo_water.prm -data data.water_box.hippo -pbc 18.
↪643 18.643 18.643       # HIPPO periodic system
python tinker2lmp.py -xyz ubiquitin.xyz -amoeba amoeba_ubiquitin.prm -data data.ubiquitin.new -pbc
↪54.99 41.91 41.91 -bitorsion bitorsion.ubiquitin.data.new   # system with bitorsions
```

Switches and their arguments may be specified in any order.

The -xyz switch is required and specifies an input XYZ file as an argument. The format of this file is an extended XYZ format defined and used by Tinker for its input. Example *.xyz files are in the examples/amoeba directory. The file lists the atoms in the system. Each atom has the following information: Tinker species name (ignored by LAMMPS), xyz coordinates, Tinker numeric type, and a list of atom IDs the atom is bonded to.

Here is more information about the extended XYZ format defined and used by Tinker, and links to programs that convert standard PDB files to the extended XYZ format:

- https://openbabel.org/docs/current/FileFormats/Tinker_XYZ_format.html
- https://github.com/emleddin/pdbxyz-xyzpdb
- https://github.com/TinkerTools/tinker/blob/release/source/pdbxyz.f

The -amoeba or -hippo switch is required. It specifies an input AMOEBA or HIPPO PRM force field file as an argument. This should be the same file used by the *pair_style* command in the input script.

The -data switch is required. It specifies an output file name for the LAMMPS data file that will be produced.

For periodic systems, the -pbc switch is required. It specifies the periodic box size for each dimension (x,y,z). For a Tinker simulation these are specified in the KEY file.

The -bitorsion switch is only needed if the system contains Tinker bitorsion interactions. The data for each type of bitorsion interaction will be written to the specified file, and read by the *fix amoeba/bitorsion* command. The data includes 2d arrays of values to which splines are fit, and thus is not compatible with the LAMMPS data file format.

(Ponder) Ponder, Wu, Ren, Pande, Chodera, Schnieders, Haque, Mobley, Lambrecht, DiStasio Jr, M. Head-Gordon, Clark, Johnson, T. Head-Gordon, J Phys Chem B, 114, 2549-2564 (2010).

(Rackers) Rackers, Silva, Wang, Ponder, J Chem Theory Comput, 17, 7056-7084 (2021).

(Ren) Ren and Ponder, J Phys Chem B, 107, 5933 (2003).

(Shi) Shi, Xia, Zhang, Best, Wu, Ponder, Ren, J Chem Theory Comp, 9, 4046, 2013.

8.4.3 TIP3P water model

The TIP3P water model as implemented in CHARMM *(MacKerell)* specifies a 3-site rigid water molecule with charges and Lennard-Jones parameters assigned to each of the three atoms.

A suitable pair style with cutoff Coulomb would be:

- *pair_style lj/cut/coul/cut*

or these commands for a long-range Coulomb model:

- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/long/soft*
- *kspace_style pppm*
- *kspace_style pppm/disp*

In LAMMPS the *fix shake or fix rattle* command can be used to hold the two O-H bonds and the H-O-H angle rigid. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used. In case of rigid bonds also bond style *zero* and angle style *zero* can be used.

The table below lists the force field parameters (in real *units*) to for the water molecule atoms to run a rigid or flexible TIP3P-CHARMM model with a cutoff, the original 1983 TIP3P model *(Jorgensen)*, or a TIP3P model with parameters optimized for a long-range Coulomb solver (e.g. Ewald or PPPM in LAMMPS) *(Price)*. The K values can be used if a flexible TIP3P model (without fix shake) is desired, for rigid bonds/angles they are ignored.

Parameter	TIP3P-CHARMM	TIP3P (original)	TIP3P (Ewald)
O mass (amu)	15.9994	15.9994	15.9994
H mass (amu)	1.008	1.008	1.008
O charge (e)	-0.834	-0.834	-0.834
H charge (e)	0.417	0.417	0.417
LJ ε of OO (kcal/mole)	0.1521	0.1521	0.1020
LJ σ of OO (Å)	3.1507	3.1507	3.188
LJ ε of HH (kcal/mole)	0.0460	0.0	0.0
LJ σ of HH (Å)	0.4	1.0	1.0
LJ ε of OH (kcal/mole)	0.0836	0.0	0.0
LJ σ of OH (Å)	1.7753	1.0	1.0
K of OH bond (kcal/mole/Å^2)	450	450	450
r_0 of OH bond (Å)	0.9572	0.9572	0.9572
K of HOH angle (kcal/mole)	55.0	55.0	55.0
θ_0 of HOH angle	104.52°	104.52°	104.52°

Below is the code for a LAMMPS input file and a molecule file (tip3p.mol) of TIP3P water for use with the *molecule command* demonstrating how to set up a small bulk water system for TIP3P with rigid bonds.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box bond/types 1 angle/types 1 &
          extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

```

(continues on next page)

(continued from previous page)

```
pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.1521 3.1507
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 0.9574

angle_style zero
angle_coeff 1 104.52

molecule water tip3p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33

fix rigid all shake 0.001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000

reset_timestep 0
timestep 1.0
velocity all create 300.0 5463576
fix integrate all nvt temp 300 300 100.0

thermo_style custom step temp press etotal pe

thermo 1000
run 20000
write_data tip3p.data nocoeff
```

```
# Water molecule. TIP3P geometry

3 atoms
2 bonds
1 angles

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000

Types

1        1   # O
2        2   # H
3        2   # H

Charges

1       -0.834
2        0.417
3        0.417

Bonds
```

(continues on next page)

(continued from previous page)

```
1   1      1      2
2   1      1      3

Angles

1   1      2      1      3

Shake Flags

1 1
2 1
3 1

Shake Atoms

1 1 2 3
2 1 2 3
3 1 2 3

Shake Bond Types

1 1 1 1
2 1 1 1
3 1 1 1

Special Bond Counts

1 2 0 0
2 1 1 0
3 1 1 0

Special Bonds

1 2 3
2 1 3
3 1 2
```

Wikipedia also has a nice article on water models.

(MacKerell) MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem, 102, 3586 (1998).

(Jorgensen) Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

(Price) Price and Brooks, J Chem Phys, 121, 10096 (2004).

8.4.4 TIP4P water model

The four-point TIP4P rigid water model extends the traditional *three-point TIP3P* model by adding an additional site M, usually massless, where the charge associated with the oxygen atom is placed. This site M is located at a fixed distance away from the oxygen along the bisector of the HOH bond angle. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used. In case of rigid bonds also bond style *zero* and angle style *zero* can be used.

There are two ways to implement TIP4P water in LAMMPS:

1. Use a specially written pair style that uses the *TIP3P geometry* without the point M. The point M location is then implicitly derived from the other atoms or each water molecule and used during the force computation. The forces on M are then projected on the oxygen and the two hydrogen atoms. This is computationally very efficient, but the charge distribution in space is only correct within the tip4p labeled styles. So all other computations using charges will "see" the negative charge incorrectly on the oxygen atom.

   This can be done with the following pair styles for Coulomb with a cutoff:

   - *pair_style tip4p/cut*
   - *pair_style lj/cut/tip4p/cut*

   or these commands for a long-range Coulomb treatment:

   - *pair_style tip4p/long*
   - *pair_style lj/cut/tip4p/long*
   - *pair_style lj/long/tip4p/long*
   - *pair_style tip4p/long/soft*
   - *pair_style lj/cut/tip4p/long/soft*
   - *kspace_style pppm/tip4p*
   - *kspace_style pppm/disp/tip4p*

   The bond lengths and bond angles should be held fixed using the *fix shake* or *fix rattle* command, unless a parameterization for a flexible TIP4P model is used. The parameter sets listed below are all for rigid TIP4P model variants and thus the bond and angle force constants are not used and can be set to any legal value; only equilibrium length and angle are used.

2. Use an *explicit 4 point TIP4P geometry* where the oxygen atom carries no charge and the M point no Lennard-Jones interactions. Since *fix shake* or *fix rattle* may not be applied to this kind of geometry, *fix rigid or fix rigid/small* or its thermostatted variants are required to maintain a rigid geometry. This avoids some of the issues with respect to analysis and non-tip4p styles, but it is a more costly force computation (more atoms in the same volume and thus more neighbors in the neighbor lists) and requires a much shorter timestep for stable integration of the rigid body motion. Since no bonds or angles are required, they do not need to be defined and atom style charge would be sufficient for a bulk TIP4P water system. In order to avoid that LAMMPS produces an error due to the massless M site a tiny non-zero mass needs to be assigned.

The table below lists the force field parameters (in real *units*) to for a selection of popular variants of the TIP4P model. There is the rigid TIP4P model with a cutoff *(Jorgensen)*, the TIP4/Ice model *(Abascal1)*, the TIP4P/2005 model *(Abascal2)* and a version of TIP4P parameters adjusted for use with a long-range Coulombic solver (e.g. Ewald or PPPM in LAMMPS). Note that for implicit TIP4P models the OM distance is specified in the *pair_style* command, not as part of the pair coefficients.

Parameter	TIP4P (original)	TIP4P/Ice	TIP4P/2005	TIP4P (Ewald)
O mass (amu)	15.9994	15.9994	15.9994	15.9994
H mass (amu)	1.008	1.008	1.008	1.008
O or M charge (e)	-1.040	-1.1794	-1.1128	-1.04844
H charge (e)	0.520	0.5897	0.5564	0.52422
LJ ε of OO (kcal/mole)	0.1550	0.21084	0.1852	0.16275
LJ σ of OO (Å)	3.1536	3.1668	3.1589	3.16435
LJ ε of HH, MM, OH, OM, HM (kcal/mole)	0.0	0.0	0.0	0.0
LJ σ of HH, MM, OH, OM, HM (Å)	1.0	1.0	1.0	1.0
r_0 of OH bond (Å)	0.9572	0.9572	0.9572	0.9572
θ_0 of HOH angle	104.52°	104.52°	104.52°	104.52°
OM distance (Å)	0.15	0.1577	0.1546	0.1250

Note that the when using the TIP4P pair style, the neighbor list cutoff for Coulomb interactions is effectively extended by a distance 2 * (OM distance), to account for the offset distance of the fictitious charges on O atoms in water molecules. Thus it is typically best in an efficiency sense to use a LJ cutoff >= Coulomb cutoff + 2*(OM distance), to shrink the size of the neighbor list. This leads to slightly larger cost for the long-range calculation, so you can test the trade-off for your model. The OM distance and the LJ and Coulombic cutoffs are set in the *pair_style lj/cut/tip4p/long* command.

Below is the code for a LAMMPS input file using the implicit method and the *TIP3P molecule file*. Because the TIP4P charges are different from TIP3P they need to be reset (or the molecule file changed):

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box bond/types 1 angle/types 1 &
           extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/tip4p/cut 1 2 1 1 0.15 8.0
pair_coeff 1 1 0.1550 3.1536
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 0.9574

angle_style zero
angle_coeff 1 104.52

molecule water tip3p.mol  # this uses the TIP3P geometry
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
# must change charges for TIP4P
set type 1 charge -1.040
set type 2 charge  0.520

fix rigid all shake 0.001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000

reset_timestep 0
```

(continues on next page)

(continued from previous page)

```
timestep 1.0
velocity all create 300.0 5463576
fix integrate all nvt temp 300 300 100.0

thermo_style custom step temp press etotal pe

thermo 1000
run 20000
write_data tip4p-implicit.data nocoeff
```

Below is the code for a LAMMPS input file using the explicit method and a TIP4P molecule file. Because of using *fix rigid/small* no bonds need to be defined and thus no extra storage needs to be reserved for them, but we need to either switch to atom style full or use *fix property/atom mol* so that fix rigid/small can identify rigid bodies by their molecule ID. Also a *neigh_modify exclude* command is added to exclude computing intramolecular non-bonded interactions, since those are removed by the rigid fix anyway:

```
units real
atom_style charge
atom_modify map array
region box block -5 5 -5 5 -5 5
create_box 3 box

mass 1 15.9994
mass 2 1.008
mass 3 1.0e-100

pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.1550 3.1536
pair_coeff 2 2 0.0    1.0
pair_coeff 3 3 0.0    1.0

fix mol all property/atom mol ghost yes
molecule water tip4p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
neigh_modify exclude molecule/intra all

timestep 0.5
fix integrate all rigid/small molecule langevin 300.0 300.0 100.0 2345634

thermo_style custom step temp press etotal density pe ke
thermo 2000
run 40000
write_data tip4p-explicit.data nocoeff
```

```
# Water molecule. Explicit TIP4P geometry for use with fix rigid

4 atoms

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
```

(continues on next page)

(continued from previous page)

```
3   -0.75695   0.52032   0.00000
4    0.00000   0.08444   0.00000

Types

1        1   # O
2        2   # H
3        2   # H
4        3   # M

Charges

1        0.000
2        0.520
3        0.520
4       -1.040
```

Wikipedia also has a nice article on water models.

(Jorgensen) Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

(Abascal1) Abascal, Sanz, Fernandez, Vega, J Chem Phys, 122, 234511 (2005)
https://doi.org/10.1063/1.1931662

(Abascal2) Abascal, J Chem Phys, 123, 234505 (2005)
https://doi.org/10.1063/1.2121687

8.4.5 TIP5P water model

The five-point TIP5P rigid water model extends the *three-point TIP3P model* by adding two additional sites L, usually massless, where the charge associated with the oxygen atom is placed. These sites L are located at a fixed distance away from the oxygen atom, forming a tetrahedral angle that is rotated by 90 degrees from the HOH plane. Those sites thus somewhat approximate lone pairs of the oxygen and consequently improve the water structure to become even more "tetrahedral" in comparison to the *four-point TIP4P model*.

A suitable pair style with cutoff Coulomb would be:

- *pair_style lj/cut/coul/cut*

or these commands for a long-range model:

- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/long/soft*
- *kspace_style pppm*
- *kspace_style pppm/disp*

A TIP5P model *must* be run using a *rigid fix* since there is no other option to keep this kind of structure rigid in LAMMPS. In order to avoid that LAMMPS produces an error due to the massless L sites, those need to be assigned a tiny non-zero mass.

The table below lists the force field parameters (in real *units*) to for a the TIP5P model with a cutoff *(Mahoney)* and the TIP5P-E model *(Rick)* for use with a long-range Coulombic solver (e.g. Ewald or PPPM in LAMMPS).

Parameter	TIP5P	TIP5P-E
O mass (amu)	15.9994	15.9994
H mass (amu)	1.008	1.008
O charge (e)	0.0	0.0
L charge (e)	-0.241	-0.241
H charge (e)	0.241	0.241
LJ ε of OO (kcal/mole)	0.1600	0.1780
LJ σ of OO (Å)	3.1200	3.0970
LJ ε of HH, LL, OH, OL, HL (kcal/mole)	0.0	0.0
LJ σ of HH, LL, OH, OL, HL (Å)	1.0	1.0
r_0 of OH bond (Å)	0.9572	0.9572
θ_0 of HOH angle	104.52°	104.52°
OL distance (Å)	0.70	0.70
θ_0 of LOL angle	109.47°	109.47°

Below is the code for a LAMMPS input file for setting up a simulation of TIP5P water with a molecule file. Because of using *fix rigid/small* no bonds need to be defined and thus no extra storage needs to be reserved for them, but we need to either switch to atom style full or use *fix property/atom mol* so that fix rigid/small can identify rigid bodies by their molecule ID. Also a *neigh_modify exclude* command is added to exclude computing intramolecular non-bonded interactions, since those are removed by the rigid fix anyway:

```
units real
atom_style charge
atom_modify map array
region box block -5 5 -5 5 -5 5
create_box 3 box

mass 1 15.9994
mass 2 1.008
mass 3 1.0e-100

pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.160  3.12
pair_coeff 2 2 0.0    1.0
pair_coeff 3 3 0.0    1.0

fix mol all property/atom mol
molecule water tip5p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
neigh_modify exclude molecule/intra all

timestep 0.5
fix integrate all rigid/small molecule langevin 300.0 300.0 50.0 235664
reset_timestep 0

thermo_style custom step temp press etotal density pe ke
thermo 1000
run 20000
write_data tip5p.data nocoeff
```

```
# Water molecule. Explicit TIP5P geometry for use with fix rigid
```

(continues on next page)

(continued from previous page)

```
5 atoms

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000
4    0.00000  -0.46971   0.57154
5    0.00000  -0.46971  -0.57154

Types

1        1  # O
2        2  # H
3        2  # H
4        3  # L
5        3  # L

Charges

1        0.000
2        0.241
3        0.241
4       -0.241
5       -0.241
```

Wikipedia also has a nice article on water models.

(Mahoney) Mahoney, Jorgensen, J Chem Phys 112, 8910 (2000)

(Rick) Rick, J Chem Phys 120, 6085 (2004)

8.4.6 SPC water model

The SPC water model specifies a 3-site rigid water molecule with charges and Lennard-Jones parameters assigned to each of the three atoms. In LAMMPS the *fix shake* command can be used to hold the two O-H bonds and the H-O-H angle rigid. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used.

These are the additional parameters (in real units) to set for O and H atoms and the water molecule to run a rigid SPC model.

O mass = 15.9994
H mass = 1.008
O charge = -0.820
H charge = 0.410
LJ ε of OO = 0.1553
LJ σ of OO = 3.166
LJ ε, σ of OH, HH = 0.0
r_0 of OH bond = 1.0

θ_0 of HOH angle = 109.47°

Note that as originally proposed, the SPC model was run with a 9 Angstrom cutoff for both LJ and Coulomb terms. It can also be used with long-range electrostatic solvers (e.g. Ewald or PPPM in LAMMPS) without changing any of the parameters above, although it becomes a different model in that mode of usage.

The SPC/E (extended) water model is the same, except the partial charge assignments change:

O charge = -0.8476
H charge = 0.4238

See the *(Berendsen2)* reference for more details on both the SPC and SPC/E models.

Below is the code for a LAMMPS input file and a molecule file (spce.mol) of SPC/E water for use with the *molecule command* demonstrating how to set up a small bulk water system for SPC/E with rigid bonds.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box  bond/types 1 angle/types 1 &
                extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/coul/cut 10.0
pair_coeff 1 1 0.1553 3.166
pair_coeff 1 2 0.0    1.0
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 1.0

angle_style zero
angle_coeff 1 109.47

molecule water spce.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33

timestep 1.0
fix rigid     all shake 0.0001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000
velocity all create 300.0 5463576
fix integrate all nvt temp 300.0 300.0 100.0

thermo_style custom step temp press etotal density pe ke
thermo 1000
run 20000 upto
write_data spce.data nocoeff
```

```
# Water molecule. SPC/E geometry

3 atoms
2 bonds
1 angles

Coords

1    0.00000  -0.06461   0.00000
2    0.81649   0.51275   0.00000
3   -0.81649   0.51275   0.00000

Types

1        1   # O
2        2   # H
3        2   # H

Charges

1       -0.8476
2        0.4238
3        0.4238

Bonds

1   1      1      2
2   1      1      3

Angles

1   1      2      1      3

Shake Flags

1 1
2 1
3 1

Shake Atoms

1 1 2 3
2 1 2 3
3 1 2 3

Shake Bond Types

1 1 1 1
2 1 1 1
3 1 1 1

Special Bond Counts
```

(continues on next page)

(continued from previous page)

```
1 2 0 0
2 1 1 0
3 1 1 0

Special Bonds

1 2 3
2 1 3
3 1 2
```

Wikipedia also has a nice article on water models.

(Berendsen2) Berendsen, Grigera, Straatsma, J Phys Chem, 91, 6269-6271 (1987).

8.5 Packages howto

8.5.1 Finite-size spherical and aspherical particles

Typical MD models treat atoms or particles as point masses. Sometimes it is desirable to have a model with finite-size particles such as spheroids or ellipsoids or generalized aspherical bodies. The difference is that such particles have a moment of inertia, rotational energy, and angular momentum. Rotation is induced by torque coming from interactions with other particles.

LAMMPS has several options for running simulations with these kinds of particles. The following aspects are discussed in turn:

- atom styles
- pair potentials
- time integration
- computes, thermodynamics, and dump output
- rigid bodies composed of finite-size particles

Example input scripts for these kinds of models are in the body, colloid, dipole, ellipse, line, peri, pour, and tri directories of the *examples directory* in the LAMMPS distribution.

Atom styles

There are several *atom styles* that allow for definition of finite-size particles: sphere, dipole, ellipsoid, line, tri, peri, and body.

The sphere style defines particles that are spheroids and each particle can have a unique diameter and mass (or density). These particles store an angular velocity (omega) and can be acted upon by torque. The "set" command can be used to modify the diameter and mass of individual particles, after then are created.

The dipole style does not actually define finite-size particles, but is often used in conjunction with spherical particles, via a command like

```
atom_style hybrid sphere dipole
```

This is because when dipoles interact with each other, they induce torques, and a particle must be finite-size (i.e. have a moment of inertia) in order to respond and rotate. See the *atom_style dipole* command for details. The "set" command can be used to modify the orientation and length of the dipole moment of individual particles, after then are created.

The ellipsoid style defines particles that are ellipsoids and thus can be aspherical. Each particle has a shape, specified by 3 diameters, and mass (or density). These particles store an angular momentum and their orientation (quaternion), and can be acted upon by torque. They do not store an angular velocity (omega), which can be in a different direction than angular momentum, rather they compute it as needed. The "set" command can be used to modify the diameter, orientation, and mass of individual particles, after then are created. It also has a brief explanation of what quaternions are.

The line style defines line segment particles with two end points and a mass (or density). They can be used in 2d simulations, and they can be joined together to form rigid bodies which represent arbitrary polygons.

The tri style defines triangular particles with three corner points and a mass (or density). They can be used in 3d simulations, and they can be joined together to form rigid bodies which represent arbitrary particles with a triangulated surface.

The peri style is used with *Peridynamic models* and defines particles as having a volume, that is used internally in the *pair_style peri* potentials.

The body style allows for definition of particles which can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc. The body style is discussed in more detail on the *Howto body* doc page.

Note that if one of these atom styles is used (or multiple styles via the *atom_style hybrid* command), not all particles in the system are required to be finite-size or aspherical.

For example, in the ellipsoid style, if the 3 shape parameters are set to the same value, the particle will be a sphere rather than an ellipsoid. If the 3 shape parameters are all set to 0.0 or if the diameter is set to 0.0, it will be a point particle. In the line or tri style, if the lineflag or triflag is specified as 0, then it will be a point particle.

Some of the pair styles used to compute pairwise interactions between finite-size particles also compute the correct interaction with point particles as well, e.g. the interaction between a point particle and a finite-size particle or between two point particles. If necessary, *pair_style hybrid* can be used to ensure the correct interactions are computed for the appropriate style of interactions. Likewise, using groups to partition particles (ellipsoids versus spheres versus point particles) will allow you to use the appropriate time integrators and temperature computations for each class of particles. See the doc pages for various commands for details.

Also note that for *2d simulations*, atom styles sphere and ellipsoid still use 3d particles, rather than as circular disks or ellipses. This means they have the same moment of inertia as the 3d object. When temperature is computed, the correct degrees of freedom are used for rotation in a 2d versus 3d system.

Pair potentials

When a system with finite-size particles is defined, the particles will only rotate and experience torque if the force field computes such interactions. These are the various *pair styles* that generate torque:

- *pair_style gran/history*
- *pair_style gran/hertz*
- *pair_style gran/no_history*
- *pair_style dipole/cut*
- *pair_style gayberne*
- *pair_style resquared*
- *pair_style brownian*

- *pair_style lubricate*
- *pair_style line/lj*
- *pair_style tri/lj*
- *pair_style body/nparticle*

The granular pair styles are used with spherical particles. The dipole pair style is used with the dipole atom style, which could be applied to spherical or ellipsoidal particles. The GayBerne and REsquared potentials require ellipsoidal particles, though they will also work if the 3 shape parameters are the same (a sphere). The Brownian and lubrication potentials are used with spherical particles. The line, tri, and body potentials are used with line segment, triangular, and body particles respectively.

Time integration

There are several fixes that perform time integration on finite-size spherical particles, meaning the integrators update the rotational orientation and angular velocity or angular momentum of the particles:

- *fix nve/sphere*
- *fix nvt/sphere*
- *fix npt/sphere*

Likewise, there are 3 fixes that perform time integration on ellipsoidal particles:

- *fix nve/asphere*
- *fix nvt/asphere*
- *fix npt/asphere*

The advantage of these fixes is that those which thermostat the particles include the rotational degrees of freedom in the temperature calculation and thermostatting. The *fix langevin* command can also be used with its *omgea* or *angmom* options to thermostat the rotational degrees of freedom for spherical or ellipsoidal particles. Other thermostatting fixes only operate on the translational kinetic energy of finite-size particles.

These fixes perform constant NVE time integration on line segment, triangular, and body particles:

- *fix nve/line*
- *fix nve/tri*
- *fix nve/body*

Note that for mixtures of point and finite-size particles, these integration fixes can only be used with *groups* which contain finite-size particles.

Computes, thermodynamics, and dump output

There are several computes that calculate the temperature or rotational energy of spherical or ellipsoidal particles:

- *compute temp/sphere*
- *compute temp/asphere*
- *compute erotate/sphere*
- *compute erotate/asphere*

These include rotational degrees of freedom in their computation. If you wish the thermodynamic output of temperature or pressure to use one of these computes (e.g. for a system entirely composed of finite-size particles), then the compute can be defined and the *thermo_modify* command used. Note that by default thermodynamic quantities will be calculated with a temperature that only includes translational degrees of freedom. See the *thermo_style* command for details.

These commands can be used to output various attributes of finite-size particles:

- *dump custom*
- *compute property/atom*
- *dump local*
- *compute body/local*

Attributes include the dipole moment, the angular velocity, the angular momentum, the quaternion, the torque, the end-point and corner-point coordinates (for line and tri particles), and sub-particle attributes of body particles.

Rigid bodies composed of finite-size particles

The *fix rigid* command treats a collection of particles as a rigid body, computes its inertia tensor, sums the total force and torque on the rigid body each timestep due to forces on its constituent particles, and integrates the motion of the rigid body.

If any of the constituent particles of a rigid body are finite-size particles (spheres or ellipsoids or line segments or triangles), then their contribution to the inertia tensor of the body is different than if they were point particles. This means the rotational dynamics of the rigid body will be different. Thus a model of a dimer is different if the dimer consists of two point masses versus two spheroids, even if the two particles have the same mass. Finite-size particles that experience torque due to their interaction with other particles will also impart that torque to a rigid body they are part of.

See the "fix rigid" command for example of complex rigid-body models it is possible to define in LAMMPS.

Note that the *fix shake* command can also be used to treat 2, 3, or 4 particles as a rigid body, but it always assumes the particles are point masses.

Also note that body particles cannot be modeled with the *fix rigid* command. Body particles are treated by LAMMPS as single particles, though they can store internal state, such as a list of sub-particles. Individual body particles are typically treated as rigid bodies, and their motion integrated with a command like *fix nve/body*. Interactions between pairs of body particles are computed via a command like *pair_style body/nparticle*.

8.5.2 Granular models

Granular system are composed of spherical particles with a diameter, as opposed to point particles. This means they have an angular velocity and torque can be imparted to them to cause them to rotate.

To run a simulation of a granular model, you will want to use the following commands:

- *atom_style sphere*
- *fix nve/sphere*
- *fix gravity*

This compute

- *compute erotate/sphere*

calculates rotational kinetic energy which can be *output with thermodynamic info*. The compute

- *compute fabric*

calculates various versions of the fabric tensor for granular and non-granular pair styles.

Use one of these 4 pair potentials, which compute forces and torques between interacting pairs of particles:

- *pair_style gran/history*
- *pair_style gran/no_history*
- *pair_style gran/hertzian*
- *pair_style granular*

These commands implement fix options specific to granular systems:

- *fix freeze*
- *fix pour*
- *fix viscous*
- *fix wall/gran*
- *fix wall/gran/region*

The fix style *freeze* zeroes both the force and torque of frozen atoms, and should be used for granular system instead of the fix style *setforce*.

To model heat conduction, one must add the temperature and heatflow atom variables with:

- *fix property/atom*

a temperature integration fix

- *fix heat/flow*

and a heat conduction option defined in both

- *pair_style granular*
- *fix wall/gran*

For computational efficiency, you can eliminate needless pairwise computations between frozen atoms by using this command:

- *neigh_modify* exclude

Note

By default, for 2d systems, granular particles are still modeled as 3d spheres, not 2d discs (circles), meaning their moment of inertia will be the same as in 3d. If you wish to model granular particles in 2d as 2d discs, see the note on this topic on the *Howto 2d* doc page, where 2d simulations are discussed.

To add custom granular contact models, see the *modifying granular sub-models page*.

8.5.3 Body particles

Overview:

In LAMMPS, body particles are generalized finite-size particles. Individual body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc. Note that other kinds of finite-size spherical and aspherical particles are also supported by LAMMPS, such as spheres, ellipsoids, line segments, and triangles, but they are simpler entities than body particles. See the *Howto spherical* page for a general overview of all these particle types.

Body particles are used via the *atom_style body* command. It takes a body style as an argument. The current body styles supported by LAMMPS are as follows. The name in the first column is used as the *bstyle* argument for the *atom_style body* command.

nparticle	rigid body with N sub-particles
rounded/polygon	2d polygons with N vertices
rounded/polyhedron	3d polyhedra with N vertices, E edges and F faces

The body style determines what attributes are stored for each body and thus how they can be used to compute pairwise body/body or bond/non-body (point particle) interactions. More details of each style are described below.

More styles may be added in the future. See the *page on creating new body styles* for details on how to add a new body style to the code.

When to use body particles:

You should not use body particles to model a rigid body made of simpler particles (e.g. point, sphere, ellipsoid, line segment, triangular particles), if the interaction between pairs of rigid bodies is just the summation of pairwise interactions between the simpler particles. LAMMPS already supports this kind of model via the *fix rigid* command. Any of the numerous pair styles that compute interactions between simpler particles can be used. The *fix rigid* command time integrates the motion of the rigid bodies. All of the standard LAMMPS commands for thermostatting, adding constraints, performing output, etc will operate as expected on the simple particles.

By contrast, when body particles are used, LAMMPS treats an entire body as a single particle for purposes of computing pairwise interactions, building neighbor lists, migrating particles between processors, output of particles to a dump file, etc. This means that interactions between pairs of bodies or between a body and non-body (point) particle need to be encoded in an appropriate pair style. If such a pair style were to mimic the *fix rigid* model, it would need to loop over the entire collection of interactions between pairs of simple particles within the two bodies, each time a single body/body interaction was computed.

Thus it only makes sense to use body particles and develop such a pair style, when particle/particle interactions are more complex than what the *fix rigid* command can already calculate. For example, consider particles with one or more of the following attributes:

- represented by a surface mesh
- represented by a collection of geometric entities (e.g. planes + spheres)
- deformable
- internal stress that induces fragmentation

For these models, the interaction between pairs of particles is likely to be more complex than the summation of simple pairwise interactions. An example is contact or frictional forces between particles with planar surfaces that interpenetrate. Likewise, the body particle may store internal state, such as a stress tensor used to compute a fracture criterion.

These are additional LAMMPS commands that can be used with body particles of different styles

fix nve/body	integrate motion of a body particle in NVE ensemble
fix nvt/body	ditto for NVT ensemble
fix npt/body	ditto for NPT ensemble
fix nph/body	ditto for NPH ensemble
compute body/local	store sub-particle attributes of a body particle
compute temp/body	compute temperature of body particles
dump local	output sub-particle attributes of a body particle
dump image	output body particle attributes as an image

The pair styles currently defined for use with specific body styles are listed in the sections below.

Note that for all the body styles, if the data file defines a general triclinic box, then the orientation of the body particle and its corresponding 6 moments of inertia and other orientation-dependent values should reflect the fact the body is defined withing a general triclinic box with edge vectors **A**,**B**,**C**. LAMMPS will rotate the box to convert it to a restricted triclinic box. This operation will also rotate the orientation of the body particles. See the *Howto triclinic* doc page for more details. The sections below highlight the orientation-dependent values specific to each body style.

Specifics of body style nparticle:

The *nparticle* body style represents body particles as a rigid body with a variable number N of sub-particles. It is provided as a vanilla, prototypical example of a body particle, although as mentioned above, the *fix rigid* command already duplicates its functionality.

The atom_style body command for this body style takes two additional arguments:

```
atom_style body nparticle Nmin Nmax
Nmin = minimum # of sub-particles in any body in the system
Nmax = maximum # of sub-particles in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each entry in the *Bodies* section of the data file:

```
atom-ID 1 M
N
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
```

where M = 6 + 3*N, and N is the number of sub-particles in the body particle.

The integer line has a single value N. The floating point line(s) list 6 moments of inertia followed by the coordinates of the N sub-particles (x1 to zN) as 3N values. These values can be listed on as many lines as you wish; see the *read_data* command for more details.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each sub-particle are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file, as is the total mass of the body particle.

Note that if the data file defines a general triclinic simulation box, these sub-particle displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/nparticle* command can be used with this body style to compute body/body and body/non-body interactions.

Specifics of body style rounded/polygon:

The *rounded/polygon* body style represents body particles as a 2d polygon with a variable number of N vertices. This style can only be used for 2d models; see the *boundary* command. See the *pair_style body/rounded/polygon* page for a diagram of two squares with rounded circles at the vertices. Special cases for N = 1 (circle) and N = 2 (rod with rounded ends) can also be specified.

One use of this body style is for 2d discrete element models, as described in *Fraige*.

Similar to body style *nparticle*, the atom_style body command for this body style takes two additional arguments:

```
atom_style body rounded/polygon Nmin Nmax
Nmin = minimum # of vertices in any body in the system
Nmax = maximum # of vertices in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each body in the *Bodies* section of the data file:

```
atom-ID 1 M
N
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
diameter
```

where M = 6 + 3*N + 1, and N is the number of vertices in the body particle.

The integer line has a single value N. The floating point line(s) list 6 moments of inertia, followed by the coordinates of the N vertices (x1 to zN) as 3N values (with z = 0.0 for each), followed by a diameter value = the rounded diameter of the circle that surrounds each vertex. The diameter value can be different for each body particle. These floating-point values can be listed on as many lines as you wish; see the *read_data* command for more details.

Note

It is important that the vertices for each polygonal body particle be listed in order around its perimeter, so that edges can be inferred. LAMMPS does not check that this is the case.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each vertex are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file.

For example, the following information would specify a square particle whose edge length is sqrt(2) and rounded diameter is 1.0. The orientation of the square is aligned with the xy coordinate axes which is consistent with the 6 moments of inertia: ixx iyy izz ixy ixz iyz = 1 1 4 0 0 0. Note that only Izz matters in 2D simulations.

```
3 1 19
4
1 1 4 0 0 0
-0.7071 -0.7071 0
-0.7071 0.7071 0
0.7071 0.7071 0
0.7071 -0.7071 0
1.0
```

A rod in 2D, whose length is 4.0, mass 1.0, rounded at two ends by circles of diameter 0.5, is specified as follows:

```
1 1 13
2
1 1 1.33333 0 0 0
-2 0 0
2 0 0
0.5
```

A disk, whose diameter is 3.0, mass 1.0, is specified as follows:

```
1 1 10
1
1 1 4.5 0 0 0
0 0 0
3.0
```

Note that if the data file defines a general triclinic simulation box, these polygon vertex displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/rounded/polygon* command can be used with this body style to compute body/body interactions. The *fix wall/body/polygon* command can be used with this body style to compute the interaction of body particles with a wall.

Specifics of body style rounded/polyhedron:

The *rounded/polyhedron* body style represents body particles as a 3d polyhedron with a variable number of N vertices, E edges and F faces. This style can only be used for 3d models; see the *boundary* command. See the "pair_style body/rounded/polygon" page for a diagram of a two 2d squares with rounded circles at the vertices. A 3d cube with rounded spheres at the 8 vertices and 12 rounded edges would be similar. Special cases for N = 1 (sphere) and N = 2 (rod with rounded ends) can also be specified.

This body style is for 3d discrete element models, as described in *Wang*.

Similar to body style *rounded/polygon*, the atom_style body command for this body style takes two additional arguments:

```
atom_style body rounded/polyhedron Nmin Nmax
Nmin = minimum # of vertices in any body in the system
Nmax = maximum # of vertices in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each entry in the *Bodies* section of the data file:

```
atom-ID 3 M
N E F
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
0 1
1 2
2 3
...
0 1 2 -1
0 2 3 -1
...
1 2 3 4
diameter
```

where M = 6 + 3*N + 2*E + 4*F + 1, and N is the number of vertices in the body particle, E = number of edges, F = number of faces. For N = 1 or 2, the format is simpler. E and F are ignored and no edges or faces are listed, so that M = 6 + 3*N + 1.

The integer line has three values: number of vertices (N), number of edges (E) and number of faces (F). The floating point line(s) list 6 moments of inertia followed by the coordinates of the N vertices (x1 to zN) as 3N values, followed by 2E vertex indices corresponding to the end points of the E edges, then 4*F vertex indices defining F faces. The last value is the diameter value = the rounded diameter of the sphere that surrounds each vertex. The diameter value can be different for each body particle. These floating-point values can be listed on as many lines as you wish; see the *read_data* command for more details.

Note that vertices are numbered from 0 to N-1 inclusive. The order of the 2 vertices in each edge does not matter. Faces can be triangles or quadrilaterals. In both cases 4 vertices must be specified. For a triangle the 4th vertex is -1. The 4 vertices within each triangle or quadrilateral face should be ordered by the right-hand rule so that the normal vector of the face points outwards from the center of mass. For polyhedron with faces with more than 4 vertices, you should split the complex face into multiple simple faces, each of which is a triangle or quadrilateral.

Note

If a face is a quadrilateral then its 4 vertices must be co-planar. LAMMPS does not check that this is the case. If you have a quad-face of a polyhedron that is not planar (e.g. a cube whose vertices have been randomly displaced), then you should represent the single quad face as two triangle faces instead.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each vertex are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file.

For example, the following information would specify a cubic particle whose edge length is 2.0 and rounded diameter is 0.5. The orientation of the cube is aligned with the xyz coordinate axes which is consistent with the 6 moments of inertia: ixx iyy izz ixy ixz iyz = 0.667 0.667 0.667 0 0 0.

```
1 3 79
8 12 6
0.667 0.667 0.667 0 0 0
1 1 1
```

(continues on next page)

(continued from previous page)

```
1 -1 1
-1 -1 1
-1 1 1
1 1 -1
1 -1 -1
-1 -1 -1
-1 1 -1
0 1
1 2
2 3
3 0
4 5
5 6
6 7
7 4
0 4
1 5
2 6
3 7
0 1 2 3
4 5 6 7
0 1 5 4
1 2 6 5
2 3 7 6
3 0 4 7
0.5
```

A rod in 3D, whose length is 4.0, mass 1.0 and rounded at two ends by circles of diameter 0.5, is specified as follows:

```
1 3 13
2 1 1
0 1.33333 1.33333 0 0 0
-2 0 0
2 0 0
0.5
```

A sphere whose diameter is 3.0 and mass 1.0, is specified as follows:

```
1 3 10
1 1 1
0.9 0.9 0.9 0 0 0
0 0 0
3.0
```

The number of edges and faces for a rod or sphere must be listed, but is ignored.

Note that if the data file defines a general triclinic simulation box, these polyhedron vertex displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/rounded/polhedron* command can be used with this body style to compute body/body interactions. The *fix wall/body/polyhedron* command can be used with this body style to compute the interaction of body particles with a wall.

Output specifics for all body styles:

For the *compute body/local* and *dump local* commands, all 3 of the body styles described on his page produces one datum for each of the N vertices (of sub-particles) in a body particle. The datum has 3 values:

```
1 = x position of vertex (or sub-particle)
2 = y position of vertex
3 = z position of vertex
```

These values are the current position of the vertex within the simulation domain, not a displacement from the center-of-mass (COM) of the body particle itself. These values are calculated using the current COM and orientation of the body particle.

The *dump image* command and its *body* keyword can be used to render body particles.

For the *nparticle* body style, each body is drawn as a collection of spheres, one for each sub-particle. The size of each sphere is determined by the *bflag1* parameter for the *body* keyword. The *bflag2* argument is ignored.

For the *rounded/polygon* body style, each body is drawn as a polygon with N line segments. For the *rounded/polyhedron* body style, each face of each body is drawn as a polygon with N line segments. The drawn diameter of each line segment is determined by the *bflag1* parameter for the *body* keyword. The *bflag2* argument is ignored.

Note that for both the *rounded/polygon* and *rounded/polyhedron* styles, line segments are drawn between the pairs of vertices. Depending on the diameters of the line segments this may be slightly different than the physical extent of the body as calculated by the *pair_style rounded/polygon* or *pair_style rounded/polyhedron* commands. Conceptually, the pair styles define the surface of a 2d or 3d body by lines or planes that are tangent to the finite-size spheres of specified diameter which are placed on each vertex position.

(Fraige) F. Y. Fraige, P. A. Langston, A. J. Matchett, J. Dodds, Particuology, 6, 455 (2008).

(Wang) J. Wang, H. S. Yu, P. A. Langston, F. Y. Fraige, Granular Matter, 13, 1 (2011).

8.5.4 Bonded particle models

The BPM package implements bonded particle models which can be used to simulate mesoscale solids. Solids are constructed as a collection of particles, which each represent a coarse-grained region of space much larger than the atomistic scale. Particles within a solid region are then connected by a network of bonds to provide solid elasticity.

Unlike traditional bonds in molecular dynamics, the equilibrium bond length can vary between bonds. Bonds store the reference state. This includes setting the equilibrium length equal to the initial distance between the two particles, but can also include data on the bond orientation for rotational models. This produces a stress-free initial state. Furthermore, bonds are allowed to break under large strains, producing fracture. The examples/bpm directory has sample input scripts for simulations of the fragmentation of an impacted plate and the pouring of extended, elastic bodies. See *(Clemmer)* for more general information on the approach and the LAMMPS implementation. Example movies illustrating some of these capabilities are found at https://www.lammps.org/movies.html#bpmpackage.

Bonds can be created using a *read data* or *create bonds* command. Alternatively, a *molecule* template with bonds can be used with *fix deposit* or *fix pour* to create solid grains.

In this implementation, bonds store their reference state when they are first computed in the setup of the first simulation run. Data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not reset the reference state of a bond. Bonds that are created midway into a run, such as those created by pouring grains using *fix pour*, are initialized on that timestep.

Currently, there are two types of bonds included in the BPM package. The first bond style, *bond bpm/spring*, only applies pairwise, central body forces. Point particles must have *bond atom style* and may be thought of as nodes in

a spring network. Alternatively, the second bond style, *bond bpm/rotational*, resolves tangential forces and torques arising with the shearing, bending, and twisting of the bond due to rotation or displacement of particles. Particles are similar to those used in the *granular package*, *atom style sphere*. However, they must also track the current orientation of particles and store bonds, and therefore use a *bpm/sphere atom style*. This also requires a unique integrator *fix nve/bpm/sphere* which numerically integrates orientation similar to *fix nve/asphere*.

In addition to bond styles, a new pair style *pair bpm/spring* was added to accompany the bpm/spring bond style. This pair style is simply a hookean repulsion with similar velocity damping as its sister bond style.

Bond data can be output using a combination of standard LAMMPS commands. A list of IDs for bonded atoms can be generated using the *compute property/local* command. Various properties of bonds can be computed using the *compute bond/local* command. This command allows one to access data saved to the bond's history, such as the reference length of the bond. More information on bond history data can be found on the documentation pages for the specific BPM bond styles. Finally, this data can be output using a *dump local* command. As one may output many columns from the same compute, the *dump modify colname* option may be used to provide more helpful column names. An example of this procedure is found in /examples/bpm/pour/. External software, such as OVITO, can read these dump files to render bond data.

As bonds can be broken between neighbor list builds, the *special_bonds* command works differently for BPM bond styles. There are two possible settings which determine how pair interactions work between bonded particles. First, one can overlay pair forces with bond forces such that all bonded particles also feel pair interactions. This can be accomplished by setting the *overlay/pair* keyword present in all bpm bond styles to *yes* and requires using the following special bond settings

```
special_bonds lj/coul 1 1 1
```

Alternatively, one can turn off all pair interactions between bonded particles. Unlike *bond quartic*, this is not done by subtracting pair forces during the bond computation, but rather by dynamically updating the special bond list. This is the default behavior of BPM bond styles and is done by updating the 1-2 special bond list as bonds break. To do this, LAMMPS requires *newton* bond off such that all processors containing an atom know when a bond breaks. Additionally, one must use the following special bond settings

```
special_bonds lj 0 1 1 coul 1 1 1
```

These settings accomplish two goals. First, they turn off 1-3 and 1-4 special bond lists, which are not currently supported for BPMs. As BPMs often have dense bond networks, generating 1-3 and 1-4 special bond lists is expensive. By setting the lj weight for 1-2 bonds to zero, this turns off pairwise interactions. Even though there are no charges in BPM models, setting a nonzero coul weight for 1-2 bonds ensures all bonded neighbors are still included in the neighbor list in case bonds break between neighbor list builds. If bond breakage is disabled during a simulation run by setting the *break* keyword to *no*, a zero coul weight for 1-2 bonds can be used to exclude bonded atoms from the neighbor list builds

```
special_bonds lj 0 1 1 coul 0 1 1
```

This can be useful for post-processing, or to determine pair interaction properties between distinct bonded particles.

To monitor the fracture of bonds in the system, all BPM bond styles have the ability to record instances of bond breakage to output using the *dump local* command. Since one may frequently output a list of broken bonds and the time they broke, the *dump modify* option *header no* may be useful to avoid repeatedly printing the header of the dump file. An example of this procedure is found in /examples/bpm/impact/. Additionally, one can use *compute nbond/atom* to tally the current number of bonds per atom.

See the *Howto* page on broken bonds for more information.

While LAMMPS has many utilities to create and delete bonds, *only* the following are currently compatible with BPM bond styles:

- *create_bonds*
- *delete_bonds*
- *fix bond/create*
- *fix bond/break*
- *fix bond/swap*

Note

The *create_bonds* command requires certain *special_bonds* settings. To subtract pair interactions, one will need to switch between different *special_bonds* settings in the input script. An example is found in examples/bpm/impact.

(Clemmer) Clemmer, Monti, Lechman, Soft Matter, 20, 1702 (2024).

8.5.5 Polarizable models

In polarizable force fields the charge distributions in molecules and materials respond to their electrostatic environments. Polarizable systems can be simulated in LAMMPS using three methods:

- the fluctuating charge method, implemented in the *QEQ* package,
- the adiabatic core-shell method, implemented in the *CORESHELL* package,
- the thermalized Drude dipole method, implemented in the *DRUDE* package.

The fluctuating charge method calculates instantaneous charges on interacting atoms based on the electronegativity equalization principle. It is implemented in the *fix qeq* which is available in several variants. It is a relatively efficient technique since no additional particles are introduced. This method allows for charge transfer between molecules or atom groups. However, because the charges are located at the interaction sites, off-plane components of polarization cannot be represented in planar molecules or atom groups.

The two other methods share the same basic idea: polarizable atoms are split into one core atom and one satellite particle (called shell or Drude particle) attached to it by a harmonic spring. Both atoms bear a charge and they represent collectively an induced electric dipole. These techniques are computationally more expensive than the QEq method because of additional particles and bonds. These two charge-on-spring methods differ in certain features, with the core-shell model being normally used for ionic/crystalline materials, whereas the so-called Drude model is normally used for molecular systems and fluid states.

The core-shell model is applicable to crystalline materials where the high symmetry around each site leads to stable trajectories of the core-shell pairs. However, bonded atoms in molecules can be so close that a core would interact too strongly or even capture the Drude particle of a neighbor. The Drude dipole model is relatively more complex in order to remedy this and other issues. Specifically, the Drude model includes specific thermostatting of the core-Drude pairs and short-range damping of the induced dipoles.

The three polarization methods can be implemented through a self-consistent calculation of charges or induced dipoles at each timestep. In the fluctuating charge scheme this is done by the matrix inversion method in *fix qeq/point*, but for core-shell or Drude-dipoles the relaxed-dipoles technique would require an slow iterative procedure. These self-consistent solutions yield accurate trajectories since the additional degrees of freedom representing polarization are massless. An alternative is to attribute a mass to the additional degrees of freedom and perform time integration using an extended Lagrangian technique. For the fluctuating charge scheme this is done by *fix qeq/dynamic*, and for the

charge-on-spring models by the methods outlined in the next two sections. The assignment of masses to the additional degrees of freedom can lead to unphysical trajectories if care is not exerted in choosing the parameters of the polarizable models and the simulation conditions.

In the core-shell model the vibration of the shells is kept faster than the ionic vibrations to mimic the fast response of the polarizable electrons. But in molecular systems thermalizing the core-Drude pairs at temperatures comparable to the rest of the simulation leads to several problems (kinetic energy transfer, too short a timestep, etc.) In order to avoid these problems the relative motion of the Drude particles with respect to their cores is kept "cold" so the vibration of the core-Drude pairs is very slow, approaching the self-consistent regime. In both models the temperature is regulated using the velocities of the center of mass of core+shell (or Drude) pairs, but in the Drude model the actual relative core-Drude particle motion is thermostatted separately as well.

8.5.6 Adiabatic core/shell model

The adiabatic core-shell model by *Mitchell and Fincham* is a simple method for adding polarizability to a system. In order to mimic the electron shell of an ion, a satellite particle is attached to it. This way the ions are split into a core and a shell where the latter is meant to react to the electrostatic environment inducing polarizability. See the *Howto polarizable* page for a discussion of all the polarizable models available in LAMMPS.

Technically, shells are attached to the cores by a spring force f = k*r where k is a parameterized spring constant and r is the distance between the core and the shell. The charges of the core and the shell add up to the ion charge, thus q(ion) = q(core) + q(shell). This setup introduces the ion polarizability (alpha) given by alpha = q(shell)^2 / k. In a similar fashion the mass of the ion is distributed on the core and the shell with the core having the larger mass.

To run this model in LAMMPS, *atom_style full* can be used since atom charge and bonds are needed. Each kind of core/shell pair requires two atom types and a bond type. The core and shell of a core/shell pair should be bonded to each other with a harmonic bond that provides the spring force. For example, a data file for NaCl, as found in examples/coreshell, has this format:

```
432  atoms  # core and shell atoms
216  bonds  # number of core/shell springs

4    atom types  # 2 cores and 2 shells for Na and Cl
2    bond types

0.0 24.09597 xlo xhi
0.0 24.09597 ylo yhi
0.0 24.09597 zlo zhi

Masses       # core/shell mass ratio = 0.1

1 20.690784  # Na core
2 31.90500   # Cl core
3 2.298976   # Na shell
4 3.54500    # Cl shell

Atoms

1   1   2   1.5005    0.00000000   0.00000000   0.00000000 # core of core/shell pair 1
2   1   4  -2.5005    0.00000000   0.00000000   0.00000000 # shell of core/shell pair 1
3   2   1   1.5056    4.01599500   4.01599500   4.01599500 # core of core/shell pair 2
4   2   3  -0.5056    4.01599500   4.01599500   4.01599500 # shell of core/shell pair 2
(...)
```

(continues on next page)

(continued from previous page)

```
Bonds  # Bond topology for spring forces

1    2    1    2  # spring for core/shell pair 1
2    2    3    4  # spring for core/shell pair 2
(...)
```

Non-Coulombic (e.g. Lennard-Jones) pairwise interactions are only defined between the shells. Coulombic interactions are defined between all cores and shells. If desired, additional bonds can be specified between cores.

The *special_bonds* command should be used to turn-off the Coulombic interaction within core/shell pairs, since that interaction is set by the bond spring. This is done using the *special_bonds* command with a 1-2 weight = 0.0, which is the default value. It needs to be considered whether one has to adjust the *special_bonds* weighting according to the molecular topology since the interactions of the shells are bypassed over an extra bond.

Note that this core/shell implementation does not require all ions to be polarized. One can mix core/shell pairs and ions without a satellite particle if desired.

Since the core/shell model permits distances of r = 0.0 between the core and shell, a pair style with a "cs" suffix needs to be used to implement a valid long-range Coulombic correction. Several such pair styles are provided in the CORESHELL package. See *this page* for details. All of the core/shell enabled pair styles require the use of a long-range Coulombic solver, as specified by the *kspace_style* command. Either the PPPM or Ewald solvers can be used.

For the NaCL example problem, these pair style and bond style settings are used:

```
pair_style      born/coul/long/cs 20.0 20.0
pair_coeff      * *      0.0 1.000   0.00  0.00   0.00
pair_coeff      3 3    487.0 0.23768 0.00  1.05   0.50 #Na-Na
pair_coeff      3 4 145134.0 0.23768 0.00  6.99   8.70 #Na-Cl
pair_coeff      4 4 405774.0 0.23768 0.00 72.40 145.40 #Cl-Cl

bond_style      harmonic
bond_coeff      1 63.014 0.0
bond_coeff      2 25.724 0.0
```

When running dynamics with the adiabatic core/shell model, the following issues should be considered. The relative motion of the core and shell particles corresponds to the polarization, hereby an instantaneous relaxation of the shells is approximated and a fast core/shell spring frequency ensures a nearly constant internal kinetic energy during the simulation. Thermostats can alter this polarization behavior, by scaling the internal kinetic energy, meaning the shell will not react freely to its electrostatic environment. Therefore it is typically desirable to decouple the relative motion of the core/shell pair, which is an imaginary degree of freedom, from the real physical system. To do that, the *compute temp/cs* command can be used, in conjunction with any of the thermostat fixes, such as *fix nvt* or *fix langevin*. This compute uses the center-of-mass velocity of the core/shell pairs to calculate a temperature, and ensures that velocity is what is rescaled for thermostatting purposes. This compute also works for a system with both core/shell pairs and non-polarized ions (ions without an attached satellite particle). The *compute temp/cs* command requires input of two groups, one for the core atoms, another for the shell atoms. Non-polarized ions which might also be included in the treated system should not be included into either of these groups, they are taken into account by the *group-ID* (second argument) of the compute. The groups can be defined using the *group *type** command. Note that to perform thermostatting using this definition of temperature, the *fix modify temp* command should be used to assign the compute to the thermostat fix. Likewise the *thermo_modify temp* command can be used to make this temperature be output for the overall system.

For the NaCl example, this can be done as follows:

```
group cores type 1 2
group shells type 3 4
```

(continues on next page)

(continued from previous page)

```
compute CSequ all temp/cs cores shells
fix thermoberendsen all temp/berendsen 1427 1427 0.4    # thermostat for the true physical system
fix thermostatequ all nve                               # integrator as needed for the berendsen thermostat
fix_modify thermoberendsen temp CSequ
thermo_modify temp CSequ                                # output of center-of-mass derived temperature
```

The pressure for the core/shell system is computed via the regular LAMMPS convention by *treating the cores and shells as individual particles*. For the thermo output of the pressure as well as for the application of a barostat, it is necessary to use an additional *pressure* compute based on the default *temperature* and specifying it as a second argument in *fix modify* and *thermo_modify* resulting in:

```
(...)
compute CSequ all temp/cs cores shells
compute thermo_press_lmp all pressure thermo_temp       # pressure for individual particles
thermo_modify temp CSequ press thermo_press_lmp         # modify thermo to regular pressure
fix press_bar all npt temp 300 300 0.04 iso 0 0 0.4
fix_modify press_bar temp CSequ press thermo_press_lmp  # pressure modification for correct kinetic␣
↪scalar
```

If *compute temp/cs* is used, the decoupled relative motion of the core and the shell should in theory be stable. However numerical fluctuation can introduce a small momentum to the system, which is noticeable over long trajectories. Therefore it is recommendable to use the *fix momentum* command in combination with *compute temp/cs* when equilibrating the system to prevent any drift.

When initializing the velocities of a system with core/shell pairs, it is also desirable to not introduce energy into the relative motion of the core/shell particles, but only assign a center-of-mass velocity to the pairs. This can be done by using the *bias* keyword of the *velocity create* command and assigning the *compute temp/cs* command to the *temp* keyword of the *velocity* command, e.g.

```
velocity all create 1427 134 bias yes temp CSequ
velocity all scale 1427 temp CSequ
```

To maintain the correct polarizability of the core/shell pairs, the kinetic energy of the internal motion shall remain nearly constant. Therefore the choice of spring force and mass ratio need to ensure much faster relative motion of the two atoms within the core/shell pair than their center-of-mass velocity. This allows the shells to effectively react instantaneously to the electrostatic environment and limits energy transfer to or from the core/shell oscillators. This fast movement also dictates the timestep that can be used.

The primary literature of the adiabatic core/shell model suggests that the fast relative motion of the core/shell pairs only allows negligible energy transfer to the environment. The mentioned energy transfer will typically lead to a small drift in total energy over time. This internal energy can be monitored using the *compute chunk/atom* and *compute temp/chunk* commands. The internal kinetic energies of each core/shell pair can then be summed using the sum() special function of the *variable* command. Or they can be time/averaged and output using the *fix ave/time* command. To use these commands, each core/shell pair must be defined as a "chunk". If each core/shell pair is defined as its own molecule, the molecule ID can be used to define the chunks. If cores are bonded to each other to form larger molecules, the chunks can be identified by the *fix property/atom* via assigning a core/shell ID to each atom using a special field in the data file read by the *read_data* command. This field can then be accessed by the *compute property/atom* command, to use as input to the *compute chunk/atom* command to define the core/shell pairs as chunks.

For example if core/shell pairs are the only molecules:

```
read_data NaCl_CS_x0.1_prop.data
compute prop all property/atom molecule
compute cs_chunk all chunk/atom c_prop
```

(continues on next page)

(continued from previous page)

```
compute cstherm all temp/chunk cs_chunk temp internal com yes cdof 3.0     # note the chosen degrees␣
↪of freedom for the core/shell pairs
fix ave_chunk all ave/time 10 1 10 c_cstherm file chunk.dump mode vector
```

For example if core/shell pairs and other molecules are present:

```
fix csinfo all property/atom i_CSID                    # property/atom command
read_data NaCl_CS_x0.1_prop.data fix csinfo NULL CS-Info  # atom property added in the data-file
compute prop all property/atom i_CSID
(...)
```

The additional section in the date file would be formatted like this:

```
CS-Info       # header of additional section

1   1         # column 1 = atom ID, column 2 = core/shell ID
2   1
3   2
4   2
5   3
6   3
7   4
8   4
(...)
```

(Mitchell and Fincham) Mitchell, Fincham, J Phys Condensed Matter, 5, 1031-1038 (1993).

(Fincham) Fincham, Mackrodt and Mitchell, J Phys Condensed Matter, 6, 393-404 (1994).

8.5.7 Drude induced dipoles

The thermalized Drude model represents induced dipoles by a pair of charges (the core atom and the Drude particle) connected by a harmonic spring. See the *Howto polarizable* doc page for a discussion of all the polarizable models available in LAMMPS.

The Drude model has a number of features aimed at its use in molecular systems (*Lamoureux and Roux*):

- Thermostatting of the additional degrees of freedom associated with the induced dipoles at very low temperature, in terms of the reduced coordinates of the Drude particles with respect to their cores. This makes the trajectory close to that of relaxed induced dipoles.
- Consistent definition of 1-2 to 1-4 neighbors. A core-Drude particle pair represents a single (polarizable) atom, so the special screening factors in a covalent structure should be the same for the core and the Drude particle. Drude particles have to inherit the 1-2, 1-3, 1-4 special neighbor relations from their respective cores.
- Stabilization of the interactions between induced dipoles. Drude dipoles on covalently bonded atoms interact too strongly due to the short distances, so an atom may capture the Drude particle of a neighbor, or the induced dipoles within the same molecule may align too much. To avoid this, damping at short range can be done by Thole functions (for which there are physical grounds). This Thole damping is applied to the point charges composing the induced dipole (the charge of the Drude particle and the opposite charge on the core, not to the total charge of the core atom).

A detailed tutorial covering the usage of Drude induced dipoles in LAMMPS is on the *here*.

As with the core-shell model, the cores and Drude particles should appear in the data file as standard atoms. The same holds for the springs between them, which are described by standard harmonic bonds. The nature of the atoms (core, Drude particle or non-polarizable) is specified via the *fix drude* command. The special list of neighbors is automatically refactored to account for the equivalence of core and Drude particles as regards special 1-2 to 1-4 screening. It may be necessary to use the *extra/special/per/atom* keyword of the *read_data* command. If using *fix shake*, make sure no Drude particle is in this fix group.

There are three ways to thermostat the Drude particles at a low temperature: use either *fix langevin/drude* for a Langevin thermostat, or *fix drude/transform/** for a Nose-Hoover thermostat, or *fix tgnvt/drude* for a temperature-grouped Nose-Hoover thermostat. The first and third require use of the command *comm_modify vel yes*. The second requires two separate integration fixes like *nvt* or *npt*. The correct temperatures of the reduced degrees of freedom can be calculated using the *compute temp/drude*. This requires also to use the command *comm_modify vel yes*.

Short-range damping of the induced dipole interactions can be achieved using Thole functions through the *pair style thole* in *pair_style hybrid/overlay* with a Coulomb pair style. It may be useful to use *coul/long/cs* or similar from the CORESHELL package if the core and Drude particle come too close, which can cause numerical issues.

(Lamoureux and Roux) G. Lamoureux, B. Roux, J. Chem. Phys 119, 3025 (2003)

8.5.8 Tutorial for Thermalized Drude oscillators in LAMMPS

This tutorial explains how to use Drude oscillators in LAMMPS to simulate polarizable systems using the DRUDE package. As an illustration, the input files for a simulation of 250 phenol molecules are documented. First of all, LAMMPS has to be compiled with the DRUDE package activated. Then, the data file and input scripts have to be modified to include the Drude dipoles and how to handle them.

Example input scripts available: examples/PACKAGES/drude

Overview of Drude induced dipoles

Polarizable atoms acquire an induced electric dipole moment under the action of an external electric field, for example the electric field created by the surrounding particles. Drude oscillators represent these dipoles by two fixed charges: the core (DC) and the Drude particle (DP) bound by a harmonic potential. The Drude particle can be thought of as the electron cloud whose center can be displaced from the position of the corresponding nucleus.

The sum of the masses of a core-Drude pair should be the mass of the initial (unsplit) atom, $m_C + m_D = m$. The sum of their charges should be the charge of the initial (unsplit) atom, $q_C + q_D = q$. A harmonic potential between the core and Drude partners should be present, with force constant k_D and an equilibrium distance of zero. The (half-)stiffness of the *harmonic bond* $K_D = k_D/2$ and the Drude charge q_D are related to the atom polarizability α by

$$K_D = \frac{1}{2}\frac{q_D^2}{\alpha}$$

Ideally, the mass of the Drude particle should be small, and the stiffness of the harmonic bond should be large, so that the Drude particle remains close to the core. The values of Drude mass, Drude charge, and force constant can be chosen following different strategies, as in the following examples of polarizable force fields:

- *Lamoureux and Roux* suggest adopting a global half-stiffness, K_D = 500 kcal/(mol Ang ^{2}) - which corresponds to a force constant k_D = 4184 kJ/(mol Ang ^{2}) - for all types of core-Drude bond, a global mass m_D = 0.4 g/mol (or u) for all types of Drude particles, and to calculate the Drude charges for individual atom types from the atom polarizabilities using equation (1). This choice is followed in the polarizable CHARMM force field.
- Alternately *Schroeder and Steinhauser* suggest adopting a global charge q_D = -1.0e and a global mass m_D = 0.1 g/mol (or u) for all Drude particles, and to calculate the force constant for each type of core-Drude bond from equation (1). The timesteps used by these authors are between 0.5 and 2 fs, with the degrees of freedom of the Drude oscillators kept cold at 1 K.

- In both these force fields hydrogen atoms are treated as non-polarizable.

The motion of of the Drude particles can be calculated by minimizing the energy of the induced dipoles at each timestep, by an iterative, self-consistent procedure. The Drude particles can be massless and therefore do not contribute to the kinetic energy. However, the relaxed method is computational slow. An extended-lagrangian method can be used to calculate the positions of the Drude particles, but this requires them to have mass. It is important in this case to decouple the degrees of freedom associated with the Drude oscillators from those of the normal atoms. Thermalizing the Drude dipoles at temperatures comparable to the rest of the simulation leads to several problems (kinetic energy transfer, very short timestep, etc.), which can be remedied by the "cold Drude" technique (*Lamoureux and Roux*).

Two closely related models are used to represent polarization through "charges on a spring": the core-shell model and the Drude model. Although the basic idea is the same, the core-shell model is normally used for ionic/crystalline materials, whereas the Drude model is normally used for molecular systems and fluid states. In ionic crystals the symmetry around each ion and the distance between them are such that the core-shell model is sufficiently stable. But to be applicable to molecular/covalent systems the Drude model includes two important features:

1. The possibility to thermostat the additional degrees of freedom associated with the induced dipoles at very low temperature, in terms of the reduced coordinates of the Drude particles with respect to their cores. This makes the trajectory close to that of relaxed induced dipoles.
2. The Drude dipoles on covalently bonded atoms interact too strongly due to the short distances, so an atom may capture the Drude particle (shell) of a neighbor, or the induced dipoles within the same molecule may align too much. To avoid this, damping at short of the interactions between the point charges composing the induced dipole can be done by *Thole* functions.

Preparation of the data file

The data file is similar to a standard LAMMPS data file for *atom_style full*. The DPs and the *harmonic bonds* connecting them to their DC should appear in the data file as normal atoms and bonds.

You can use the *polarizer* tool (Python script distributed with the DRUDE package) to convert a non-polarizable data file (here *data.102494.lmp*) to a polarizable data file (*data-p.lmp*)

```
polarizer -q -f phenol.dff data.102494.lmp data-p.lmp
```

This will automatically insert the new atoms and bonds. The masses and charges of DCs and DPs are computed from *phenol.dff*, as well as the DC-DP bond constants. The file *phenol.dff* contains the polarizabilities of the atom types and the mass of the Drude particles, for instance:

```
# units: kJ/mol, A, deg
# kforce is in the form k/2 r_D^2
# type  m_D/u   q_D/e    k_D   alpha/A3  thole
OH      0.4    -1.0    4184.0  0.63     0.67
CA      0.4    -1.0    4184.0  1.36     2.51
CAI     0.4    -1.0    4184.0  1.09     2.51
```

The hydrogen atoms are absent from this file, so they will be treated as non-polarizable atoms. In the non-polarizable data file *data.102494.lmp*, atom names corresponding to the atom type numbers have to be specified as comments at the end of lines of the *Masses* section. You probably need to edit it to add these names. It should look like

```
Masses

1 12.011 # CAI
2 12.011 # CA
3 15.999 # OH
4 1.008  # HA
5 1.008  # HO
```

Basic input file

The atom style should be set to (or derive from) *full*, so that you can define atomic charges and molecular bonds, angles, dihedrals…

The *polarizer* tool also outputs certain lines related to the input script (the use of these lines will be explained below). In order for LAMMPS to recognize that you are using Drude oscillators, you should use the fix *drude*. The command is

```
fix DRUDE all drude C C C N N D D D
```

The N, C, D following the *drude* keyword have the following meaning: There is one tag for each atom type. This tag is C for DCs, D for DPs and N for non-polarizable atoms. Here the atom types 1 to 3 (C and O atoms) are DC, atom types 4 and 5 (H atoms) are non-polarizable and the atom types 6 to 8 are the newly created DPs.

By recognizing the fix *drude*, LAMMPS will find and store matching DC-DP pairs and will treat DP as equivalent to their DC in the *special bonds* relations. It may be necessary to extend the space for storing such special relations. In this case extra space should be reserved by using the *extra/special/per/atom* keyword of either the *read_data* or *create_box* command. With our phenol, there is 1 more special neighbor for which space is required. Otherwise LAMMPS crashes and gives the required value.

```
read_data data-p.lmp extra/special/per/atom 1
```

Let us assume we want to run a simple NVT simulation at 300 K. Note that Drude oscillators need to be thermalized at a low temperature in order to approximate a self-consistent field (SCF), therefore it is not possible to simulate an NVE ensemble with this package. Since dipoles are approximated by a charged DC-DP pair, the *pair_style* must include Coulomb interactions, for instance *lj/cut/coul/long* with *kspace_style pppm*. For example, with a cutoff of 10. and a precision 1.e-4:

```
pair_style lj/cut/coul/long 10.0
kspace_style pppm 1.0e-4
```

As compared to the non-polarizable input file, *pair_coeff* lines need to be added for the DPs. Since the DPs have no Lennard-Jones interactions, their ε is 0. so the only *pair_coeff* line that needs to be added is

```
pair_coeff * 6* 0.0 0.0 # All-DPs
```

Now for the thermalization, the simplest choice is to use the *fix langevin/drude*.

```
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
```

This applies a Langevin thermostat at temperature 300. to the centers of mass of the DC-DP pairs, with relaxation time 100 and with random seed 12345. This fix applies also a Langevin thermostat at temperature 1. to the relative motion of the DPs around their DCs, with relaxation time 20 and random seed 13977. Only the DCs and non-polarizable atoms need to be in this fix's group. LAMMPS will thermostat the DPs together with their DC. For this, ghost atoms need to know their velocities. Thus you need to add the following command:

```
comm_modify vel yes
```

In order to avoid that the center of mass of the whole system drifts due to the random forces of the Langevin thermostat on DCs, you can add the *zero yes* option at the end of the fix line.

If the fix *shake* is used to constrain the C-H bonds, it should be invoked after the fix *langevin/drude* for more accuracy.

```
fix SHAKE ATOMS shake 0.0001 20 0 t 4 5
```

Note

The group of the fix *shake* must not include the DPs. If the group *ATOMS* is defined by non-DPs atom types, you could use

Since the fix *langevin/drude* does not perform time integration (just modification of forces but no position/velocity updates), the fix *nve* should be used in conjunction.

```
fix NVE all nve
```

To avoid the flying ice cube artifact, where the atoms progressively freeze and the center of mass of the whole system drifts faster and faster, the *fix momentum* can be used. For instance:

```
fix MOMENTUM all momentum 100 linear 1 1 1
```

Finally, do not forget to update the atom type elements if you use them in a *dump_modify ... element ...* command, by adding the element type of the DPs. Here for instance

```
dump DUMP all custom 10 dump.lammpstrj id mol type element x y z ix iy iz
dump_modify DUMP element C C O H H D D D
```

The input file should now be ready for use!

You will notice that the global temperature *thermo_temp* computed by LAMMPS is not 300. K as wanted. This is because LAMMPS treats DPs as standard atoms in his default compute. If you want to output the temperatures of the DC-DP pair centers of mass and of the DPs relative to their DCs, you should use the *compute temp_drude*

```
compute TDRUDE all temp/drude
```

And then output the correct temperatures of the Drude oscillators using *thermo_style custom* with respectively *c_TDRUDE[1]* and *c_TDRUDE[2]*. These should be close to 300.0 and 1.0 on average.

```
thermo_style custom step temp c_TDRUDE[1] c_TDRUDE[2]
```

Thole screening

Dipolar interactions represented by point charges on springs may not be stable, for example if the atomic polarizability is too high for instance, a DP can escape from its DC and be captured by another DC, which makes the force and energy diverge and the simulation crash. Even without reaching this extreme case, the correlation between nearby dipoles on the same molecule may be exaggerated. Often, special bond relations prevent bonded neighboring atoms to see the charge of each other's DP, so that the problem does not always appear. It is possible to use screened dipole-dipole interactions by using the **pair_style thole**. This is implemented as a correction to the Coulomb pair_styles, which dampens at short distance the interactions between the charges representing the induced dipoles. It is to be used as *hybrid/overlay* with any standard *coul* pair style. In our example, we would use

```
pair_style hybrid/overlay lj/cut/coul/long 10.0 thole 2.6 10.0
```

This tells LAMMPS that we are using two pair_styles. The first one is as above (*lj/cut/coul/long 10.0*). The second one is a *thole* pair_style with default screening factor 2.6 (*Noskov*) and cutoff 10.0.

Since *hybrid/overlay* does not support mixing rules, the interaction coefficients of all the pairs of atom types with i < j should be explicitly defined. The output of the *polarizer* script can be used to complete the *pair_coeff* section of the input file. In our example, this will look like:

```
pair_coeff   1   1 lj/cut/coul/long   0.0700   3.550
pair_coeff   1   2 lj/cut/coul/long   0.0700   3.550
pair_coeff   1   3 lj/cut/coul/long   0.1091   3.310
pair_coeff   1   4 lj/cut/coul/long   0.0458   2.985
pair_coeff   2   2 lj/cut/coul/long   0.0700   3.550
pair_coeff   2   3 lj/cut/coul/long   0.1091   3.310
pair_coeff   2   4 lj/cut/coul/long   0.0458   2.985
pair_coeff   3   3 lj/cut/coul/long   0.1700   3.070
pair_coeff   3   4 lj/cut/coul/long   0.0714   2.745
pair_coeff   4   4 lj/cut/coul/long   0.0300   2.420
pair_coeff   *   5 lj/cut/coul/long   0.0000   0.000
pair_coeff   *  6* lj/cut/coul/long   0.0000   0.000
pair_coeff   1   1 thole   1.090   2.510
pair_coeff   1   2 thole   1.218   2.510
pair_coeff   1   3 thole   0.829   1.590
pair_coeff   1   6 thole   1.090   2.510
pair_coeff   1   7 thole   1.218   2.510
pair_coeff   1   8 thole   0.829   1.590
pair_coeff   2   2 thole   1.360   2.510
pair_coeff   2   3 thole   0.926   1.590
pair_coeff   2   6 thole   1.218   2.510
pair_coeff   2   7 thole   1.360   2.510
pair_coeff   2   8 thole   0.926   1.590
pair_coeff   3   3 thole   0.630   0.670
pair_coeff   3   6 thole   0.829   1.590
pair_coeff   3   7 thole   0.926   1.590
pair_coeff   3   8 thole   0.630   0.670
pair_coeff   6   6 thole   1.090   2.510
pair_coeff   6   7 thole   1.218   2.510
pair_coeff   6   8 thole   0.829   1.590
pair_coeff   7   7 thole   1.360   2.510
pair_coeff   7   8 thole   0.926   1.590
pair_coeff   8   8 thole   0.630   0.670
```

For the *thole* pair style the coefficients are

1. the atom polarizability in units of cubic length
2. the screening factor of the Thole function (optional, default value specified by the pair_style command)
3. the cutoff (optional, default value defined by the pair_style command)

The special neighbors have charge-charge and charge-dipole interactions screened by the *coul* factors of the *special_bonds* command (0.0, 0.0, and 0.5 in the example above). Without using the pair_style *thole*, dipole-dipole interactions are screened by the same factor. By using the pair_style *thole*, dipole-dipole interactions are screened by Thole's function, whatever their special relationship (except within each DC-DP pair of course). Consider for example 1-2 neighbors: using the pair_style *thole*, their dipoles will see each other (despite the *coul* factor being 0.) and the interactions between these dipoles will be damped by Thole's function.

Thermostats and barostats

Using a Nose-Hoover barostat with the *langevin/drude* thermostat is straightforward using fix *nph* instead of *nve*. For example:

```
fix NPH all nph iso 1. 1. 500
```

It is also possible to use a Nose-Hoover instead of a Langevin thermostat. This requires to use **fix drude/transform** just before and after the time integration fixes. The *fix drude/transform/direct* converts the atomic masses, positions, velocities and forces into a reduced representation, where the DCs transform into the centers of mass of the DC-DP pairs and the DPs transform into their relative position with respect to their DC. The *fix drude/transform/inverse* performs the reverse transformation. For a NVT simulation, with the DCs and atoms at 300 K and the DPs at 1 K relative to their DC one would use

```
fix DIRECT all drude/transform/direct
fix NVT1 ATOMS nvt temp 300. 300. 100
fix NVT2 DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

For our phenol example, the groups would be defined as

```
group ATOMS  type 1 2 3 4 5 # DCs and non-polarizable atoms
group CORES  type 1 2 3     # DCs
group DRUDES type 6 7 8     # DPs
```

Note that with the fixes *drude/transform*, it is not required to specify *comm_modify vel yes* because the fixes do it anyway (several times and for the forces also).

It is a bit more tricky to run a NPT simulation with Nose-Hoover barostat and thermostat. First, the volume should be integrated only once. So the fix for DCs and atoms should be *npt* while the fix for DPs should be *nvt* (or vice versa). Second, the *fix npt* computes a global pressure and thus a global temperature whatever the fix group. We do want the pressure to correspond to the whole system, but we want the temperature to correspond to the fix group only. We must then use the *fix_modify* command for this. In the end, the block of instructions for thermostatting and barostatting will look like

```
compute TATOMS ATOMS temp
fix DIRECT all drude/transform/direct
fix NPT ATOMS npt temp 300. 300. 100 iso 1. 1. 500
fix_modify NPT temp TATOMS press thermo_press
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

Another option for thermalizing the Drude model is to use the temperature-grouped Nose-Hoover (TGNH) thermostat proposed by *(Son)*. This is implemented as *fix tgnvt/drude* and *fix tgnpt/drude*. It separates the kinetic energy into three contributions: the molecular center of mass (COM) motion, the motion of atoms or atom-Drude pairs relative to molecular COMs, and the relative motion of atom-Drude pairs. An independent Nose-Hoover chain is applied to each type of motion. When TGNH is used, the temperatures of molecular, atomic and Drude motion can be printed out with *thermo_style command* command.

NVT simulation with TGNH thermostat

```
comm_modify vel yes
fix TGNVT all tgnvt/drude temp 300. 300. 100 1. 20
thermo_style custom f_TGNVT[1] f_TGNVT[2] f_TGNVT[3]
```

NPT simulation with TGNH thermostat

```
comm_modify vel yes
fix TGNPT all tgnpt/drude temp 300. 300. 100 1. 20 iso 1. 1. 500
thermo_style custom f_TGNPT[1] f_TGNPT[2] f_TGNPT[3]
```

Rigid bodies

You may want to simulate molecules as rigid bodies (but polarizable). Common cases are water models such as *SWM4-NDP*, which is a kind of polarizable TIP4P water. The rigid bodies and the DPs should be integrated separately, even with the Langevin thermostat. Let us review the different thermostats and ensemble combinations.

NVT ensemble using Langevin thermostat:

```
comm_modify vel yes
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
fix RIGID ATOMS rigid/nve/small molecule
fix NVE DRUDES nve
```

NVT ensemble using Nose-Hoover thermostat:

```
fix DIRECT all drude/transform/direct
fix RIGID ATOMS rigid/nvt/small molecule temp 300. 300. 100
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

NPT ensemble with Langevin thermostat:

```
comm_modify vel yes
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
fix RIGID ATOMS rigid/nph/small molecule iso 1. 1. 500
fix NVE DRUDES nve
```

NPT ensemble using Nose-Hoover thermostat:

```
compute TATOM ATOMS temp
fix DIRECT all drude/transform/direct
fix RIGID ATOMS rigid/npt/small molecule temp 300. 300. 100 iso 1. 1. 500
fix_modify RIGID temp TATOM press thermo_press
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

(Lamoureux and Roux) Lamoureux and Roux, J Chem Phys, 119, 3025-3039 (2003)

(Schroeder) Schroeder and Steinhauser, J Chem Phys, 133, 154511 (2010).

(Thole) Chem Phys, 59, 341 (1981).

(Noskov) Noskov, Lamoureux and Roux, J Phys Chem B, 109, 6705 (2005).

(SWM4-NDP) Lamoureux, Harder, Vorobyov, Roux, MacKerell, Chem Phys Let, 418, 245-249 (2006)

(Son) Son, McDaniel, Cui and Yethiraj, J Phys Chem Lett, 10, 7523 (2019).

8.5.9 Peridynamics with LAMMPS

This Howto is based on the Sandia report 2010-5549 by Michael L. Parks, Pablo Seleson, Steven J. Plimpton, Richard B. Lehoucq, and Stewart A. Silling.

Overview

Peridynamics is a nonlocal extension of classical continuum mechanics. The discrete peridynamic model has the same computational structure as a molecular dynamics model. This Howto provides a brief overview of the peridynamic model of a continuum, then discusses how the peridynamic model is discretized within LAMMPS as described in the original article *(Parks)*. An example problem with comments is also included.

Quick Start

The peridynamics styles are included in the optional *PERI package*. If your LAMMPS executable does not already include the PERI package, you can see the *build instructions for packages* for how to enable the package when compiling a custom version of LAMMPS from source.

Here is a minimal example for setting up a peridynamics simulation.

```
units          si
boundary       s s s
lattice        sc 0.0005
atom_style     peri
atom_modify    map array
neighbor       0.0010 bin
region         target cylinder y 0.0 0.0 0.0050 -0.0050 0.0 units box
create_box     1 target
create_atoms   1 region target

pair_style     peri/pmb
pair_coeff     * * 1.6863e22 0.0015001 0.0005 0.25
set            group all density 2200
set            group all volume 1.25e-10
velocity       all set 0.0 0.0 0.0 sum no units box
fix            1 all nve
compute        1 all damage/atom
timestep       1.0e-7
```

Some notes on this input example:

- peridynamics simulations typically use SI *units*
- particles must be created on a *simple cubic lattice*
- using the *atom style peri* is required
- an *atom map* is required for indexing particles
- The *skin distance* used when computing neighbor lists should be defined appropriately for your choice of simulation parameters. The *skin* should be set to a value such that the peridynamic horizon plus the skin distance is larger than the maximum possible distance between two bonded particles (before their bond breaks). Here it is set to 0.001 meters.
- a *peridynamics pair style* is required. Available choices are currently: *peri/eps*, *peri/lps*, *peri/pmb*, and *peri/ves*. The model parameters are set with a *pair_coeff* command.

- the mass density and volume fraction for each particle must be defined. This is done with the two *set* commands for *density* and *volume*. For a simple cubic lattice, the volume of a particle should be equal to the cube of the lattice constant, here $V_i = \Delta x^3$.
- with the *velocity* command all particles are initially at rest
- a plain *velocity-Verlet time integrator* is used, which is algebraically equivalent to a centered difference in time, but numerically more stable
- you can compute the damage at the location of each particle with *compute damage/atom*
- finally, the timestep is set to 0.1 microseconds with the *timestep* command.

Peridynamic Model of a Continuum

The following is not a complete overview of peridynamics, but a discussion of only those details specific to the model we have implemented within LAMMPS. For more on the peridynamic theory, the reader is referred to *(Silling 2007)*. To begin, we define the notation we will use.

Basic Notation

Within the peridynamic literature, the following notational conventions are generally used. The position of a given point in the reference configuration is $\mathbf{x}$. Let $\mathbf{u}(\mathbf{x}, t)$ and $\mathbf{y}(\mathbf{x}, t)$ denote the displacement and position, respectively, of the point $\mathbf{x}$ at time t. Define the relative position and displacement vectors of two bonded points $\mathbf{x}$ and $\mathbf{x}'$ as $\xi = \mathbf{x}' - \mathbf{x}$ and $\eta = \mathbf{u}(\mathbf{x}', t) - \mathbf{u}(\mathbf{x}, t)$, respectively. We note here that η is time-dependent, and that ξ is not. It follows that the relative position of the two bonded points in the current configuration can be written as $\xi + \eta = \mathbf{y}(\mathbf{x}', t) - \mathbf{y}(\mathbf{x}, t)$.

Peridynamic models are frequently written using *states*, which we briefly describe here. For the purposes of our discussion, all states are operators that act on vectors in $\mathbb{R}^3$. For a more complete discussion of states, see *(Silling 2007)*. A *vector state* is an operator whose image is a vector, and may be viewed as a generalization of a second-rank tensor. Similarly, a *scalar state* is an operator whose image is a scalar. Of particular interest is the vector force state $\underline{\mathbf{T}}[\mathbf{x}, t]\langle \mathbf{x}' - \mathbf{x} \rangle$, which is a mapping, having units of force per volume squared, of the vector $\mathbf{x}' - \mathbf{x}$ to the force vector state field. The vector state operator $\underline{\mathbf{T}}$ may itself be a function of $\mathbf{x}$ and t. The constitutive model is completely contained within $\underline{\mathbf{T}}$.

In the peridynamic theory, the deformation at a point depends collectively on all points interacting with that point. Using the notation of *(Silling 2007)*, we write the peridynamic equation of motion as

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x}, t) = \int_{\mathcal{H}_\mathbf{x}} \left\{ \underline{\mathbf{T}}[\mathbf{x}, t]\langle \mathbf{x}' - \mathbf{x} \rangle - \underline{\mathbf{T}}\left[\mathbf{x}', t\right]\langle \mathbf{x} - \mathbf{x}' \rangle \right\} dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x}, t), \qquad (1)$$

where ρ represents the mass density, $\underline{\mathbf{T}}$ the force vector state, and $\mathbf{b}$ an external body force density. A point $\mathbf{x}$ interacts with all the points $\mathbf{x}'$ within the neighborhood $\mathcal{H}_\mathbf{x}$, assumed to be a spherical region of radius $\delta > 0$ centered at $\mathbf{x}$. δ is called the *horizon*, and is analogous to the cutoff radius used in molecular dynamics. Conditions on $\underline{\mathbf{T}}$ for which *(1)* satisfies the balance of linear and angular momentum are given in *(Silling 2007)*.

We consider only force vector states that can be written as

$$\underline{\mathbf{T}} = \underline{t}\,\underline{\mathbf{M}},$$

with $\underline{t}$ a *scalar force state* and $\underline{\mathbf{M}}$ the *deformed direction vector state*, defined by

$$\underline{\mathbf{M}}\langle \xi \rangle = \begin{cases} \frac{\xi+\eta}{\|\xi+\eta\|} & \|\xi + \eta\| \neq 0 \\ 0 & \text{otherwise} \end{cases}. \qquad (2)$$

Such force states correspond to so-called *ordinary* materials *(Silling 2007)*. These are the materials for which the force between any two interacting points $\mathbf{x}$ and $\mathbf{x}'$ acts along the line between the points.

Linear Peridynamic Solid (LPS) Model

We summarize the linear peridynamic solid (LPS) material model. For more on this model, the reader is referred to *(Silling 2007)*. This model is a nonlocal analogue to a classical linear elastic isotropic material. The elastic properties of a a classical linear elastic isotropic material are determined by (for example) the bulk and shear moduli. For the LPS model, the elastic properties are analogously determined by the bulk and shear moduli, along with the horizon δ.

The LPS model has a force scalar state

$$\underline{t} = \frac{3K\theta}{m}\underline{\omega}\,\underline{x} + \alpha\underline{\omega}\,\underline{e}^{\mathrm{d}}, \qquad (3)$$

with K the bulk modulus and α related to the shear modulus G as

$$\alpha = \frac{15G}{m}.$$

The remaining components of the model are described as follows. Define the reference position scalar state $\underline{x}$ so that $\underline{x}\langle\xi\rangle = \|\xi\|$. Then, the weighted volume m is defined as

$$m[\mathbf{x}] = \int_{\mathcal{H}_{\mathbf{x}}} \underline{\omega}\langle\xi\rangle\,\underline{x}\langle\xi\rangle\,\underline{x}\langle\xi\rangle\,dV_{\xi}. \qquad (4)$$

Let

$$\underline{e}[\mathbf{x},t]\langle\xi\rangle = \|\xi+\eta\| - \|\xi\|$$

be the extension scalar state, and

$$\theta[\mathbf{x},t] = \frac{3}{m[\mathbf{x}]}\int_{\mathcal{H}_{\mathbf{x}}} \underline{\omega}\langle\xi\rangle\,\underline{x}\langle\xi\rangle\,\underline{e}[\mathbf{x},t]\langle\xi\rangle\,dV_{\xi}$$

be the dilatation. The isotropic and deviatoric parts of the extension scalar state are defined, respectively, as

$$\underline{e}^{\mathrm{i}} = \frac{\theta\underline{x}}{3}, \qquad \underline{e}^{\mathrm{d}} = \underline{e} - \underline{e}^{\mathrm{i}},$$

where the arguments of the state functions and the vectors on which they operate are omitted for simplicity. We note that the LPS model is linear in the dilatation θ, and in the deviatoric part of the extension $\underline{e}^{\mathrm{d}}$.

Note

The weighted volume m is time-independent, and does not change as bonds break. It is computed with respect to the bond family defined at the reference (initial) configuration.

The non-negative scalar state $\underline{\omega}$ is an *influence function (Silling 2007)*. For more on influence functions, see *(Seleson 2010)*. If an influence function $\underline{\omega}$ depends only upon the scalar $\|\xi\|$, (i.e., $\underline{\omega}\langle\xi\rangle = \underline{\omega}\langle\|\xi\|\rangle$), then $\underline{\omega}$ is a spherical influence function. For a spherical influence function, the LPS model is isotropic *(Silling 2007)*.

Note

In the LAMMPS implementation of the LPS model, the influence function $\underline{\omega}\langle\|\xi\|\rangle = 1/\|\xi\|$ is used. However, the user can define their own influence function by altering the method "influence_function" in the file pair_peri_lps.cpp. The LAMMPS peridynamics code permits both spherical and non-spherical influence functions (e.g., isotropic and non-isotropic materials).

Prototype Microelastic Brittle (PMB) Model

We summarize the prototype microelastic brittle (PMB) material model. For more on this model, the reader is referred to *(Silling 2000)* and *(Silling 2005)*. This model is a special case of the LPS model; see *(Seleson 2010)* for the derivation. The elastic properties of the PMB model are determined by the bulk modulus K and the horizon δ.

The PMB model is expressed using the scalar force state field

$$\underline{t}\,[\mathbf{x},t]\,\langle\xi\rangle = \frac{1}{2} f\left(\eta,\xi\right), \qquad (5)$$

with f a scalar-valued function. We assume that f takes the form

$$f = cs,$$

where

$$c = \frac{18K}{\pi\delta^4}, \qquad (6)$$

with K the bulk modulus and δ the horizon, and s the bond stretch, defined as

$$s(t,\eta,\xi) = \frac{\|\eta+\xi\| - \|\xi\|}{\|\xi\|}.$$

Bond stretch is a unitless quantity, and identical to a one-dimensional definition of strain. As such, we see that a bond at its equilibrium length has stretch $s = 0$, and a bond at twice its equilibrium length has stretch $s = 1$. The constant c given above is appropriate for 3D models only. For more on the origins of the constant c, see *(Silling 2005)*. For the derivation of c for 1D and 2D models, see *(Emmrich)*.

Given *(5)*, *(1)* reduces to

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{\mathcal{H}_\mathbf{x}} \mathbf{f}\left(\eta,\xi\right) dV_\xi + \mathbf{b}(\mathbf{x},t), \qquad (7)$$

with

$$\mathbf{f}\left(\eta,\xi\right) = f\left(\eta,\xi\right)\frac{\xi+\eta}{\|\xi+\eta\|}.$$

Unlike the LPS model, the PMB model has a Poisson ratio of $\nu = 1/4$ in 3D, and $\nu = 1/3$ in 2D. This is reflected in the input for the PMB model, which requires only the bulk modulus of the material, whereas the LPS model requires both the bulk and shear moduli.

Damage

Bonds are made to break when they are stretched beyond a given limit. Once a bond fails, it is failed forever *(Silling)*. Further, new bonds are never created during the course of a simulation. We discuss only one criterion for bond breaking, called the *critical stretch* criterion.

Define μ to be the history-dependent scalar boolean function

$$\mu(t,\eta,\xi) = \left\{ \begin{array}{ll} 1 & \text{if } s(t',\eta,\xi) < \min\left(s_0(t',\eta,\xi), s_0(t',\eta',\xi')\right) \text{ for all } 0 \le t' \le t \\ 0 & \text{otherwise} \end{array} \right\}. \qquad (8)$$

where $\eta' = \mathbf{u}(\mathbf{x}'', t) - \mathbf{u}(\mathbf{x}', t)$ and $\xi' = \mathbf{x}'' - \mathbf{x}'$. Here, $s_0(t, \eta, \xi)$ is a critical stretch defined as

$$s_0(t, \eta, \xi) = s_{00} - \alpha s_{\min}(t, \eta, \xi), \qquad s_{\min}(t) = \min_{\xi} s(t, \eta, \xi), \tag{9}$$

where s_{00} and α are material-dependent constants. The history function μ breaks bonds when the stretch s exceeds the critical stretch s_0.

Although $s_0(t, \eta, \xi)$ is expressed as a property of a particle, bond breaking must be a symmetric operation for all particle pairs sharing a bond. That is, particles $\mathbf{x}$ and $\mathbf{x}'$ must utilize the same test when deciding to break their common bond. This can be done by any method that treats the particles symmetrically. In the definition of μ above, we have chosen to take the minimum of the two s_0 values for particles $\mathbf{x}$ and $\mathbf{x}'$ when determining if the $\mathbf{x}$–$\mathbf{x}'$ bond should be broken.

Following *(Silling)*, we can define the damage at a point $\mathbf{x}$ as

$$\varphi(\mathbf{x}, t) = 1 - \frac{\int_{\mathcal{H}_\mathbf{x}} \mu(t, \eta, \xi) dV_{\mathbf{x}'}}{\int_{\mathcal{H}_\mathbf{x}} dV_{\mathbf{x}'}}. \tag{10}$$

Discrete Peridynamic Model and LAMMPS Implementation

In LAMMPS, instead of *(1)*, we model this equation of motion:

$$\rho(\mathbf{x})\ddot{\mathbf{y}}(\mathbf{x}, t) = \int_{\mathcal{H}_\mathbf{x}} \left\{ \underline{\mathbf{T}}\left[\mathbf{x}, t\right] \langle \mathbf{x}' - \mathbf{x} \rangle - \underline{\mathbf{T}}\left[\mathbf{x}', t\right] \langle \mathbf{x} - \mathbf{x}' \rangle \right\} dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x}, t),$$

where we explicitly track and store at each timestep the positions and not the displacements of the particles. We observe that $\ddot{\mathbf{y}}(\mathbf{x}, t) = \ddot{\mathbf{x}} + \ddot{\mathbf{u}}(\mathbf{x}, t) = \ddot{\mathbf{u}}(\mathbf{x}, t)$, so that this is equivalent to *(1)*.

Spatial Discretization

The region defining a peridynamic material is discretized into particles forming a simple cubic lattice with lattice constant Δx, where each particle i is associated with some volume fraction V_i. For any particle i, let $\mathcal{F}_i$ denote the family of particles for which particle i shares a bond in the reference configuration. That is,

$$\mathcal{F}_i = \{p \mid \|\mathbf{x}_p - \mathbf{x}_i\| \leq \delta\}. \tag{11}$$

The discretized equation of motion replaces *(1)* with

$$\rho\ddot{\mathbf{y}}_i^n = \sum_{p \in \mathcal{F}_i} \left\{ \underline{\mathbf{T}}\left[\mathbf{x}_i, t\right] \left\langle \mathbf{x}_p' - \mathbf{x}_i \right\rangle - \underline{\mathbf{T}}\left[\mathbf{x}_p, t\right] \left\langle \mathbf{x}_i - \mathbf{x}_p \right\rangle \right\} V_p + \mathbf{b}_i^n, \tag{12}$$

where n is the timestep number and subscripts denote the particle number.

Short-Range Forces

In the model discussed so far, particles interact only through their bond forces. A particle with no bonds becomes a free non-interacting particle. To account for contact forces, short-range forces are introduced *(Silling 2007)*. We add to the force in *(12)* the following force

$$\mathbf{f}_S(\mathbf{y}_p, \mathbf{y}_i) = \min\left\{0, \frac{c_S}{\delta}(\left\|\mathbf{y}_p - \mathbf{y}_i\right\| - d_{pi})\right\} \frac{\mathbf{y}_p - \mathbf{y}_i}{\left\|\mathbf{y}_p - \mathbf{y}_i\right\|}, \tag{13}$$

where d_{pi} is the short-range interaction distance between particles p and i, and c_S is a multiple of the constant c from *(6)*. Note that the short-range force is always repulsive, never attractive. In practice, we choose

$$c_S = 15\frac{18K}{\pi\delta^4}. \qquad (14)$$

For the short-range interaction distance, we choose *(Silling 2007)*

$$d_{pi} = \min\left\{0.9\left\|\mathbf{x}_p - \mathbf{x}_i\right\|, 1.35(r_p + r_i)\right\}, \qquad (15)$$

where r_i is called the *node radius* of particle i. Given a discrete lattice, we choose r_i to be half the lattice constant.

Note

For a simple cubic lattice, $\Delta x = \Delta y = \Delta z$.

Given this definition of d_{pi}, contact forces appear only when particles are under compression.

When accounting for short-range forces, it is convenient to define the short-range family of particles

$$\mathcal{F}_i^S = \{p \mid \left\|\mathbf{y}_p - \mathbf{y}_i\right\| \leq d_{pi}\}.$$

Modification to the Particle Volume

The right-hand side of *(12)* may be thought of as a midpoint quadrature of *(1)*. To slightly improve the accuracy of this quadrature, we discuss a modification to the particle volume used in *(12)*. In a situation where two particles share a bond with $\|\mathbf{x}_p - \mathbf{x}_i\| = \delta$, for example, we suppose that only approximately half the volume of each particle is "seen" by the other *(Silling 2007)*. When computing the force of each particle on the other we use $V_p/2$ rather than V_p in *(12)*. As such, we introduce a nodal volume scaling function for all bonded particles where $\delta - r_i \leq \|\mathbf{x}_p - \mathbf{x}_i\| \leq \delta$ (see the Figure below).

We choose to use a linear unitless nodal volume scaling function

$$\nu(\mathbf{x}_p - \mathbf{x}_i) = \left\{ \begin{array}{cc} -\frac{1}{2r_i}\|\mathbf{x}_p - \mathbf{x}_i\| + \left(\frac{\delta}{2r_i} + \frac{1}{2}\right) & \text{if } \delta - r_i \leq \|\mathbf{x}_p - \mathbf{x}_i\| \leq \delta \\ 1 & \text{if } \|\mathbf{x}_p - \mathbf{x}_i\| \leq \delta - r_i \\ 0 & \text{otherwise} \end{array} \right\}$$

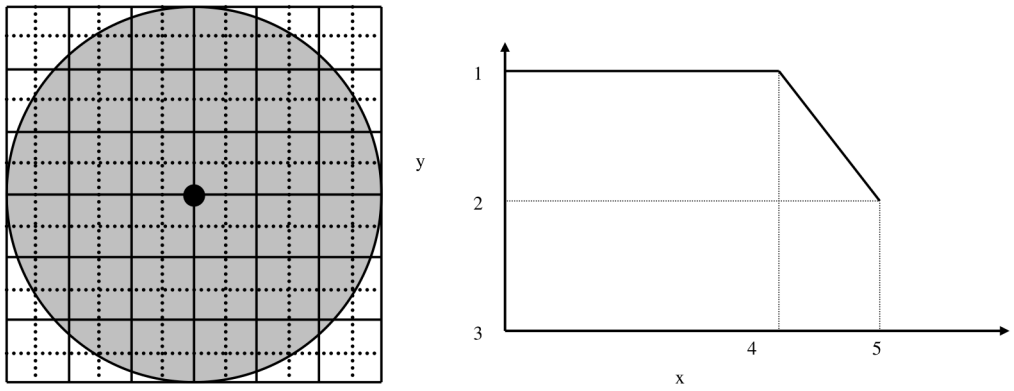
If $\|\mathbf{x}_p - \mathbf{x}_i\| = \delta$, $\nu = 0.5$, and if $\|\mathbf{x}_p - \mathbf{x}_i\| = \delta - r_i$, $\nu = 1.0$, for example.

Temporal Discretization

When discretizing time in LAMMPS, we use a velocity-Verlet scheme, where both the position and velocity of the particle are stored explicitly. The velocity-Verlet scheme is generally expressed in three steps. In *Algorithm 1*, ρ_i denotes the mass density of a particle and $\tilde{\mathbf{f}}_i^n$ denotes the the net force density on particle i at timestep n. The LAMMPS command *fix nve* performs a velocity-Verlet integration.

Algorithm 1: Velocity Verlet

1: $\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \frac{\Delta t}{2\rho_i}\tilde{\mathbf{f}}_i^n$



(a) Two-dimensional diagram showing particle on mesh (solid lines) with neighborhood $\mathcal{H}_{\mathbf{x}}$ as grey circular region. Dual mesh (dotted lines) shows boundaries of each particle.

(b) Plot of $\nu(\mathbf{x}_p - \mathbf{x}_i)$ vs. $\|\mathbf{x}_p - \mathbf{x}_i\|$.

Fig. 1: Diagram showing horizon of a particular particle, demonstrating that the volume associated with particles near the boundary of the horizon is not completely contained within the horizon.

2: $\mathbf{y}_i^{n+1} = \mathbf{y}_i^n + \Delta t \mathbf{v}_i^{n+1/2}$
3: $\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{\Delta t}{2\rho_i} \widetilde{\mathbf{f}}_i^{n+1}$

Breaking Bonds

During the course of simulation, it may be necessary to break bonds, as described in the *Damage section*. Bonds are recorded as broken in a simulation by removing them from the bond family $\mathcal{F}_i$ (see *(11)*).

A naive implementation would have us first loop over all bonds and compute s_{min} in *(9)*, then loop over all bonds again and break bonds with a stretch $s > s0$ as in *(8)*, and finally loop over all particles and compute forces for the next step of *Algorithm 1*. For reasons of computational efficiency, we will utilize the values of s_0 from the *previous* timestep when deciding to break a bond.

Note

For the first timestep, s_0 is initialized to ∞ for all nodes. This means that no bonds may be broken until the second timestep. As such, it is recommended that the first few timesteps of the peridynamic simulation not involve any actions that might result in the breaking of bonds. As a practical example, the projectile in the *commented example below* is placed such that it does not impact the target brittle plate until several timesteps into the simulation.

LPS Pseudocode

A sketch of the LPS model implementation in the PERI package appears in *Algorithm 2*. This algorithm makes use of the routines in *Algorithm 3* and *Algorithm 4*.

Algorithm 2: LPS Peridynamic Model Pseudocode

Fix s_{00}, α, horizon δ, bulk modulus K, shear modulus G, timestep Δt, and generate initial lattice of particles with lattice constant Δx. Let there be N particles. Define constant c_S for repulsive short-range forces.
Initialize bonds between all particles $i \neq j$ where $\|\mathbf{x}_j - \mathbf{x}_i\| \leq \delta$
Initialize weighted volume m for all particles using *Algorithm 3*
Initialize $s_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
while not done **do**
  Perform step 1 of *Algorithm 1*, updating velocities of all particles
  Perform step 2 of *Algorithm 1*, updating positions of all particles
  $\tilde{s}_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
  for i = 1 to N **do**
    {Compute short-range forces}
    for all particles $j \in \mathcal{F}_i^S$ (the short-range family of nodes for particle i) **do**
      $r = \|\mathbf{y}_j - \mathbf{y}_i\|$
      $dr = \min\{0, r - d\}$ {*Short-range forces are only repulsive, never attractive*}
      $k = \frac{c_S}{\delta} V_k dr$ {c_S *defined in :ref: `(14) <pericS> `*}
      $\mathbf{f} = \mathbf{f} + k \frac{\mathbf{y}_j - \mathbf{y}_i}{\|\mathbf{y}_j - \mathbf{y}_i\|}$
    end for
  end for
  Compute the dilatation for each particle using *Algorithm 4*
  for i = 1 to N **do**
    {Compute bond forces}
    for all particles j sharing an unbroken bond with particle i **do**
      $e = \|\mathbf{y}_j - \mathbf{y}_i\| - \|\mathbf{x}_j - \mathbf{x}_i\|$
      $\omega_+ = \underline{\omega}\langle \mathbf{x}_j - \mathbf{x}_i \rangle$ {*Influence function evaluation*}
      $\omega_- = \underline{\omega}\langle \mathbf{x}_i - \mathbf{x}_j \rangle$ {*Influence function evaluation*}
      $\hat{f} = \left[(3K - 5G) \left(\frac{\theta(i)}{m(i)} \omega_+ + \frac{\theta(j)}{m(j)} \omega_- \right) \|\mathbf{x}_j - \mathbf{x}_i\| + 15G \left(\frac{\omega_+}{m(i)} + \frac{\omega_-}{m(j)} \right) e \right] \nu(\mathbf{x}_j - \mathbf{x}_i) V_j$
      $\mathbf{f} = \mathbf{f} + \hat{f} \frac{\mathbf{y}_j - \mathbf{y}_i}{\|\mathbf{y}_j - \mathbf{y}_i\|}$
      if $(dr / \|\mathbf{x}_j - \mathbf{x}_i\|) > \min(s_0(i), s_0(j))$ **then**
        Break i's bond with j {j *'s bond with i will be broken when this loop iterates on* j}
      end if
      $\tilde{s}_0(i) = \min(\tilde{s}_0(i), s_{00} - \alpha(dr / \|\mathbf{x}_j - \mathbf{x}_i\|))$
    end for
  end for
  $s_0 = \tilde{s}_0$ {*Store for use in next timestep*}
  Perform step 3 of *Algorithm 1*, updating velocities of all particles

end while

Algorithm 3: Computation of Weighted Volume m

for $i = 1$ to N **do**
  $m(i) = 0.0$
  for all particles j sharing a bond with particle i **do**
    $m(i) = m(i) + \underline{\omega} \langle \mathbf{x}_j - \mathbf{x}_i \rangle \|\mathbf{x}_j - \mathbf{x}_i\|^2 \nu(\mathbf{x}_j - \mathbf{x}_i) V_j$
  end for
end for

Algorithm 4: Computation of Dilatation θ

for $i = 1$ to N **do**
  $\theta(i) = 0.0$
  for all particles j sharing an unbroken bond with particle i **do**
    $e = \|\mathbf{y}_i - \mathbf{y}_j\| - \|\mathbf{x}_i - \mathbf{x}_j\|$
    $\theta(i) = \theta(i) + \underline{\omega} \langle \mathbf{x}_j - \mathbf{x}_i \rangle \|\mathbf{x}_j - \mathbf{x}_i\| e \nu(\mathbf{x}_j - \mathbf{x}_i) V_j$
  end for
  $\theta(i) = \frac{3}{m(i)} \theta(i)$
end for

PMB Pseudocode

A sketch of the PMB model implementation in the PERI package appears in *Algorithm 5*.

Algorithm 5: PMB Peridynamic Model Pseudocode

Fix s_{00}, α, horizon δ, spring constant c, timestep Δt, and generate initial lattice of particles with lattice constant Δx. Let there be N particles.
Initialize bonds between all particles $i \neq j$ where $\|\mathbf{x}_j - \mathbf{x}_i\| \leq \delta$
Initialize $s_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
while not done **do**
  Perform step 1 of *Algorithm 1*, updating velocities of all particles
  Perform step 2 of *Algorithm 1*, updating positions of all particles
  $\tilde{s}_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
  for $i = 1$ to N **do**
    {Compute short-range forces}
    for all particles $j \in \mathcal{F}_i^S$ (the short-range family of nodes for particle i) **do**
      $r = \|\mathbf{y}_j - \mathbf{y}_i\|$

$dr = \min\{0, r - d\}$ {*Short-range forces are only repulsive, never attractive*}
$k = \frac{c_S}{\delta} V_k dr$ {*c_S defined in :ref:`(14) <pericS>`*}
$\mathbf{f} = \mathbf{f} + k\frac{\mathbf{y}_j - \mathbf{y}_i}{\|\mathbf{y}_j - \mathbf{y}_i\|}$
end for
end for
for $i = 1$ to N **do**
{Compute bond forces}
for all particles j sharing an unbroken bond with particle i **do**
$r = \|\mathbf{y}_j - \mathbf{y}_i\|$
$dr = r - \|\mathbf{x}_j - \mathbf{x}_i\|$
$k = \frac{c}{\|\mathbf{x}_i - \mathbf{x}_j\|} \nu(\mathbf{x}_i - \mathbf{x}_j) V_j dr$ {*c defined in :ref:`(6) <peric>`*}
$\mathbf{f} = \mathbf{f} + k\frac{\mathbf{y}_j - \mathbf{y}_i}{\|\mathbf{y}_j - \mathbf{y}_i\|}$
if $(dr / \|\mathbf{x}_j - \mathbf{x}_i\|) > \min(s_0(i), s_0(j))$ **then**
Break i's bond with j {*j's bond with i will be broken when this loop iterates on j*}
end if
$\tilde{s}_0(i) = \min(\tilde{s}_0(i), s_{00} - \alpha(dr / \|\mathbf{x}_j - \mathbf{x}_i\|))$
end for
end for
$s_0 = \tilde{s}_0$ {*Store for use in next timestep*}
Perform step 3 of *Algorithm 1*, updating velocities of all particles
end while

Damage

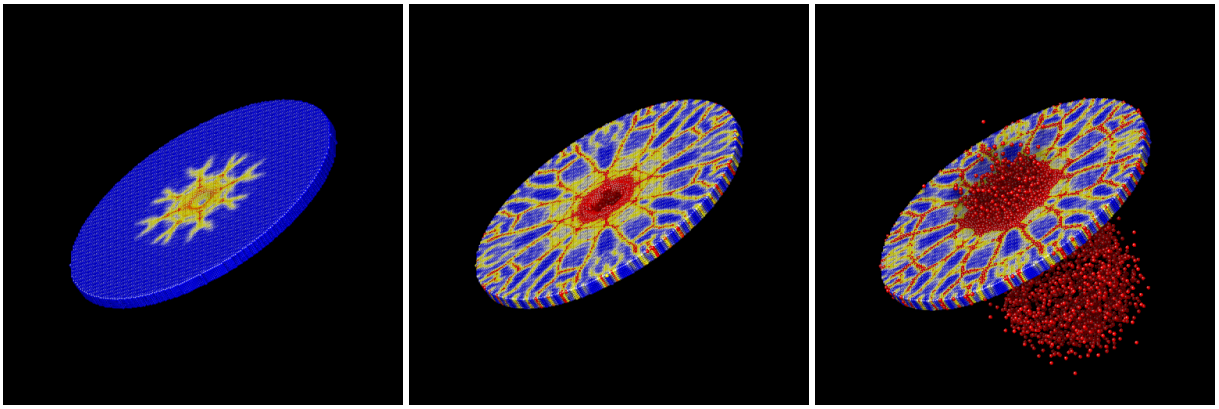
The damage associated with every particle (see *(10)*) can optionally be computed and output with a LAMMPS data dump. To do this, your input script must contain the command *compute damage/atom* This enables a LAMMPS per-atom compute to calculate the damage associated with each particle every time a LAMMPS *data dump* frame is written.

Visualizing Simulation Results

There are multiple ways to visualize the simulation results. Typically, you want to display the particles and color code them by the value computed with the *compute damage/atom* command.

This can be done, for example, by using the built-in visualizer of the *dump image or dump movie* command to create snapshot images or a movie. Below are example command lines for using dump image with the *example listed below* and a set of images created for steps 300, 600, and 2000 this way.

```
dump            D2 all image 100 dump.peri.*.png c_C1 type box no 0.0 view 30 60 zoom 1.5 up 0 0 -1 ␣
↪ssao yes 4539 0.6
dump_modify     D2 pad 5 adiam * 0.001 amap 0.0 1.0 ca 0.1 3 min blue 0.5 yellow max red
```



For interactive visualization, the Ovito is very convenient to use. Below are steps to create a visualization of the *same example from below* now using the generated trajectory in the dump.peri file.

- Launch Ovito
- File -> Load File -> dump.peri
- Select "-> Particle types" and under "Appearance" set "Display radius:" to 0.0005
- From the "Add modification:" drop down list select "Color coding"
- Under "Color coding" select from the "Color gradient" drop down list "Jet"
- Also under "Color coding" set "Start value:" to 0 and "End value:" to 1
- You can improve the image quality by adding the "Ambient occlusion" modification

Pitfalls

Parallel Scalability

LAMMPS operates in parallel in a *spatial-decomposition mode*, where each processor owns a spatial subdomain of the overall simulation domain and communicates with its neighboring processors via distributed-memory message passing (MPI) to acquire ghost atom information to allow forces on the atoms it owns to be computed. LAMMPS also uses Verlet neighbor lists which are recomputed every few timesteps as particles move. On these timesteps, particles also migrate to new processors as needed. LAMMPS decomposes the overall simulation domain so that spatial subdomains of nearly equal volume are assigned to each processor. When each subdomain contains nearly the same number of particles, this results in a reasonable load balance among all processors. As is more typical with some peridynamic simulations, some subdomains may contain many particles while other subdomains contain few particles, resulting in a load imbalance that impacts parallel scalability.

Setting the "skin" distance

The *neighbor* command with LAMMPS is used to set the so-called "skin" distance used when building neighbor lists. All atom pairs within a cutoff distance equal to the horizon δ plus the skin distance are stored in the list. Unexpected crashes in LAMMPS may be due to too small a skin distance. The skin should be set to a value such that δ plus the skin distance is larger than the maximum possible distance between two bonded particles. For example, if s_{00} is increased, the skin distance may also need to be increased.

"Lost" particles

All particles are contained within the "simulation box" of LAMMPS. The boundaries of this box may change with time, or not, depending on how the LAMMPS *boundary* command has been set. If a particle drifts outside the simulation box during the course of a simulation, it is called *lost*.

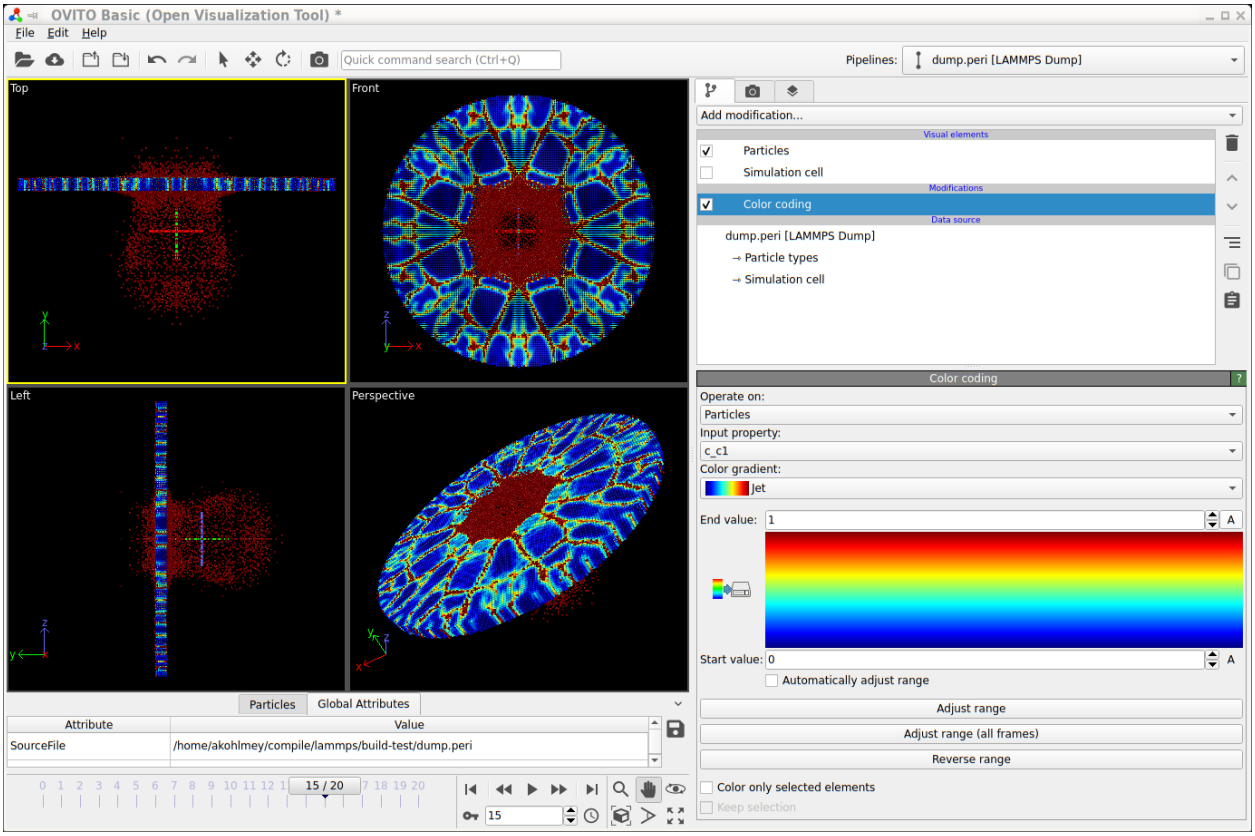


Fig. 2: Screenshot of visualizing a trajectory with Ovito

As an option of the *themo_modify* command of LAMMPS, the lost keyword determines whether LAMMPS checks for lost atoms each time it computes thermodynamics and what it does if atoms are lost. If the value is *ignore*, LAMMPS does not check for lost atoms. If the value is *error* or *warn*, LAMMPS checks and either issues an error or warning. The code will exit with an error and continue with a warning. This can be a useful debugging option. The default behavior of LAMMPS is to exit with an error if a particle is lost.

The peridynamic module within LAMMPS does not check for lost atoms. If a particle with unbroken bonds is lost, those bonds are marked as broken by the remaining particles.

Defining the peridynamic horizon δ

In the *pair_coeff* command, the user must specify the horizon δ. This argument determines which particles are bonded when the simulation is initialized. It is recommended that δ be set to a small fraction of a lattice constant larger than desired.

For example, if the lattice constant is 0.0005 and you wish to set the horizon to three times the lattice constant, then set δ to be 0.0015001, a value slightly larger than three times the lattice constant. This guarantees that particles three lattice constants away from each other are still bonded. If δ is set to 0.0015, for example, floating point error may result in some pairs of particles three lattice constants apart not being bonded.

Breaking bonds too early

For technical reasons, the bonds in the simulation are not created until the end of the first timestep of the simulation. Therefore, one should not attempt to break bonds until at least the second step of the simulation.

Bugs

The user is cautioned that this code is a beta release. If you are confident that you have found a bug in the peridynamic module, please report it in a *GitHub Issue <https://github.com/lammps/lammps/issues>* or send an email to the LAMMPS developers. First, check the New features and bug fixes section of the LAMMPS website site to see if the bug has already been reported or fixed. If not, the most useful thing you can do for us is to isolate the problem. Run it on the smallest number of atoms and fewest number of processors and with the simplest input script that reproduces the bug. In your message, describe the problem and any ideas you have as to what is causing it or where in the code the problem might be. We'll request your input script and data files if necessary.

Modifying and Extending the Peridynamic Module

To add new features or peridynamic potentials to the peridynamic module, the user is referred to the *Modifying & extending LAMMPS* section. To develop a new bond-based material, start with the *peri/pmb* pair style as a template. To develop a new state-based material, start with the *peri/lps* pair style as a template.

A Numerical Example

To introduce the peridynamic implementation within LAMMPS, we replicate a numerical experiment taken from section 6 of *(Silling 2005)*.

Problem Description and Setup

We consider the impact of a rigid sphere on a homogeneous disk of brittle material. The sphere has diameter 0.01 m and velocity 100 m/s directed normal to the surface of the target. The target material has density ρ = 2200 kg/m:math:^*3*. A PMB material model is used with K = 14.9 GPa and critical bond stretch parameters given by s_{00} = 0.0005 and α = 0.25. A three-dimensional simple cubic lattice is constructed with lattice constant 0.0005 m and horizon 0.0015 m. (The horizon is three times the lattice constant.) The target is a cylinder of diameter 0.074 m and thickness 0.0025 m, and the associated lattice contains 103,110 particles. Each particle i has volume fraction $V_i = 1.25 \times 10^{-10}\text{m}^3$.

The spring constant in the PMB material model is (see *(6)*)

$$c = \frac{18k}{\pi\delta^4} = \frac{18(14.9 \times 10^9)}{\pi(1.5 \times 10^{-3})^4} \approx 1.6863 \times 10^{22}.$$

The CFL analysis from *(Silling2005)* shows that a timestep of 1.0×10^{-7} is safe.

We observe here that in IEEE double-precision floating point arithmetic when computing the bond stretch $s(t, \eta, \xi)$ at each iteration where $\|\eta + \xi\|$ is computed during the iteration and $\|\xi\|$ was computed and stored for the initial lattice, it may be that $fl(s) = \varepsilon$ with $|\varepsilon| \leq \varepsilon_{machine}$ for an unstretched bond. Taking $\varepsilon = 2.220446049250313 \times 10^{-16}$, we see that the value $csV_i \approx 4.68 \times 10^{-4}$, computed when determining f, is perhaps larger than we would like, especially when the true force should be zero. One simple way to avoid this issue is to insert the following instructions in Algorithm *Algorithm 5* after instruction 21 (and similarly for Algorithm *Algorithm 2*):

if $|dr| < \varepsilon_{machine}$ **then**
    $dr = 0$
end if

Qualitatively, this says that displacements from equilibrium on the order of 10^{-16}m are taken to be exactly zero, a seemingly reasonable assumption.

The Projectile

The projectile used in the following experiments is not the one used in *(Silling 2005)*. The projectile used here exerts a force

$$F(r) = -k_s(r - R)^2$$

on each atom where k_s is a specified force constant, r is the distance from the atom to the center of the indenter, and R is the radius of the projectile. The force is repulsive and $F(r) = 0$ for $r > R$. For our problem, the projectile radius R = 0.05 m, and we have chosen $k_s = 1.0 \times 10^{17}$ (compare with *(6)* above).

Writing the LAMMPS Input File

We discuss the example input script *listed below*. In line 2 we specify that SI units are to be used. We specify the dimension (3) and boundary conditions ("shrink-wrapped") for the computational domain in lines 3 and 4. In line 5 we specify that peridynamic particles are to be used for this simulation. In line 7, we set the "skin" distance used in building the LAMMPS neighbor list. In line 8 we set the lattice constant (in meters) and in line 10 we define the spatial region where the target will be placed. In line 12 we specify a rectangular box enclosing the target region that defines the simulation domain. Line 14 fills the target region with atoms. Lines 15 and 17 define the peridynamic material model, and lines 19 and 21 set the particle density and particle volume, respectively. The particle volume should be set to the cube of the lattice constant for a simple cubic lattice. Line 23 sets the initial velocity of all particles to zero. Line 25 instructs LAMMPS to integrate time with velocity-Verlet, and lines 27-30 create the spherical projectile, sending it with a velocity of 100 m/s towards the target. Line 32 declares a compute style for the damage (percentage of broken bonds) associated with each particle. Line 33 sets the timestep, line 34 instructs LAMMPS to provide a screen dump of

thermodynamic quantities every 200 timesteps, and line 35 instructs LAMMPS to create a data file (dump.peri) with a complete snapshot of the system every 100 timesteps. This file can be used to create still images or movies. Finally, line 36 instructs LAMMPS to run for 2000 timesteps.

Listing 5: Peridynamics Example LAMMPS Input Script

```
1  # 3D Peridynamic simulation with projectile"
2  units           si
3  dimension       3
4  boundary        s s s
5  atom_style      peri
6  atom_modify     map array
7  neighbor        0.0010 bin
8  lattice         sc 0.0005
9  # Create desired target
10 region          target cylinder y 0.0 0.0 0.037 -0.0025 0.0 units box
11 # Make 1 atom type
12 create_box      1 target
13 # Create the atoms in the simulation region
14 create_atoms    1 region target
15 pair_style      peri/pmb
16 #               <type1> <type2>    <c>    <horizon>  <s00> <alpha>
17 pair_coeff         *       *    1.6863e22 0.0015001 0.0005  0.25
18 # Set mass density
19 set             group all density 2200
20 # volume = lattice constant^3
21 set             group all volume 1.25e-10
22 # Zero out velocities of particles
23 velocity        all set 0.0 0.0 0.0 sum no units box
24 # Use velocity-Verlet time integrator
25 fix             F1 all nve
26 # Construct spherical indenter to shatter target
27 variable        y0 equal 0.00510
28 variable        vy equal -100
29 variable        y equal "v_y0 + step*dt*v_vy"
30 fix             F2 all indent 1e17 sphere 0.0000 v_y 0.0000 0.0050 units box
31 # Compute damage for each particle
32 compute         C1 all damage/atom
33 timestep        1.0e-7
34 thermo          200
35 dump            D1 all custom 100 dump.peri id type x y z c_C1
36 run             2000
```

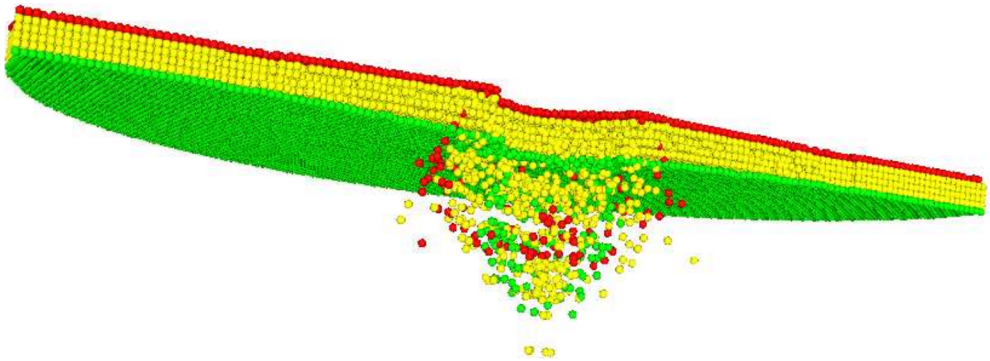
Note

To use the LPS model, replace line 15 with *pair_style peri/lps* and modify line 16 accordingly.

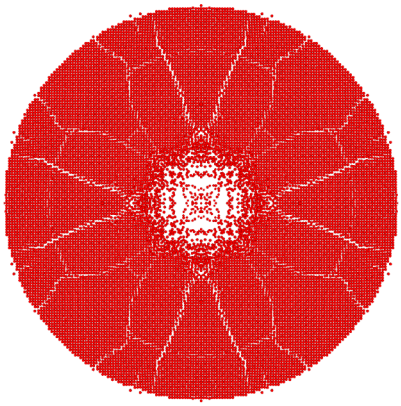
Numerical Results and Discussion

We ran the *input script from above*. Images of the disk (projectile not shown) appear in Figure below. The plot of damage on the top monolayer was created by coloring each particle according to its damage.

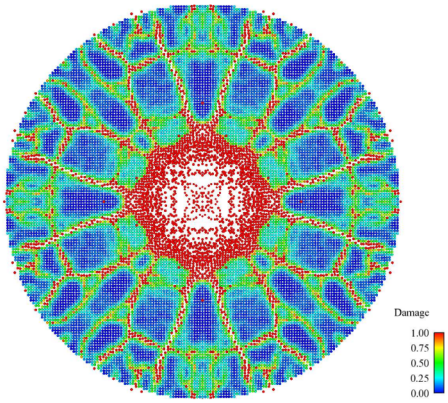
The symmetry in the computed solution arises because a "perfect" lattice was used, and a because a perfectly spherical projectile impacted the lattice at its geometric center. To break the symmetry in the solution, the nodes in the peridynamic body may be perturbed slightly from the lattice sites. To do this, the lattice of points can be slightly perturbed using the *displace_atoms* command.



(a) Cut view of target during impact.



(b) Top monolayer showing fragmentation.



(c) Top monolayer showing damage. (blue = 0% broken bonds; red = 100% broken bonds)

Fig. 3: Target during (a) and after (b,c) impact

(Emmrich) Emmrich, Weckner, Commun. Math. Sci., 5, 851-864 (2007),

(Parks) Parks, Lehoucq, Plimpton, Silling, Comp Phys Comm, 179(11), 777-783 (2008).

(Silling 2000) Silling, J Mech Phys Solids, 48, 175-209 (2000).

(Silling 2005) Silling Askari, Computer and Structures, 83, 1526-1535 (2005).

(Silling 2007) Silling, Epton, Weckner, Xu, Askari, J Elasticity, 88, 151-184 (2007).

(Seleson 2010) Seleson, Parks, Int J Mult Comp Eng 9(6), pp. 689-706, 2011.

8.5.10 Manifolds (surfaces)

Overview:

This page is not about a LAMMPS input script command, but about manifolds, which are generalized surfaces, as defined and used by the MANIFOLD package, to track particle motion on the manifolds. See the src/MANIFOLD/README file for more details about the package and its commands.

Below is a list of currently supported manifolds by the MANIFOLD package, their parameters and a short description of them. The parameters listed here are in the same order as they should be passed to the relevant fixes.

manifold	*parameters*	*equation*	*description*
cylinder	R	x^2 + y^2 - R^2 = 0	Cylinder along z-axis, axis going through (0,0,0)
cylinder_dei	R l a	x^2 + y^2 - r(z)^2 = 0, r(x) = R if | z | > l, r(z) = R - a*(1 + cos(z/l))/2 otherwise	A cylinder with a dent around z = 0
dumbbell	a A B c	-(x^2 + y^2) + (a^2 - z^2/c^2) * (1 + (A*sin(B*z^2))^4) = 0	A dumbbell
ellipsoid	a b c	(x/a)^2 + (y/b)^2 + (z/c)^2 = 0	An ellipsoid
gaussian_bu	A l rc1 rc2	if(x < rc1) -z + A * exp(-x^2 / (2 l^2)); else if(x < rc2) -z + a + b*x + c*x^2 + d*x^3; else z	A Gaussian bump at x = y = 0, smoothly tapered to a flat plane z = 0.
plane	a b c x0 y0 z0	a*(x-x0) + b*(y-y0) + c*(z-z0) = 0	A plane with normal (a,b,c) going through point (x0,y0,z0)
plane_v	a w	z - a*sin(w*x) = 0	A plane with a sinusoidal modulation on z along x.
sphere	R	x^2 + y^2 + z^2 - R^2 = 0	A sphere of radius R
supersphere	R q	| x |^q + | y |^q + | z |^q - R^q = 0	A supersphere of hyperradius R
spine	a, A, B, B2, c	-(x^2 + y^2) + (a^2 - z^2/f(z)^2)*(1 + (A*sin(g(z)*z^2))^4), f(z) = c if z > 0, 1 otherwise; g(z) = B if z > 0, B2 otherwise	An approximation to a dendritic spine
spine_t	a, A, B, B2, c	-(x^2 + y^2) + (a^2 - z^2/f(z)^2)*(1 + (A*sin(g(z)*z^2))^2), f(z) = c if z > 0, 1 otherwise; g(z) = B if z > 0, B2 otherwise	Another approximation to a dendritic spine
thylakoid	wB LB lB	Various, see *(Paquay)*	A model grana thylakoid consisting of two block-like compartments connected by a bridge of width wB, length LB and taper length lB
torus	R r	(R - sqrt(x^2 + y^2))^2 + z^2 - r^2	A torus with large radius R and small radius r, centered on (0,0,0)

(Paquay) Paquay and Kusters, Biophys. J., 110, 6, (2016). preprint available at arXiv:1411.3019.

8.5.11 Reproducing hydrodynamics and elastic objects (RHEO)

The RHEO package is a hybrid implementation of smoothed particle hydrodynamics (SPH) for fluid flow, which can couple to the *BPM package* to model solid elements. RHEO combines these methods to enable mesh-free modeling of multi-phase material systems. Its SPH solver supports many advanced options including reproducing kernels, particle shifting, free surface identification, and solid surface reconstruction. To model fluid-solid systems, the status of particles can dynamically change between a fluid and solid state, e.g. during melting/solidification, which determines how they interact and their physical behavior. The package is designed with modularity in mind, so one can easily turn various features on/off, adjust physical details of the system, or develop new capabilities. For instance, the numerics associated with calculating gradients, reproducing kernels, etc. are separated into distinct classes to simplify the development of new integration schemes which can call these calculations. Additional numerical details can be found in *(Clemmer)*. Example movies illustrating some of these capabilities are found at https://www.lammps.org/movies.html#rheopackage.

Note, if you simply want to run a traditional SPH simulation, the *SPH package* package is likely better suited for your application. It has fewer advanced features and therefore benefits from improved performance. The *MACHDYN* package for solids may also be relevant for fluid-solid problems.

At the core of the package is *fix rheo* which integrates particle trajectories and controls many optional features (e.g. the use of reproducing kernels). In conjunction to fix rheo, one must specify an instance of *fix rheo/pressure* and *fix rheo/viscosity* to define a pressure equation of state and viscosity model, respectively. Optionally, one can model a heat equation with *fix rheo/thermal*, which also allows the user to specify equations for a particle's thermal conductivity, specific heat, latent heat, and melting temperature. The ordering of these fixes in an an input script matters. Fix rheo must be defined prior to all other RHEO fixes.

Typically, RHEO requires atom style rheo. In addition to typical atom properties like positions and forces, particles store a local density, viscosity, pressure, and status. If thermal evolution is modeled, one must use atom style rheo/thermal which also includes a local energy, temperature, and conductivity. Note that the temperature is always derived from the energy. This implies the *temperature* attribute of *the set command* does not affect particles. Instead, one should use the *sph/e* attribute.

The status variable uses bit-masking to track various properties of a particle such as its current state of matter (fluid or solid) and its location relative to a surface. Some of these properties (and others) can be accessed using *compute rheo/property/atom*. The *status* attribute in *the set command* only allows control over the first bit which sets the state of matter, 0 is fluid and 1 is solid.

Fluid interactions, including pressure forces, viscous forces, and heat exchange, are calculated using *pair rheo*. Unlike typical pair styles, pair rheo ignores the *special bond* settings. Instead, it determines whether to calculate forces based on the status of particles: e.g., hydrodynamic forces are only calculated if a fluid particle is involved.

To model elastic objects, there are currently two mechanisms in RHEO, one designed for bulk solid bodies and the other for thin shells. Both mechanisms rely on introducing bonded forces between particles and therefore require a hybrid of atom style bond and rheo (or rheo/thermal).

To create an elastic solid body, one has to (a) change the status of constituent particles to solid (e.g. with the *set* command), (b) create bpm bonds between the particles (see the *bpm howto* page for more details), and (c) use *pair rheo/solid* to apply repulsive contact forces between distinct solid bodies. Akin to pair rheo, pair rheo/solid considers a particles fluid/solid phase to determine whether to apply forces. However, unlike pair rheo, pair rheo/solid does obey special bond settings such that contact forces do not have to be calculated between two bonded solid particles in the same elastic body.

In systems with thermal evolution, fix rheo/thermal can optionally set a melting/solidification temperature allowing particles to dynamically swap their state between fluid and solid when the temperature exceeds or drops below the critical temperature, respectively. Using the *react* option, one can specify a maximum bond length and a bond type. Then, when solidifying, particles will search their local neighbors and automatically create bonds with any neighboring solid particles in range. For BPM bond styles, bonds will then use the immediate position of the two particles to calculate

a reference state. When melting, particles will delete any bonds of the specified type when reverting to a fluid state. Special bonds are updated as bonds are created/broken.

The other option for elastic objects is an elastic shell that is nominally much thinner than a particle diameter, e.g. a oxide skin which gradually forms over time on the surface of a fluid. Currently, this is implemented using *fix rheo/oxidation* and bond style *rheo/shell*. Essentially, fix rheo/oxidation creates candidate bonds of a specified type between surface fluid particles within a specified distance. a newly created rheo/shell bond will then start a timer. While the timer is counting down, the bond will delete itself if particles move too far apart or move away from the surface. However, if the timer reaches a user-defined threshold, then the bond will activate and apply additional forces to the fluid particles. Bond style rheo/shell then operates very similarly to a BPM bond style, storing a reference length and breaking if stretched too far. Unlike the above method, this option does not remove the underlying fluid interactions (although particle shifting is turned off) and does not modify special bond settings of particles.

While these two options are not expected to be appropriate for every system, either framework can be modified to create more suitable models (e.g. by changing the criteria for creating/deleting a bond or altering force calculations).

(Clemmer) Clemmer, Pierce, O'Connor, Nevins, Jones, Lechman, Tencer, Appl. Math. Model., 130, 310-326 (2024).

8.5.12 Magnetic spins

The magnetic spin simulations are enabled by the SPIN package, whose implementation is detailed in *Tranchida*.

The model represents the simulation of atomic magnetic spins coupled to lattice vibrations. The dynamics of those magnetic spins can be used to simulate a broad range a phenomena related to magneto-elasticity, or or to study the influence of defects on the magnetic properties of materials.

The magnetic spins are interacting with each others and with the lattice via pair interactions. Typically, the magnetic exchange interaction can be defined using the *pair/spin/exchange* command. This exchange applies a magnetic torque to a given spin, considering the orientation of its neighboring spins and their relative distances. It also applies a force on the atoms as a function of the spin orientations and their associated inter-atomic distances.

The command *fix precession/spin* allows to apply a constant magnetic torque on all the spins in the system. This torque can be an external magnetic field (Zeeman interaction), and an uniaxial or cubic magnetic anisotropy.

A Langevin thermostat can be applied to those magnetic spins using *fix langevin/spin*. Typically, this thermostat can be coupled to another Langevin thermostat applied to the atoms using *fix langevin* in order to simulate thermostatted spin-lattice systems.

The magnetic damping can also be applied using *fix langevin/spin*. It allows to either dissipate the thermal energy of the Langevin thermostat, or to perform a relaxation of the magnetic configuration toward an equilibrium state.

The command *fix setforce/spin* allows to set the components of the magnetic precession vectors (while erasing and replacing the previously computed magnetic precession vectors on the atom). This command can be used to freeze the magnetic moment of certain atoms in the simulation by zeroing their precession vector.

The command *fix nve/spin* can be used to perform a symplectic integration of the combined dynamics of spins and atomic motions.

The minimization style *min/spin* can be applied to the spins to perform a minimization of the spin configuration.

All the computed magnetic properties can be output by two main commands. The first one is *compute spin*, that enables to evaluate magnetic averaged quantities, such as the total magnetization of the system along x, y, or z, the spin temperature, or the magnetic energy. The second command is *compute property/atom*. It enables to output all the per atom magnetic quantities. Typically, the orientation of a given magnetic spin, or the magnetic force acting on this spin.

(Tranchida) Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

8.6 Tutorials howto

8.6.1 Using CMake with LAMMPS

The support for building LAMMPS with CMake is a recent addition to LAMMPS thanks to the efforts of Christoph Junghans (LANL) and Richard Berger (LANL). One of the key strengths of CMake is that it is not tied to a specific platform or build system. Instead it generates the files necessary to build and develop for different build systems and on different platforms. Note, that this applies to the build system itself not the LAMMPS code. In other words, without additional porting effort, it is not possible - for example - to compile LAMMPS with Visual C++ on Windows. The build system output can also include support files necessary to program LAMMPS as a project in integrated development environments (IDE) like Eclipse, Visual Studio, QtCreator, Xcode, CodeBlocks, Kate and others.

A second important feature of CMake is that it can detect and validate available libraries, optimal settings, available support tools and so on, so that by default LAMMPS will take advantage of available tools without requiring to provide the details about how to enable/integrate them.

The downside of this approach is, that there is some complexity associated with running CMake itself and how to customize the building of LAMMPS. This tutorial will show how to manage this through some selected examples. Please see the chapter about *building LAMMPS* for descriptions of specific flags and options for LAMMPS in general and for specific packages.

CMake can be used through either the command-line interface (CLI) program `cmake` (or `cmake3`), a text mode interactive user interface (TUI) program `ccmake` (or `ccmake3`), or a graphical user interface (GUI) program `cmake-gui`. All of them are portable software available on all supported platforms and can be used interchangeably. As of LAMMPS version 2 August 2023, the minimum required CMake version is 3.16.

All details about features and settings for CMake are in the CMake online documentation. We focus below on the most important aspects with respect to compiling LAMMPS.

Prerequisites

This tutorial assumes that you are operating in a command-line environment using a shell like Bash or Zsh.

- Linux: any Terminal window will work or text console
- macOS: launch the Terminal application
- Windows 10 or 11: install and run the *Windows Subsystem for Linux*
- other Unix-like operating systems like FreeBSD

> **Note**
>
> It is also possible to use CMake on Windows 10 or 11 through either the Microsoft Visual Studio IDE with the bundled CMake or from the Windows command prompt using a separately installed CMake package, both using the native Microsoft Visual C++ compilers and (optionally) the Microsoft MPI SDK. This tutorial, however, only covers unix-like command line interfaces.

We also assume that you have downloaded and unpacked a recent LAMMPS source code package or used Git to create a clone of the LAMMPS sources on your compilation machine.

You should change into the top level directory of the LAMMPS source tree all paths mentioned in the tutorial are relative to that. Immediately after downloading it should look like this:

```
$ ls
bench  doc      lib      potentials  README  tools
cmake  examples  LICENSE  python     src
```

Build versus source directory

When using CMake the build procedure is separated into multiple distinct phases:

1. **Configuration:** detect or define which features and settings should be enable and used and how LAMMPS should be compiled
2. **Compilation:** generate and compile all necessary source files and build libraries and executables.
3. **Installation:** copy selected files from the compilation into your file system, so they can be used without having to keep the source and build tree around.

The configuration and compilation of LAMMPS has to happen in a dedicated *build directory* which must be different from the source directory. Also the source directory (src) must remain pristine, so it is not allowed to "install" packages using the traditional make process and after an compilation attempt all created source files must be removed. This can be achieved with make no-all purge.

You can pick **any** folder outside the source tree. We recommend to create a folder build in the top-level directory, or multiple folders in case you want to have separate builds of LAMMPS with different options (build-parallel, build-serial) or with different compilers (build-gnu, build-clang, build-intel) and so on. All the auxiliary files created by one build process (executable, object files, log files, etc) are stored in this directory or subdirectories within it that CMake creates.

Running CMake

CLI version

In the (empty) build directory, we now run the command cmake ../cmake, which will start the configuration phase and you will see the progress of the configuration printed to the screen followed by a summary of the enabled features, options and compiler settings. A typical summary screen will look like this:

```
$ cmake ../cmake/
-- The CXX compiler identification is GNU 8.2.0
-- Check for working CXX compiler: /opt/tools/gcc-8.2.0/bin/c++
-- Check for working CXX compiler: /opt/tools/gcc-8.2.0/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Git: /usr/bin/git (found version "2.25.2")
-- Running check for auto-generated files from make-based build system
-- Found MPI_CXX: /usr/lib64/mpich/lib/libmpicxx.so (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
-- Looking for C++ include omp.h
-- Looking for C++ include omp.h - found
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Found JPEG: /usr/lib64/libjpeg.so (found version "62")
-- Found PNG: /usr/lib64/libpng.so (found version "1.6.37")
```

(continues on next page)

(continued from previous page)

```
-- Found ZLIB: /usr/lib64/libz.so (found version "1.2.11")
-- Found GZIP: /usr/bin/gzip
-- Found FFMPEG: /usr/bin/ffmpeg
-- Performing Test COMPILER_SUPPORTS-ffast-math
-- Performing Test COMPILER_SUPPORTS-ffast-math - Success
-- Performing Test COMPILER_SUPPORTS-march=native
-- Performing Test COMPILER_SUPPORTS-march=native - Success
-- Looking for C++ include cmath
-- Looking for C++ include cmath - found
-- Generating style_angle.h...
[...]
-- Generating lmpinstalledpkgs.h...
-- The following tools and libraries have been found and configured:
 * Git
 * MPI
 * OpenMP
 * JPEG
 * PNG
 * ZLIB

-- <<< Build configuration >>>
   Build type:       RelWithDebInfo
   Install path:     /home/akohlmey/.local
   Generator:        Unix Makefiles using /usr/bin/gmake
-- <<< Compilers and Flags: >>>
-- C++ Compiler:     /opt/tools/gcc-8.2.0/bin/c++
      Type:          GNU
      Version:       8.2.0
      C++ Flags:     -O2 -g -DNDEBUG
      Defines:       LAMMPS_SMALLBIG;LAMMPS_MEMALIGN=64;LAMMPS_JPEG;LAMMPS_
↪PNG;LAMMPS_GZIP;LAMMPS_FFMPEG
      Options:       -ffast-math;-march=native
-- <<< Linker flags: >>>
-- Executable name:  lmp
-- Static library flags:
-- <<< MPI flags >>>
-- MPI includes:     /usr/include/mpich-x86_64
-- MPI libraries:    /usr/lib64/mpich/lib/libmpicxx.so;/usr/lib64/mpich/lib/libmpi.so;
-- Configuring done
-- Generating done
-- Build files have been written to: /home/akohlmey/compile/lammps/build
```

The cmake command has one mandatory argument, and that is a folder with either the file CMakeLists.txt or CMakeCache.txt. The CMakeCache.txt file is created during the CMake configuration run and contains all active settings, thus after a first run of CMake all future runs in the build folder can use the folder . and CMake will know where to find the CMake scripts and reload the settings from the previous step. This means, that one can modify an existing configuration by re-running CMake, but only needs to provide flags indicating the desired change, everything else will be retained. One can also mix compilation and configuration, i.e. start with a minimal configuration and then, if needed, enable additional features and recompile.

The steps above **will NOT compile the code**. The compilation can be started in a portable fashion with cmake --build ., or you use the selected built tool, e.g. make.

TUI version

For the text mode UI CMake program the basic principle is the same. You start the command ccmake ../cmake in the build folder.

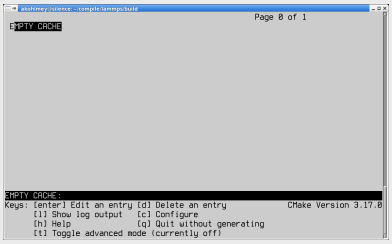


Fig. 4: Initial ccmake screen

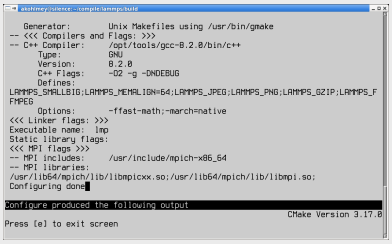


Fig. 5: Configure output of ccmake

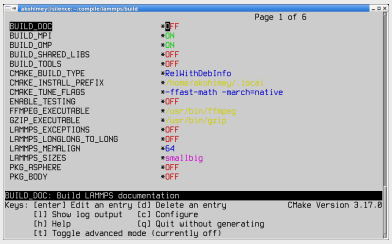


Fig. 6: Options screen of ccmake

This will show you the initial screen (left image) with the empty configuration cache. Now you type the 'c' key to run the configuration step. That will do a first configuration run and show the summary (center image). You exit the summary screen with 'e' and see now the main screen with detected options and settings. You can now make changes by moving and down with the arrow keys of the keyboard and modify entries. For on/off settings, the enter key will toggle the state. For others, hitting enter will allow you to modify the value and you commit the change by hitting the enter key again or cancel using the escape key. All "new" settings will be marked with a star '*' and for as long as one setting is marked like this, you have to re-run the configuration by hitting the 'c' key again, sometimes multiple times unless the TUI shows the word "generate" next to the letter 'g' and by hitting the 'g' key the build files will be written to the folder and the TUI exits. You can quit without generating build files by hitting 'q'.

GUI version

For the graphical CMake program the steps are similar to the TUI version. You can type the command cmake-gui ../cmake in the build folder. In this case the path to the CMake script folder is not required, it can also be entered from the GUI.

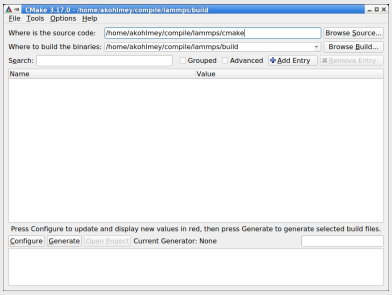


Fig. 7: Initial cmake-gui screen

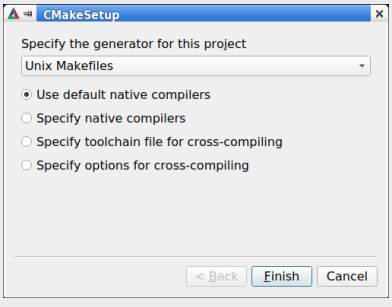


Fig. 8: Generator selection in cmake-gui

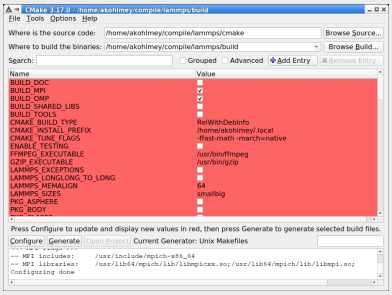


Fig. 9: Options screen of cmake-gui

Again, you start with an empty configuration cache (left image) and need to start the configuration step. For the very first configuration in a folder, you will have a pop-up dialog (center image) asking to select the desired build tool and some configuration settings (stick with the default) and then you get the option screen with all new settings highlighted in red. You can modify them (or not) and click on the "configure" button again until satisfied and click on the "generate" button to write out the build files. You can exit the GUI from the "File" menu or hit "ctrl-q".

Setting options

Options that enable, disable or modify settings are modified by setting the value of CMake variables. This is done on the command line with the *-D* flag in the format -D VARIABLE=value, e.g. -D CMAKE_BUILD_TYPE=Release or -D BUILD_MPI=on. There is one quirk: when used before the CMake directory, there may be a space between the *-D* flag and the variable, after it must not be. Such CMake variables can have boolean values (on/off, yes/no, or 1/0 are all valid) or are strings representing a choice, or a path, or are free format. If the string would contain whitespace, it must be put in quotes, for example -D CMAKE_TUNE_FLAGS="-ftree-vectorize -ffast-math".

CMake variables fall into two categories: 1) common CMake variables that are used by default for any CMake configuration setup and 2) project specific variables, i.e. settings that are specific for LAMMPS. Also CMake variables can be flagged as *advanced*, which means they are not shown in the text mode or graphical CMake program in the overview of all settings by default, but only when explicitly requested (by hitting the 't' key or clicking on the 'Advanced' check-box).

Some common CMake variables

Variable	Description
CMAKE_INSTALL_PREFIX	root directory of install location for make install (default: $HOME/.local)
LAMMPS_INSTALL_RPATH	set or remove runtime path setting from binaries for make install (default: off)
CMAKE_BUILD_TYPE	controls compilation options: one of RelWithDebInfo (default), Release, Debug, MinSizeRel
BUILD_SHARED_LIBS	if set to on build the LAMMPS library as shared library (default: off)
CMAKE_MAKE_PROGRAM	name/path of the compilation command (default depends on *-G* option, usually make)
CMAKE_VERBOSE_MAKE	if set to on echo commands while executing during build (default: off)
CMAKE_C_COMPILER	C compiler to be used for compilation (default: system specific, gcc on Linux)
CMAKE_CXX_COMPILER	C++ compiler to be used for compilation (default: system specific, g++ on Linux)
CMAKE_Fortran_COMPILER	Fortran compiler to be used for compilation (default: system specific, gfortran on Linux)
CXX_COMPILER_LAUNCHER	tool to launch the C++ compiler, e.g. ccache or distcc for faster compilation (default: empty)

Some common LAMMPS specific variables

Variable	Description
BUILD_MPI	build LAMMPS with MPI support (default: on if a working MPI available, else off)
BUILD_OMP	build LAMMPS with OpenMP support (default: on if compiler supports OpenMP fully, else off)
BUILD_TOOLS	compile some additional executables from the tools folder (default: off)
BUILD_DOC	include building the HTML format documentation for packaging/installing (default: off)
CMAKE_TUNE_FL/	common compiler flags, for optimization or instrumentation (default:)
LAMMPS_MACHINE	when set to name the LAMMPS executable and library will be called lmp_name and liblammps_name.a
FFT	select which FFT library to use: FFTW3, MKL, KISS (default, unless FFTW3 is found)
FFT_KOKKOS	select which FFT library to use in Kokkos-enabled styles: FFTW3, MKL, HIPFFT, CUFFT, KISS (default)
FFT_SINGLE	select whether to use single precision FFTs (default: off)
WITH_JPEG	whether to support JPEG format in *dump image* (default: on if found)
WITH_PNG	whether to support PNG format in *dump image* (default: on if found)
WITH_GZIP	whether to support reading and writing compressed files (default: on if found)
WITH_FFMPEG	whether to support generating movies with *dump movie* (default: on if found)

Enabling or disabling LAMMPS packages

The LAMMPS software is organized into a common core that is always included and a large number of *add-on packages* that have to be enabled to be included into a LAMMPS executable. Packages are enabled through setting variables of the kind PKG_<NAME> to on and disabled by setting them to off (or using yes, no, 1, 0 correspondingly). <NAME> has to be replaced by the name of the package, e.g. MOLECULE or EXTRA-PAIR.

Using presets

Since LAMMPS has a lot of optional features and packages, specifying them all on the command line can be tedious. Or when selecting a different compiler toolchain, multiple options have to be changed consistently and that is rather error prone. Or when enabling certain packages, they require consistent settings to be operated in a particular mode. For this purpose, we are providing a selection of "preset files" for CMake in the folder cmake/presets. They represent a way to pre-load or override the CMake configuration cache by setting or changing CMake variables. Preset files are loaded using the *-C* command line flag. You can combine loading multiple preset files or change some variables later with additional *-D* flags. A few examples:

```
cmake -C ../cmake/presets/basic.cmake -D PKG_MISC=on ../cmake
cmake -C ../cmake/presets/clang.cmake -C ../cmake/presets/most.cmake ../cmake
cmake -C ../cmake/presets/basic.cmake -D BUILD_MPI=off ../cmake
```

The first command will install the packages KSPACE, MANYBODY, MOLECULE, RIGID and MISC; the first four from the preset file and the fifth from the explicit variable definition. The second command will first switch the compiler toolchain to use the Clang compilers and install a large number of packages that are not depending on any special external libraries or tools and are not very unusual. The third command will enable the first four packages like above and then enforce compiling LAMMPS as a serial program (using the MPI STUBS library).

It is also possible to do this incrementally.

```
cmake -C ../cmake/presets/basic.cmake ../cmake
cmake -D PKG_MISC=on .
```

will achieve the same final configuration as in the first example above. In this scenario it is particularly convenient to do the second configuration step using either the text mode or graphical user interface (ccmake or cmake-gui).

Note

Using a preset to select a compiler package (clang.cmake, gcc.cmake, intel.cmake, oneapi.cmake, or pgi.cmake) are an exception to the mechanism of updating the configuration incrementally, as they will trigger a reset of cached internal CMake settings and thus reset settings to their default values.

Compilation and build targets

The actual compilation will be started by running the selected build command (on Linux this is by default make, see below how to select alternatives). You can also use the portable command cmake --build . which will adapt to whatever the selected build command is. This is particularly convenient, if you have set a custom build command via the CMAKE_MAKE_PROGRAM variable.

When calling the build program, you can also select which "target" is to be build through appending the --target flag and the name of the target to the build command. When using make as build tool, you can just append the target name to the command. Example: cmake --build . --target all or make all. The following abstract targets are available:

Target	Description
all	build "everything" (default)
lammps	build the LAMMPS library and executable
doc	build the HTML documentation (if configured)
install	install all target files into folders in CMAKE_INSTALL_PREFIX
test	run some tests (if configured with -D ENABLE_TESTING=on)
clean	remove all generated files

Choosing generators

While CMake usually defaults to creating makefiles to compile software with the make program, it supports multiple alternate build tools (e.g. ninja-build which tends to be faster and more efficient in parallelizing builds than make) and can generate project files for integrated development environments (IDEs) like VisualStudio, Eclipse or CodeBlocks. This is specific to how the local CMake version was configured and compiled. The list of available options can be seen at the end of the output of cmake --help. Example on Fedora 31 this is:

```
Generators

The following generators are available on this platform (* marks default):
* Unix Makefiles               = Generates standard UNIX makefiles.
  Green Hills MULTI            = Generates Green Hills MULTI files
                                 (experimental, work-in-progress).
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files.
  CodeLite - Ninja             = Generates CodeLite project files.
  CodeLite - Unix Makefiles    = Generates CodeLite project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
```

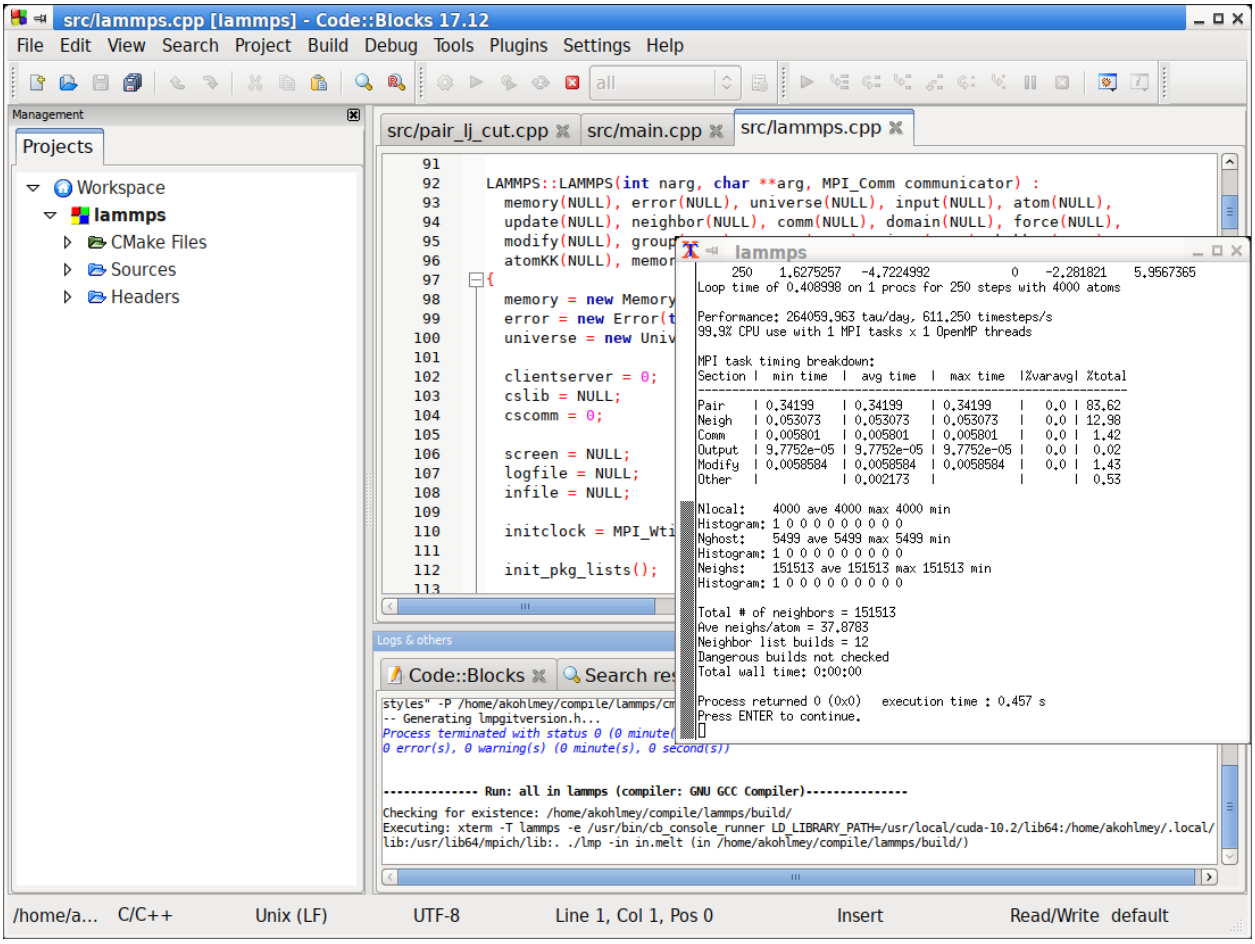
(continues on next page)

(continued from previous page)

```
Sublime Text 2 - Unix Makefiles
                        = Generates Sublime Text 2 project files.
Kate - Ninja                = Generates Kate project files.
Kate - Unix Makefiles       = Generates Kate project files.
Eclipse CDT4 - Ninja        = Generates Eclipse CDT 4.0 project files.
Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
```

Below is a screenshot of using the CodeBlocks IDE with the ninja build tool after running CMake as follows:

```
cmake -G 'CodeBlocks - Ninja' ../cmake/presets/most.cmake ../cmake/
```



8.6.2 LAMMPS GitHub tutorial

written by Stefan Paquay

This document describes the process of how to use GitHub to integrate changes or additions you have made to LAMMPS into the official LAMMPS distribution. It uses the process of updating this very tutorial as an example to describe the individual steps and options. You need to be familiar with git and you may want to have a look at the git book to familiarize yourself with some of the more advanced git features used below.

As of fall 2016, submitting contributions to LAMMPS via pull requests on GitHub is the preferred option for integrating contributed features or improvements to LAMMPS, as it significantly reduces the amount of work required by the

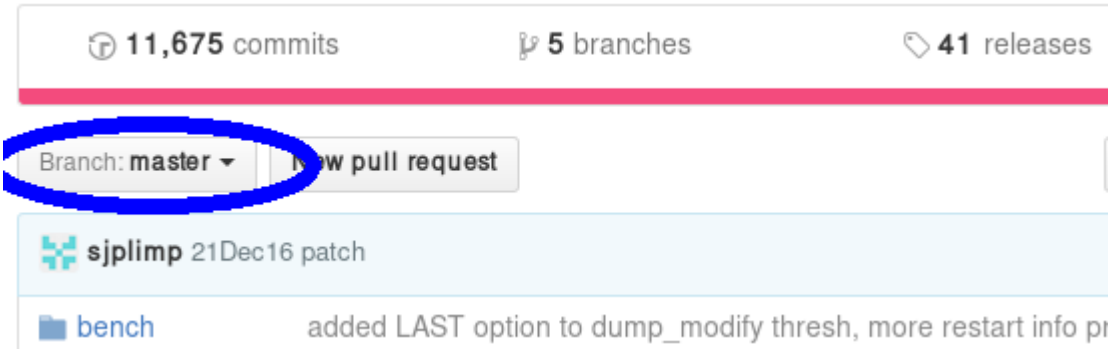
LAMMPS developers. Consequently, creating a pull request will increase your chances to have your contribution included and will reduce the time until the integration is complete. For more information on the requirements to have your code included into LAMMPS please see *this page*.

Making an account

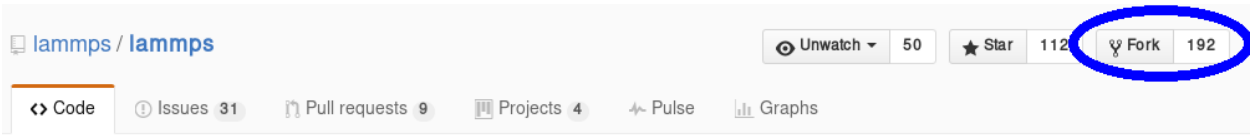
First of all, you need a GitHub account. This is fairly simple, just go to GitHub and create an account by clicking the "Sign up for GitHub" button. Once your account is created, you can sign in by clicking the button in the top left and filling in your username or e-mail address and password.

Forking the repository

To get changes into LAMMPS, you need to first fork the *lammps/lammps* repository on GitHub. At the time of writing, *develop* is the preferred target branch. Thus go to LAMMPS on GitHub and make sure branch is set to "develop", as shown in the figure below.



If it is not, use the button to change it to *develop*. Once it is, use the fork button to create a fork.



This will create a fork (which is essentially a copy, but uses less resources) of the LAMMPS repository under your own GitHub account. You can make changes in this fork and later file *pull requests* to allow the upstream repository to merge changes from your own fork into the one we just forked from (or others that were forked from the same repository). At the same time, you can set things up, so you can include changes from upstream into your repository and thus keep it in sync with the ongoing LAMMPS development.

Adding changes to your own fork

Additions to the upstream version of LAMMPS are handled using *feature branches*. For every new feature, a so-called feature branch is created, which contains only those modification relevant to one specific feature. For example, adding a single fix would consist of creating a branch with only the fix header and source file and nothing else. It is explained in more detail here: feature branch workflow.

Feature branches

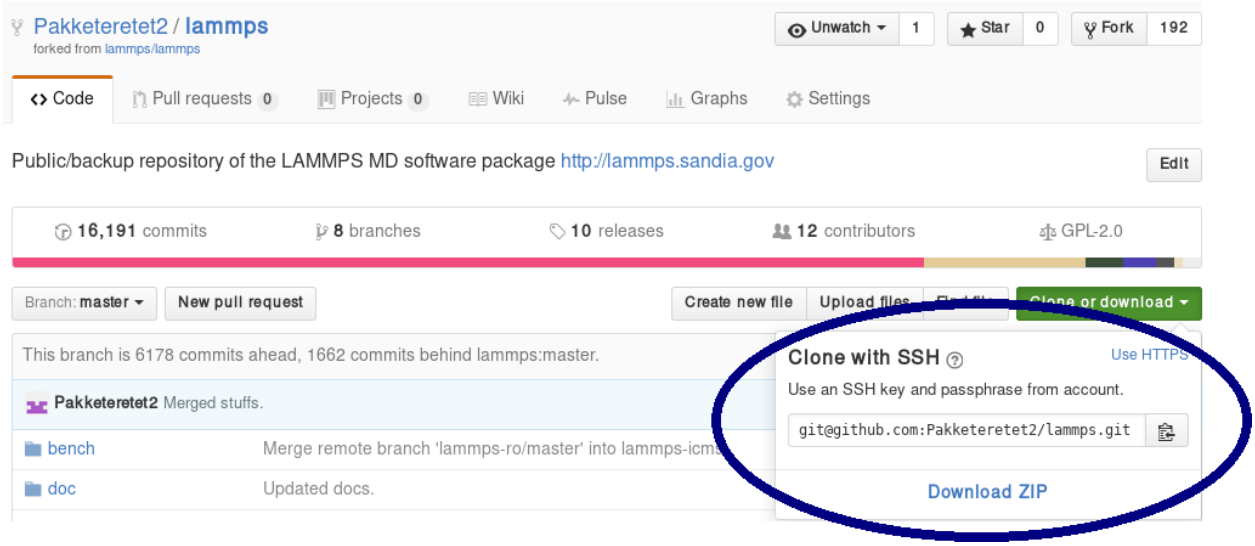
First of all, create a clone of your version on GitHub on your local machine via HTTPS:

```
git clone https://github.com/<your user name>/lammps.git <some name>
```

or, if you have set up your GitHub account for using SSH keys, via SSH:

```
git clone git@github.com:<your user name>/lammps.git
```

You can find the proper URL by clicking the "Clone or download"-button:



The above command copies ("clones") the git repository to your local machine to a directory with the name you chose. If none is given, it will default to "lammps". Typical names are "mylammps" or something similar.

You can use this local clone to make changes and test them without interfering with the repository on GitHub.

To pull changes from upstream into this copy, you can go to the directory and use git pull:

```
cd mylammps
git checkout develop
git pull https://github.com/lammps/lammps develop
```

You can also add this URL as a remote:

```
git remote add upstream https://www.github.com/lammps/lammps
```

From then on you can update your upstream branches with:

```
git fetch upstream
```

and then refer to the upstream repository branches with *upstream/develop* or *upstream/release* and so on.

At this point, you typically make a feature branch from the updated branch for the feature you want to work on. This tutorial contains the workflow that updated this tutorial, and hence we will call the branch "github-tutorial-update":

```
git fetch upstream
git checkout -b github-tutorial-update upstream/develop
```

Now that we have changed branches, we can make our changes to our local repository. Just remember that if you want to start working on another, unrelated feature, you should switch branches!

Note

Committing changes to the *develop*, *release*, or *stable* branches is strongly discouraged. While it may be convenient initially, it will create more work in the long run. Various texts and tutorials on using git effectively discuss the motivation for using feature branches instead.

After changes are made

After everything is done, add the files to the branch and commit them:

```
git add doc/src/Howto_github.txt
git add doc/src/JPG/tutorial*.png
```

Warning

Do not use *git commit -a* (or *git add -A*). The -a flag (or -A flag) will automatically include **all** modified **and** new files and that is rarely the behavior you want. It can easily lead to accidentally adding unrelated and unwanted changes into the repository. Instead it is preferable to explicitly use *git add*, *git rm*, *git mv* for adding, removing, renaming individual files, respectively, and then *git commit* to finalize the commit. Carefully check all pending changes with *git status* before committing them. If you find doing this on the command line too tedious, consider using a GUI, for example the one included in git distributions written in Tk, i.e. use *git gui* (on some Linux distributions it may be required to install an additional package to use it).

After adding all files, the change set can be committed with some useful message that explains the change.

```
git commit -m 'Finally updated the GitHub tutorial'
```

After the commit, the changes can be pushed to the same branch on GitHub:

```
git push
```

Git will ask you for your user name and password on GitHub if you have not configured anything. If your local branch is not present on GitHub yet, it will ask you to add it by running

```
git push --set-upstream origin github-tutorial-update
```

If you correctly type your user name and password, the feature branch should be added to your fork on GitHub.

If you want to make really sure you push to the right repository (which is good practice), you can provide it explicitly:

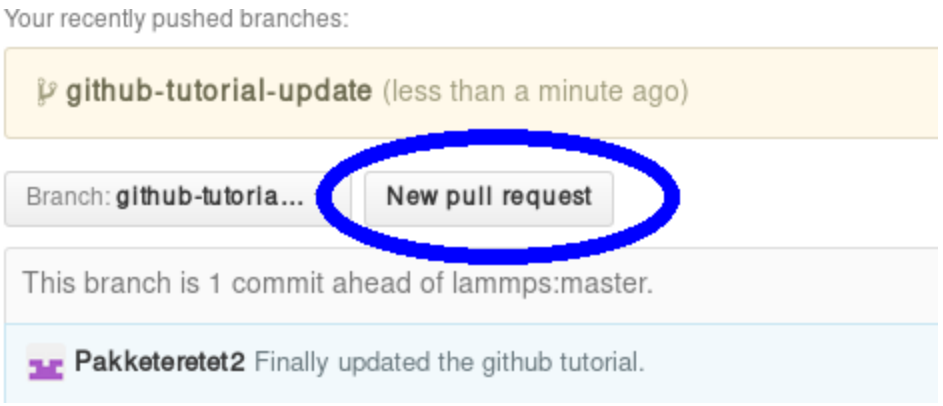
```
git push origin
```

or using an explicit URL:

```
git push git@github.com:Pakketeretet2/lammps.git
```

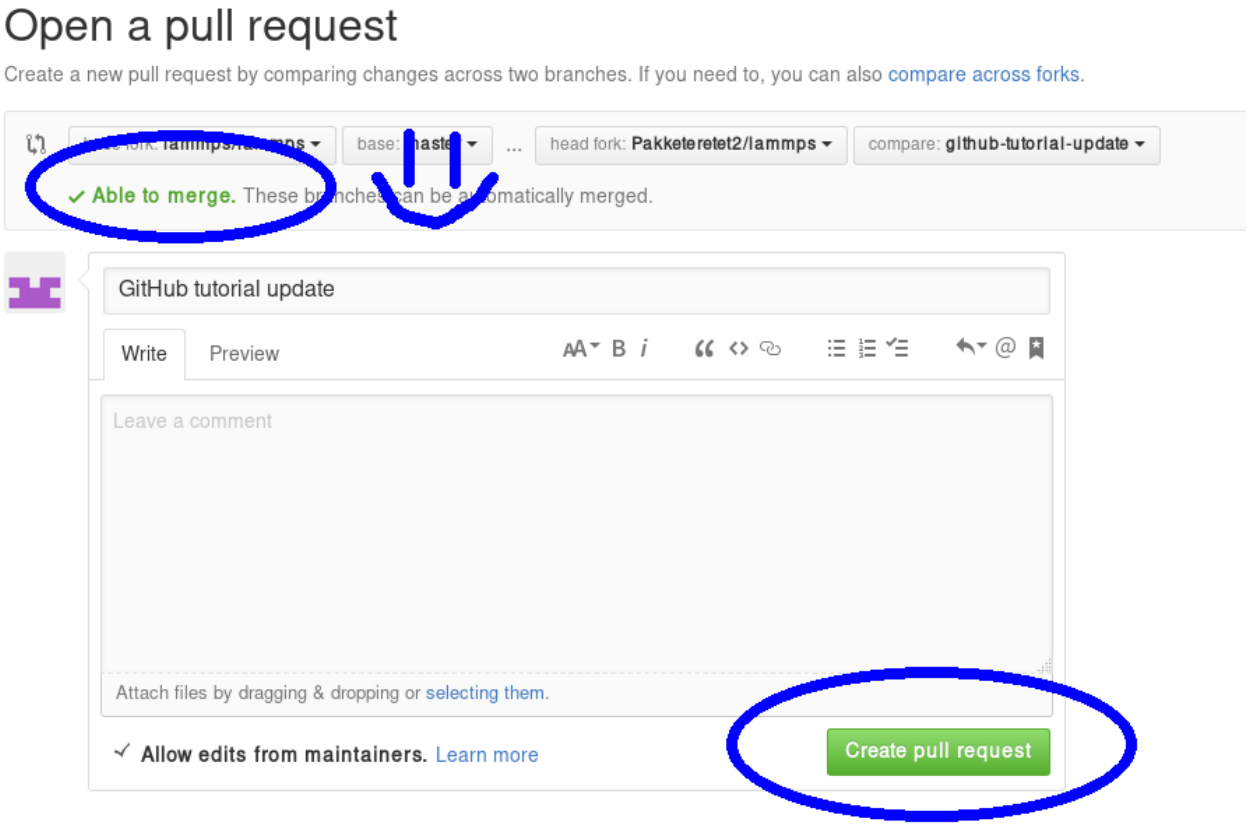
Filing a pull request

Up to this point in the tutorial, all changes were to *your* clones of LAMMPS. Eventually, however, you want this feature to be included into the official LAMMPS version. To do this, you will want to file a pull request by clicking on the "New pull request" button:



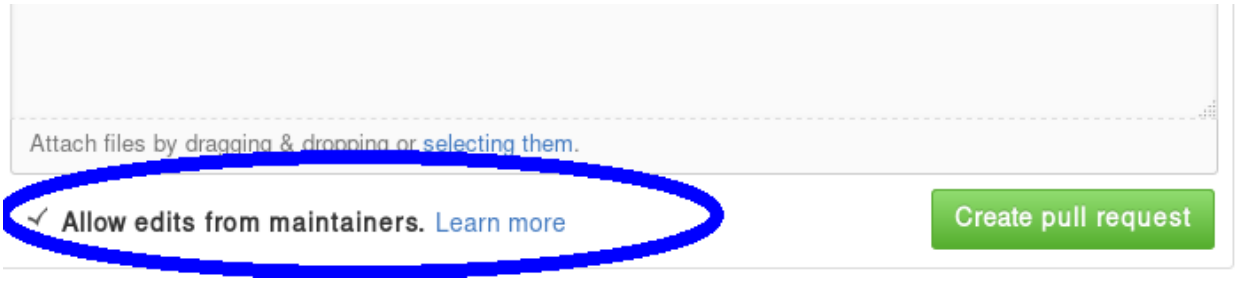
Make sure that the current branch is set to the correct one, which, in this case, is "github-tutorial-update". If done correctly, the only changes you will see are those that were made on this branch.

This will open up a new window that lists changes made to the repository. If you are just adding new files, there is not much to do, but I suppose merge conflicts are to be resolved here if there are changes in existing files. If all changes can automatically be merged, green text at the top will say so and you can click the "Create pull request" button, see image.



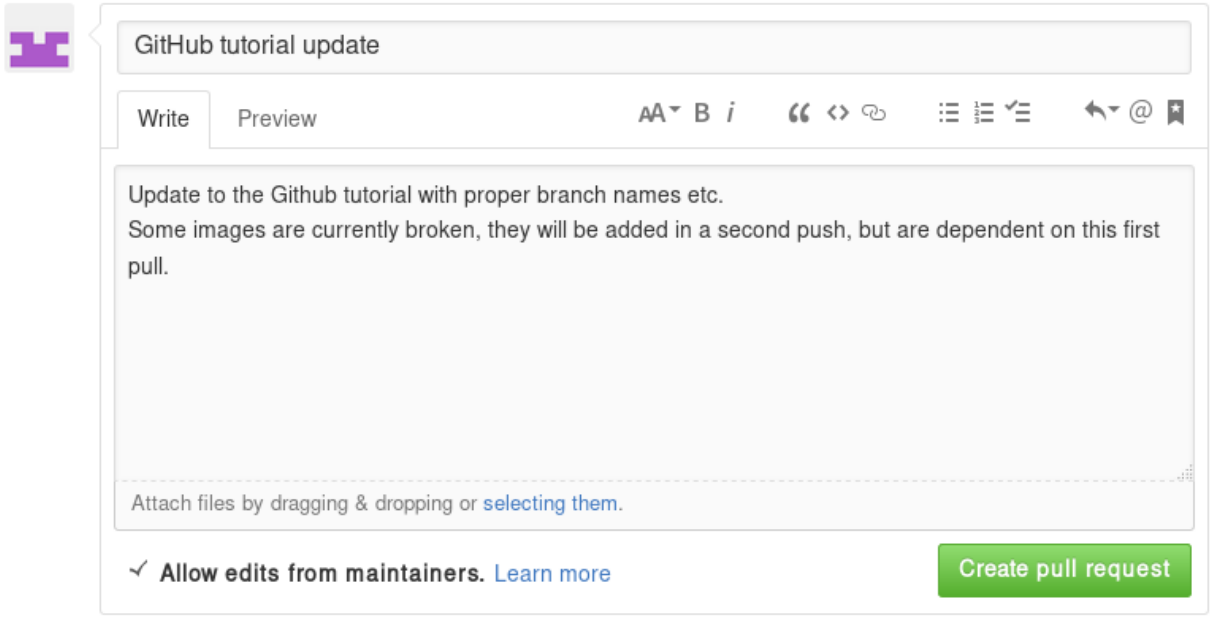
Before creating the pull request, make sure the short title is accurate and add a comment with details about your pull request. Here you write what your modifications do and why they should be incorporated upstream.

Note the checkbox that says "Allow edits from maintainers". This is checked by default checkbox (although in my version of Firefox, only the checkmark is visible):



If it is checked, maintainers can immediately add their own edits to the pull request. This helps the inclusion of your branch significantly, as simple/trivial changes can be added directly to your pull request branch by the LAMMPS maintainers. The alternative would be that they make changes on their own version of the branch and file a reverse pull request to you. Just leave this box checked unless you have a very good reason not to.

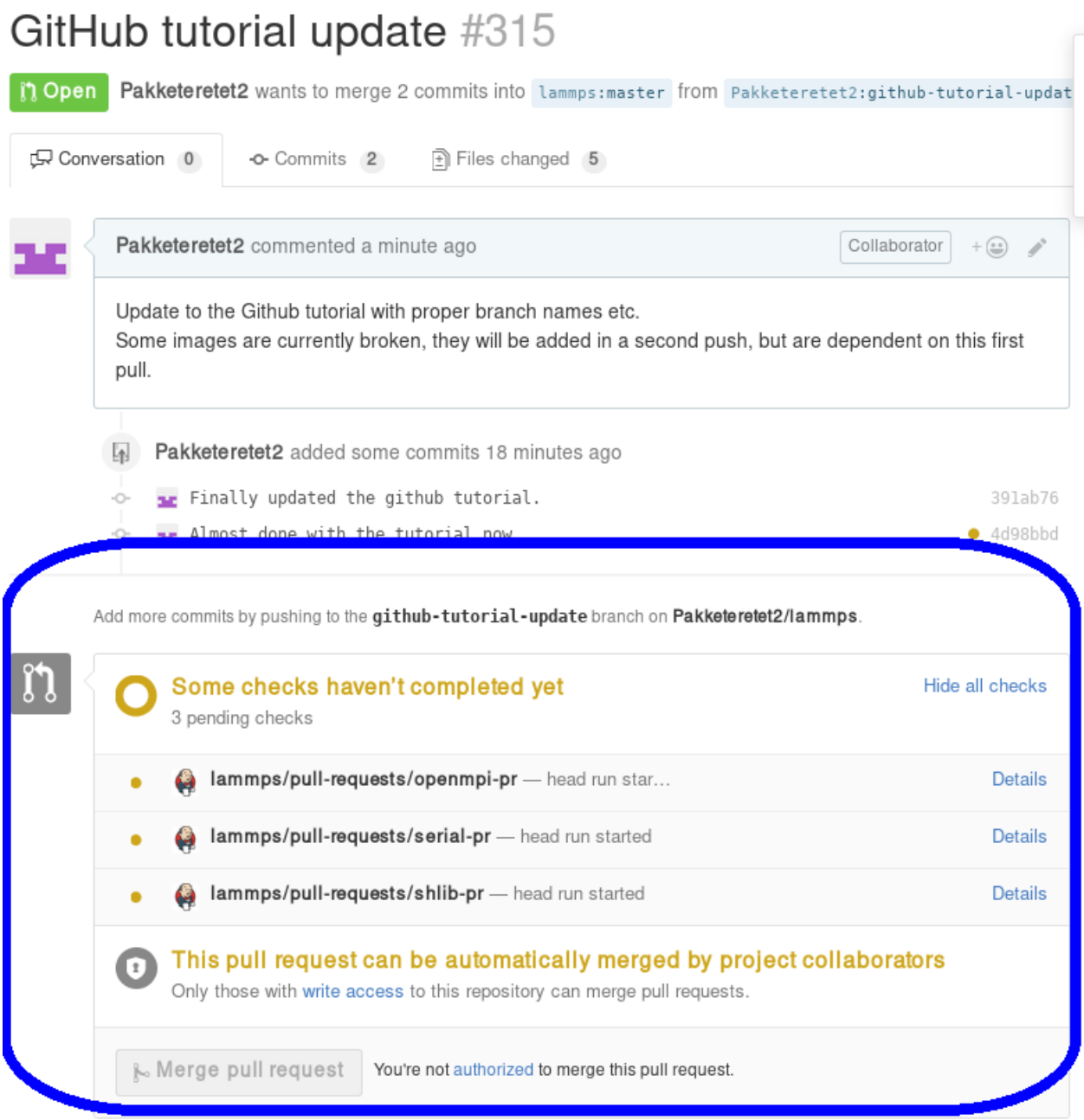
Now just write some nice comments and click on "Create pull request".



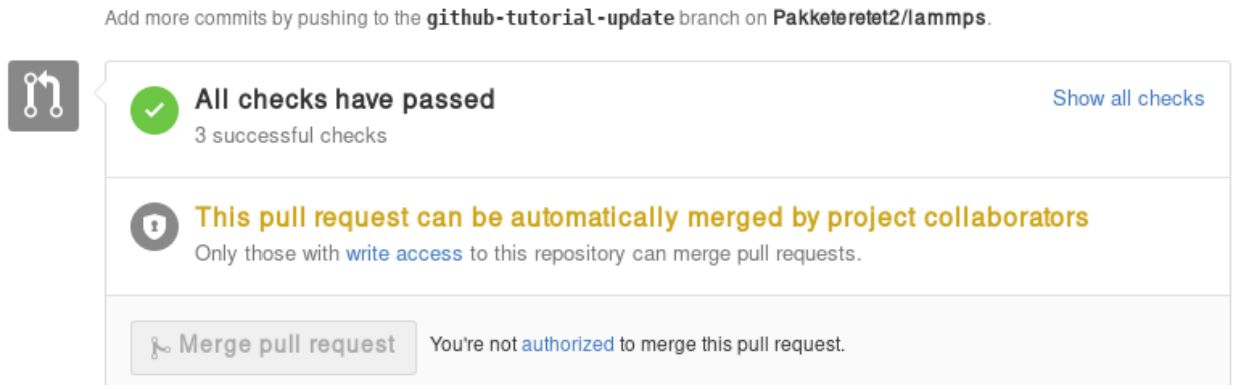
After filing a pull request

> **Note**
>
> When you submit a pull request (or ask for a pull request) for the first time, you will receive an invitation to become a LAMMPS project collaborator. Please accept this invite as being a collaborator will simplify certain administrative tasks and will probably speed up the merging of your feature, too.

You will notice that after filing the pull request, some checks are performed automatically:



If all is fine, you will see this:

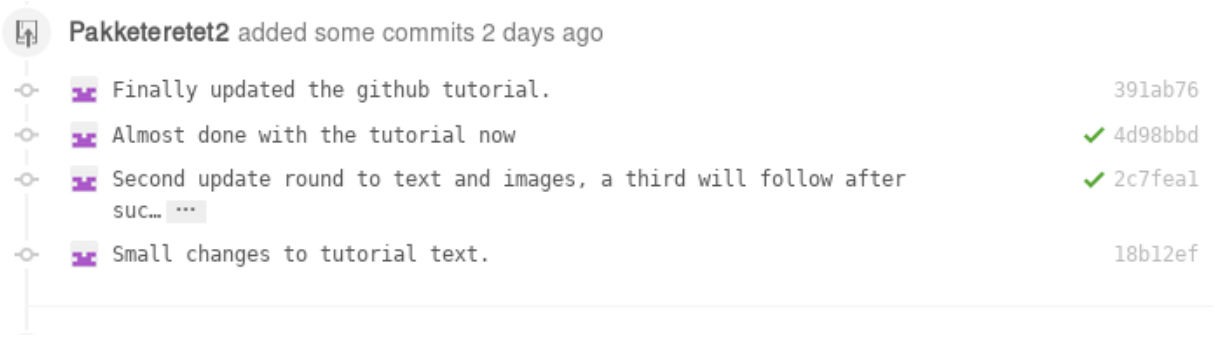


If any of the checks are failing, your pull request will not be processed, as your changes may break compilation for certain configurations or may not merge cleanly. It is your responsibility to remove the reason(s) for the failed test(s). If you need help with this, please contact the LAMMPS developers by adding a comment explaining your problems with resolving the failed tests.

A few further interesting things (can) happen to pull requests before they are included.

Additional changes

First of all, any additional changes you push into your branch in your repository will automatically become part of the pull request:



This means you can add changes that should be part of the feature after filing the pull request, which is useful in case you have forgotten them, or if a developer has requested that something needs to be changed before the feature can be accepted into the official LAMMPS version. After each push, the automated checks are run again.

Labels

LAMMPS developers may add labels to your pull request to assign it to categories (mostly for bookkeeping purposes), but a few of them are important: *needs_work*, *work_in_progress*, *run_tests*, *test_for_regression*, and *ready_for_merge*. The first two indicate, that your pull request is not considered to be complete. With "needs_work" the burden is on exclusively on you; while "work_in_progress" can also mean, that a LAMMPS developer may want to add changes. Please watch the comments to the pull requests. The two "test" labels are used to trigger extended tests before the code is merged. This is sometimes done by LAMMPS developers, if they suspect that there may be some subtle side effects from your changes. It is not done by default, because those tests are very time-consuming. The *ready_for_merge* label is usually attached when the LAMMPS developer assigned to the pull request considers this request complete and to trigger a final full test evaluation.

Reviews

As of Fall 2021, a pull request needs to pass all automatic tests and at least 1 approving review from a LAMMPS developer with write access to the repository before it is eligible for merging. In case your changes touch code that certain developers are associated with, they are auto-requested by the GitHub software. Those associations are set in the file .github/CODEOWNERS Thus if you want to be automatically notified to review when anybody changes files or

packages, that **you** have contributed to LAMMPS, you can add suitable patterns to that file, or a LAMMPS developer may add you.

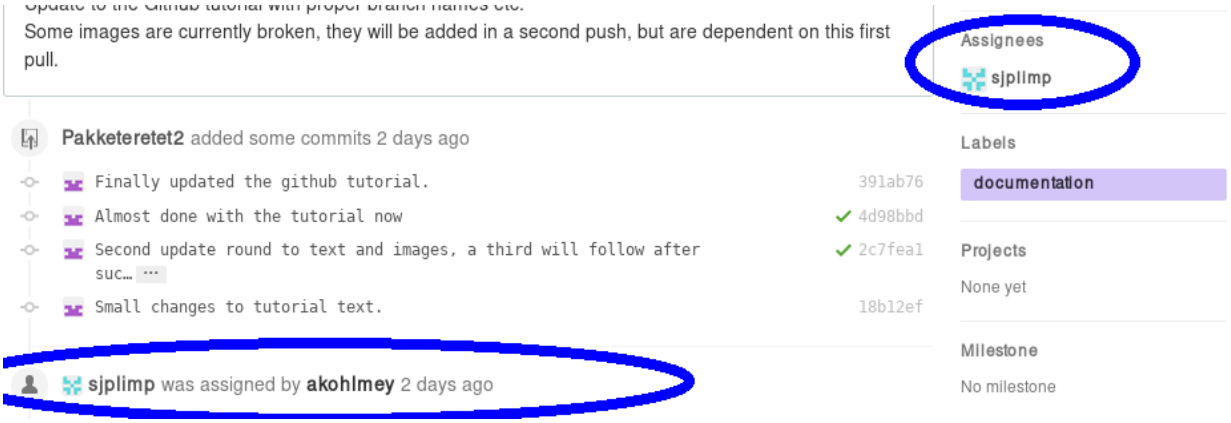
Otherwise, you can also manually request reviews from specific developers, or LAMMPS developers - in their assessment of your pull request - may determine who else should be reviewing your contribution and add that person. Through reviews, LAMMPS developers also may request specific changes from you. If those are not addressed, your pull requests cannot be merged.

Assignees

There is an assignee property for pull requests. If the request has not been reviewed by any developer yet, it is not assigned to anyone. After revision, a developer can choose to assign it to either a) you, b) a LAMMPS developer (including him/herself) or c) Axel Kohlmeyer (akohlmey).

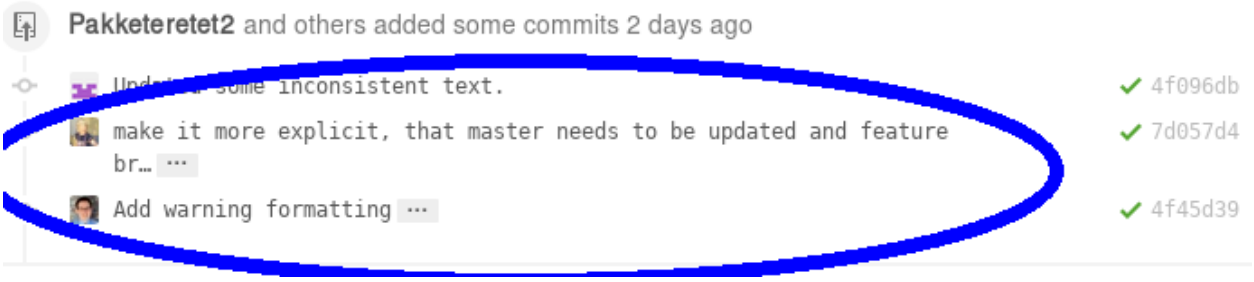
- Case a) happens if changes are required on your part
- Case b) means that at the moment, it is being tested and reviewed by a LAMMPS developer with the expectation that some changes would be required. After the review, the developer can choose to implement changes directly or suggest them to you.
- Case c) means that the pull request has been assigned to the developer overseeing the merging of pull requests into the *develop* branch.

In this case, Axel assigned the tutorial to Steve:



Edits from LAMMPS maintainers

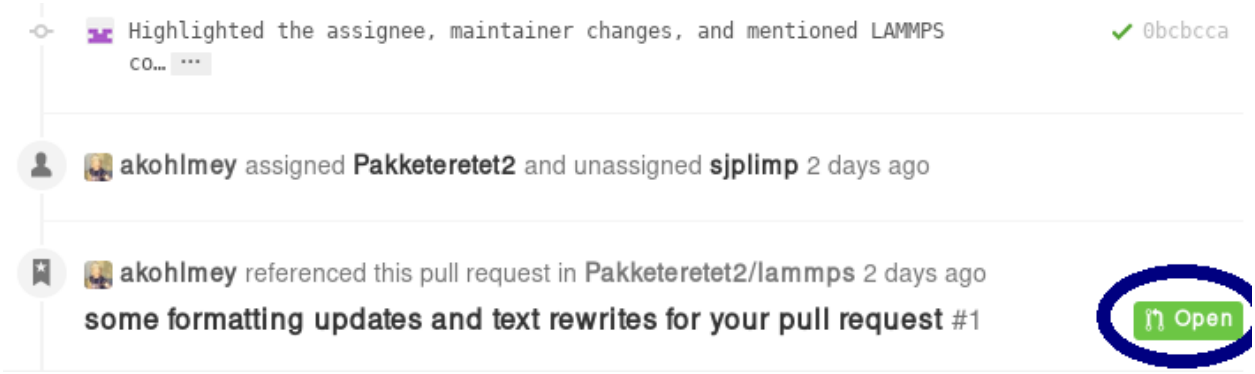
If you allowed edits from maintainers (the default), any LAMMPS maintainer can add changes to your pull request. In this case, both Axel and Richard made changes to the tutorial:



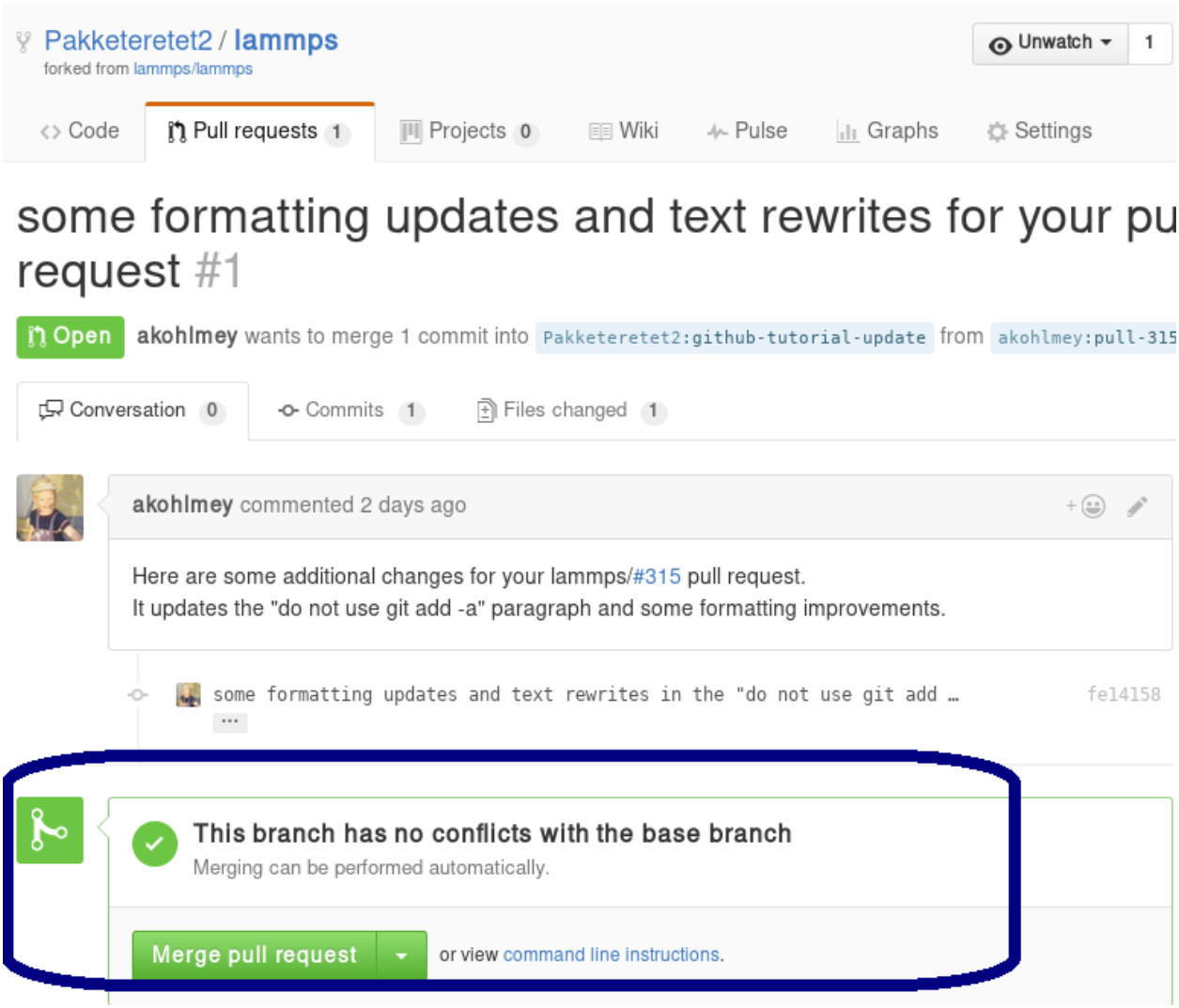
Reverse pull requests

Sometimes, however, you might not feel comfortable having other people push changes into your own branch, or maybe the maintainers are not sure their idea was the right one. In such a case, they can make changes, reassign you as the assignee, and file a "reverse pull request", i.e. file a pull request in **your** forked GitHub repository to include changes in the branch, that you have submitted as a pull request yourself. In that case, you can choose to merge their changes

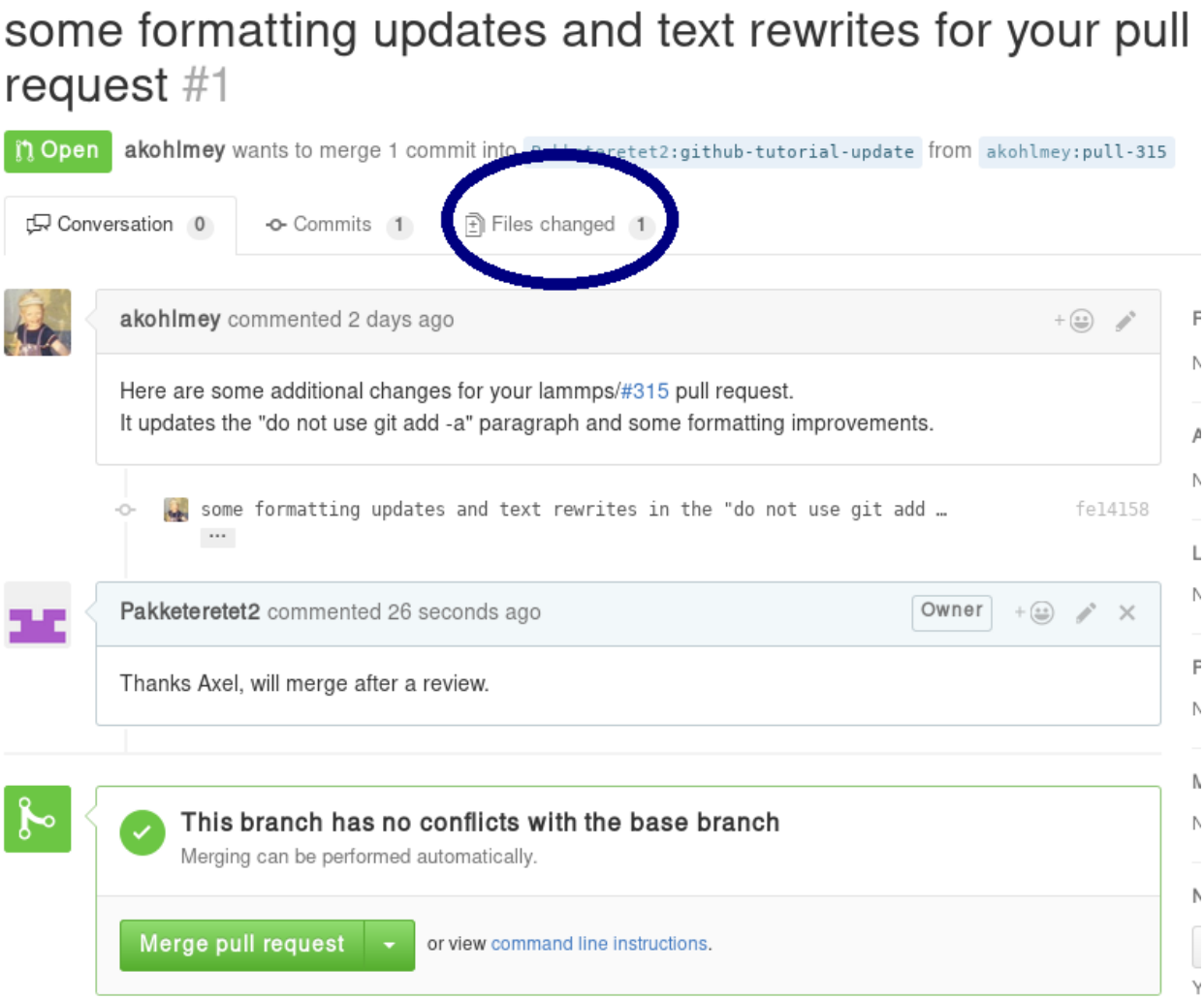
back into your branch, possibly make additional changes or corrections and proceed from there. It looks something like this:



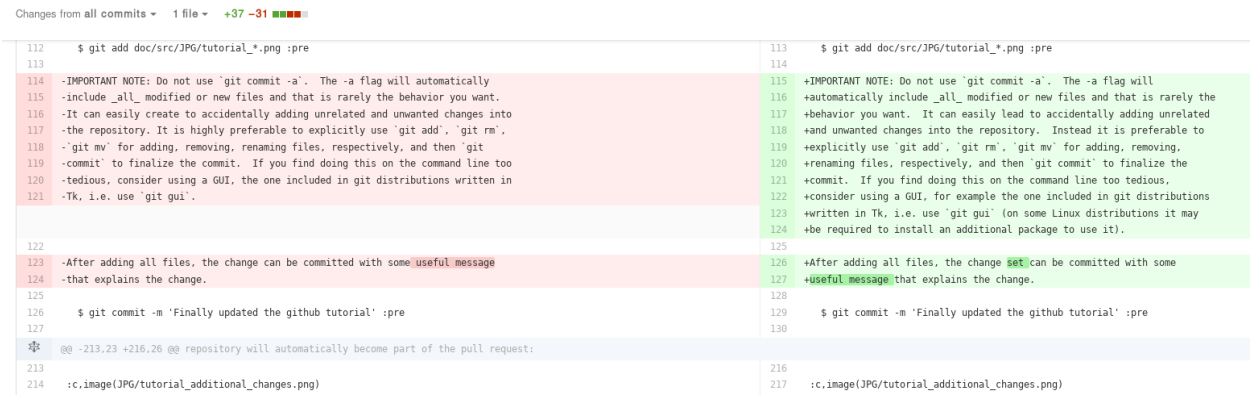
For some reason, the highlighted button did not work in my case, but I can go to my own repository and merge the pull request from there:



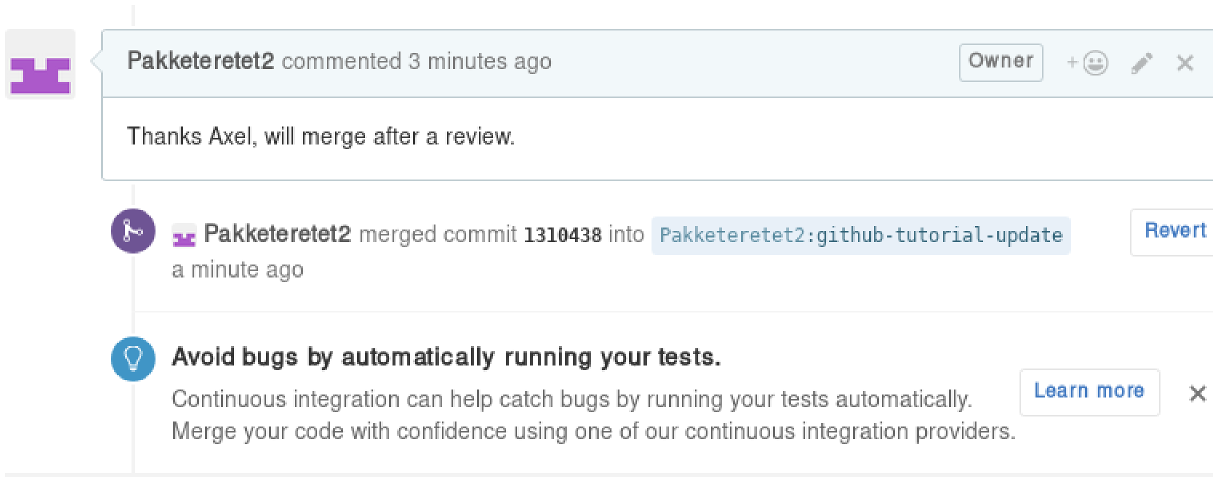
Be sure to check the changes to see if you agree with them by clicking on the tab button:



In this case, most of it is changes in the markup and a short rewrite of Axel's explanation of the "git gui" and "git add" commands.



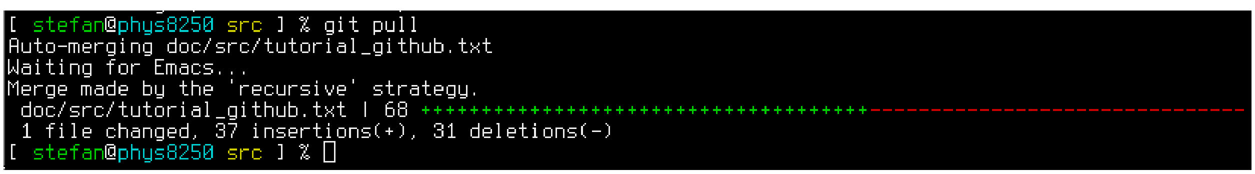
Because the changes are OK with us, we are going to merge by clicking on "Merge pull request". After a merge it looks like this:



Now, since in the meantime our local text for the tutorial also changed, we need to pull Axel's change back into our branch, and merge them:

```
git add Howto_github.txt
git add JPG/tutorial_reverse_pull_request*.png
git commit -m "Updated text and images on reverse pull requests"
git pull
```

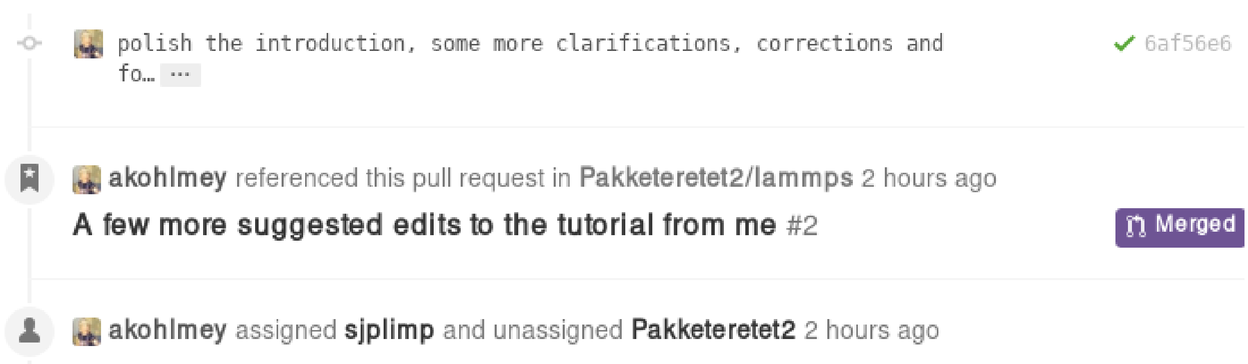
In this case, the merge was painless because git could auto-merge:



With Axel's changes merged in and some final text updates, our feature branch is now perfect as far as we are concerned, so we are going to commit and push again:

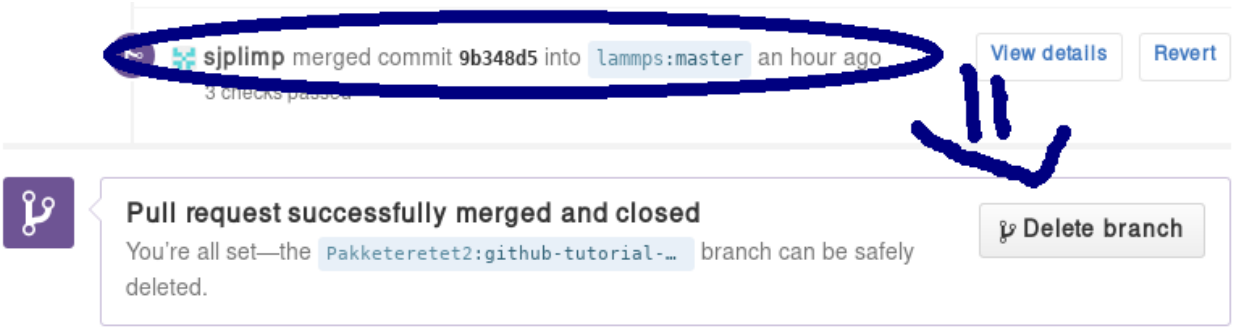
```
git add Howto_github.txt
git add JPG/tutorial_reverse_pull_request6.png
git commit -m "Merged Axel's suggestions and updated text"
git push git@github.com:Pakketeretet2/lammps
```

This merge also shows up on the lammps GitHub page:



After a merge

When everything is fine, the feature branch is merged into the *develop* branch:



Now one question remains: What to do with the feature branch that got merged into upstream?

It is in principle safe to delete them from your own fork. This helps keep it a bit more tidy. Note that you first have to switch to another branch!

```
git checkout develop
git pull https://github.com/lammps/lammps develop
git branch -d github-tutorial-update
```

If you do not pull first, it is not really a problem but git will warn you at the next statement that you are deleting a local branch that was not yet fully merged into HEAD. This is because git does not yet know your branch just got merged into LAMMPS upstream. If you first delete and then pull, everything should still be fine. You can display all branches that are fully merged by:

Finally, if you delete the branch locally, you might want to push this to your remote(s) as well:

```
git push origin :github-tutorial-update
```

Recent changes in the workflow

Some recent changes to the workflow are not captured in this tutorial. For example, in addition to the *develop* branch, to which all new features should be submitted, there is also a *release*, a *stable*, and a *maintenance* branch; the *release* branch is updated from the *develop* branch as part of a "feature release", and *stable* (together with *release*) are updated from *develop* when a "stable release" is made. In between stable releases, selected bug fixes and infrastructure updates are back-ported from the *develop* branch to the *maintenance* branch and occasionally merged to *stable* as an update release.

Furthermore, the naming of the release tags now follow the pattern "patch_<Day><Month><Year>" to simplify comparisons between releases. For stable releases additional "stable_<Day><Month><Year>" tags are applied and update releases are tagged with "stable_<Day><Month><Year>_update<Number>", Finally, all releases and submissions are subject to automatic testing and code checks to make sure they compile with a variety of compilers and popular operating systems. Some unit and regression testing is applied as well.

A detailed discussion of the LAMMPS developer GitHub workflow can be found in the file doc/github-development-workflow.md

8.6.3 Using LAMMPS-GUI

This document describes **LAMMPS-GUI version 1.6**.

LAMMPS-GUI is a graphical text editor customized for editing LAMMPS input files that is linked to the *LAMMPS library* and thus can run LAMMPS directly using the contents of the editor's text buffer as input. It can retrieve and display information from LAMMPS while it is running, display visualizations created with the *dump image command*, and is adapted specifically for editing LAMMPS input files through text completion and reformatting, and linking to the online LAMMPS documentation for known LAMMPS commands and styles.

> **Note**
>
> Pre-compiled, ready-to-use LAMMPS-GUI executables for Linux x86_64 (Ubuntu 20.04LTS or later and compatible), macOS (version 11 aka Big Sur or later), and Windows (version 10 or later) *are available* for download. Non-MPI LAMMPS executables (as lmp) for running LAMMPS from the command line and *some LAMMPS tools* compiled executables are also included.
>
> The source code for LAMMPS-GUI is included in the LAMMPS source code distribution and can be found in the tools/lammps-gui folder. It can be compiled alongside LAMMPS when *compiling with CMake*.

LAMMPS-GUI tries to provide an experience similar to what people traditionally would have running LAMMPS using a command line window and the console LAMMPS executable but just rolled into a single executable:

- writing & editing LAMMPS input files with a text editor
- run LAMMPS on those input file with selected command line flags
- extract data from the created files and visualize it with and external software

That procedure is quite effective for people proficient in using the command line, as that allows them to use tools for the individual steps that they are most comfortable with. In fact, it is often *required* to adopt this workflow when running LAMMPS simulations on high-performance computing facilities.

The main benefit of using LAMMPS-GUI is that many basic tasks can be done directly from the GUI **without** switching to a text console window or using external programs, let alone writing scripts to extract data from the generated output. It also integrates well with graphical desktop environments where the *.lmp* filename extension can be registered with LAMMPS-GUI as the executable to launch when double clicking on such files. Also, LAMMPS-GUI has support for drag-n-drop, i.e. an input file can be selected and then moved and dropped on the LAMMPS-GUI executable, and LAMMPS-GUI will launch and read the file into its buffer. In many cases LAMMPS-GUI will be integrated into the graphical desktop environment and can be launched like other applications.

LAMMPS-GUI thus makes it easier for beginners to get started running simple LAMMPS simulations. It is very suitable for tutorials on LAMMPS since you only need to learn how to use a single program for most tasks and thus time can be saved and people can focus on learning LAMMPS. The tutorials at https://lammpstutorials.github.io/ are specifically updated for use with LAMMPS-GUI.

Another design goal is to keep the barrier low when replacing part of the functionality of LAMMPS-GUI with external tools. That said, LAMMPS-GUI has some unique functionality that is not found elsewhere:

- auto-adapting to features available in the integrated LAMMPS library
- interactive visualization using the *dump image* command with the option to copy-paste the resulting settings
- automatic slide show generation from dump image out at runtime
- automatic plotting of thermodynamics data at runtime
- inspection of binary restart files

The following text provides a detailed tour of the features and functionality of LAMMPS-GUI. Suggestions for new features and reports of bugs are always welcome. You can use the *the same channels as for LAMMPS itself* for that purpose.

Installing Pre-compiled LAMMPS-GUI Packages

LAMMPS-GUI is available as pre-compiled binary packages for Linux x86_64, macOS 11 and later, and Windows 10 and later. Alternately, it can be compiled from source.

Windows 10 and later

After downloading the LAMMPS-Win10-64bit-GUI-<version>.exe installer package, you need to execute it, and start the installation process. Since those packages are currently unsigned, you have to enable "Developer Mode" in the Windows System Settings to run the installer.

MacOS 11 and later

After downloading the LAMMPS-macOS-multiarch-GUI-<version>.dmg installer package, you need to double-click it and then, in the window that opens, drag the app bundle as indicated into the "Applications" folder. The follow the instructions in the "README.txt" file to get access to the other included executables.

Linux on x86_64

For Linux with x86_64 CPU there are currently two variants. The first is compiled on Ubuntu 20.04LTS, is using some wrapper scripts, and should be compatible with more recent Linux distributions. After downloading and unpacking the LAMMPS-Linux-x86_64-GUI-<version>.tar.gz package. You can switch into the "LAMMPS_GUI" folder and execute "./lammps-gui" directly.

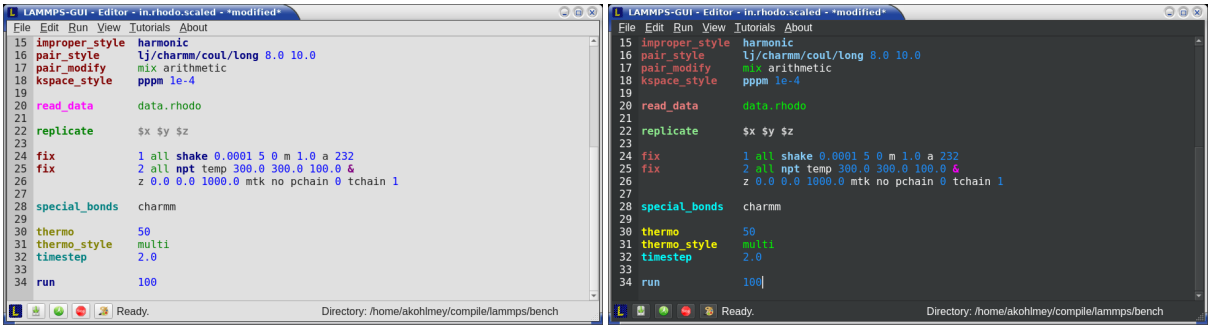
The second variant uses flatpak and requires the flatpak management and runtime software to be installed. After downloading the LAMMPS-GUI-Linux-x86_64-GUI-<version>.tar.gz flatpak bundle, you can install it with flatpak install --user LAMMPS-GUI-Linux-x86_64-GUI-<version>.tar.gz. After installation, LAMMPS-GUI should be integrated into your desktop environment under "Applications > Science" but also can be launched from the console with flatpak run org.lammps.lammps-gui. The flatpak bundle also includes the console LAMMPS executable lmp which can be launched to run simulations with, for example: flatpak run --command=lmp org.lammps.lammps-gui -in in.melt.

Compiling from Source

There also are instructions for *compiling LAMMPS-GUI from source code* available elsewhere in the manual. Compilation from source *requires* using CMake.

Starting LAMMPS-GUI

When LAMMPS-GUI starts, it shows the main window, labeled *Editor*, with either an empty buffer or the contents of the file used as argument. In the latter case it may look like the following:



There is the typical menu bar at the top, then the main editor buffer, and a status bar at the bottom. The input file contents are shown with line numbers on the left and the input is colored according to the LAMMPS input file syntax. The status bar shows the status of LAMMPS execution on the left (e.g. "Ready." when idle) and the current working directory on the right. The name of the current file in the buffer is shown in the window title; the word **modified** is added if the buffer edits have not yet saved to a file. The geometry of the main window is stored when exiting and restored when starting again.

Opening Files

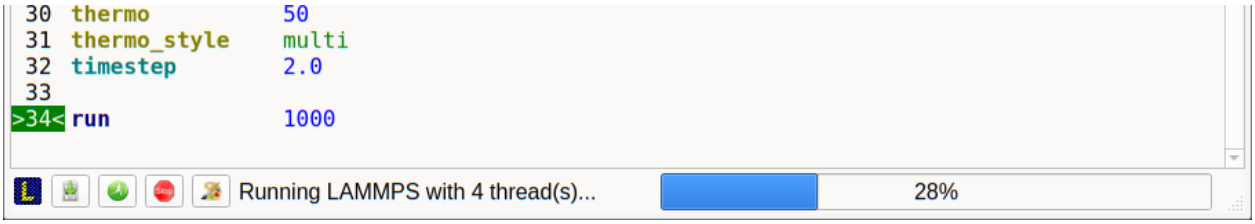
The LAMMPS-GUI application can be launched without command line arguments and then starts with an empty buffer in the *Editor* window. If arguments are given LAMMPS will use first command line argument as the file name for the *Editor* buffer and reads its contents into the buffer, if the file exists. All further arguments are ignored. Files can also be opened via the *File* menu, the *Ctrl-O* (*Command-O* on macOS) keyboard shortcut or by drag-and-drop of a file from a graphical file manager into the editor window. If a file extension (e.g. .lmp) has been registered with the graphical environment to launch LAMMPS-GUI, an existing input file can be launched with LAMMPS-GUI through double clicking.

Only one file can be edited at a time, so opening a new file with a filled buffer closes that buffer. If the buffer has unsaved modifications, you are asked to either cancel the operation, discard the changes, or save them. A buffer with modifications can be saved any time from the "File" menu, by the keyboard shortcut *Ctrl-S* (*Command-S* on macOS), or by clicking on the "Save" button at the very left in the status bar.

Running LAMMPS

From within the LAMMPS-GUI main window LAMMPS can be started either from the *Run* menu using the *Run LAMMPS from Editor Buffer* entry, by the keyboard shortcut *Ctrl-Enter* (*Command-Enter* on macOS), or by clicking on the green "Run" button in the status bar. All of these operations causes LAMMPS to process the entire input script in the editor buffer, which may contain multiple *run* or *minimize* commands.

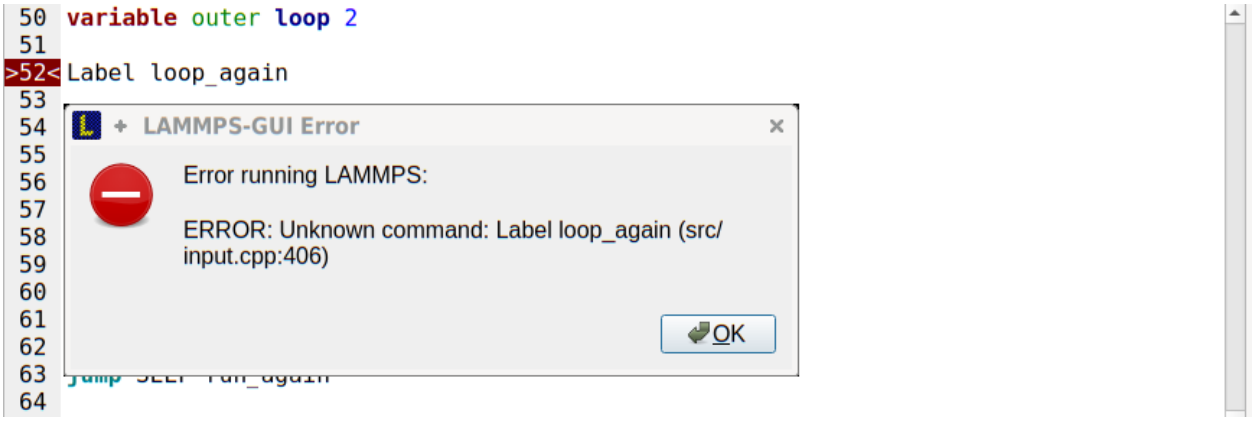
LAMMPS runs in a separate thread, so the GUI stays responsive and is able to interact with the running calculation and access data it produces. It is important to note that running LAMMPS this way is using the contents of the input buffer for the run (via the lammps_commands_string() function of the LAMMPS C-library interface), and **not** the original file it was read from. Thus, if there are unsaved changes in the buffer, they *will* be used. As an alternative, it is also possible to run LAMMPS by reading the contents of a file from the *Run LAMMPS from File* menu entry or with *Ctrl-Shift-Enter*. This option may be required in some rare cases where the input uses some functionality that is not compatible with running LAMMPS from a string buffer. For consistency, any unsaved changes in the buffer must be either saved to the file or undone before LAMMPS can be run from a file.



While LAMMPS is running, the contents of the status bar change. On the left side there is a text indicating that LAMMPS is running, which also indicates the number of active threads, when thread-parallel acceleration was selected in the *Preferences* dialog. On the right side, a progress bar is shown that displays the estimated progress for the current *run* or *minimize* command.

Also, the line number of the currently executed command is highlighted in green.

If an error occurs (in the example below the command *label* was incorrectly capitalized as "Label"), an error message dialog is shown and the line of the input which triggered the error is highlighted. The state of LAMMPS in the status bar is set to "Failed." instead of "Ready."



Up to three additional windows may open during a run:

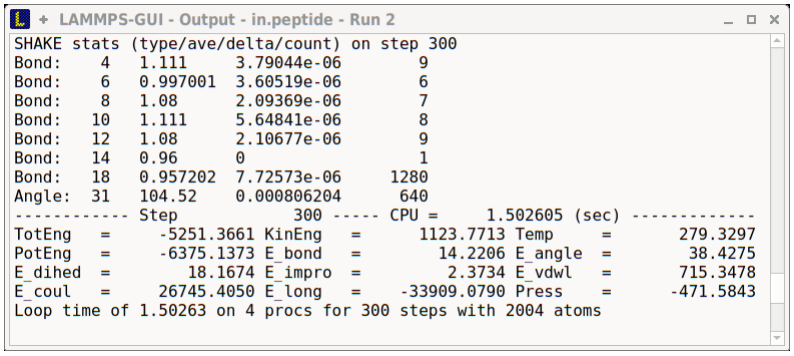
- an *Output* window with the captured screen output from LAMMPS
- a *Charts* window with a line graph created from thermodynamic output of the run
- a *Slide Show* window with images created by a *dump image command* in the input

More information on those windows and how to adjust their behavior and contents is given below.

An active LAMMPS run can be stopped cleanly by using either the *Stop LAMMPS* entry in the *Run* menu, the keyboard shortcut *Ctrl-/* (*Command-/* on macOS), or by clicking on the red button in the status bar. This will cause the running LAMMPS process to complete the current timestep (or iteration for energy minimization) and then complete the processing of the buffer while skipping all run or minimize commands. This is equivalent to the input script command *timer timeout 0* and is implemented by calling the lammps_force_timeout() function of the LAMMPS C-library interface. Please see the corresponding documentation pages to understand the implications of this operation.

Output Window

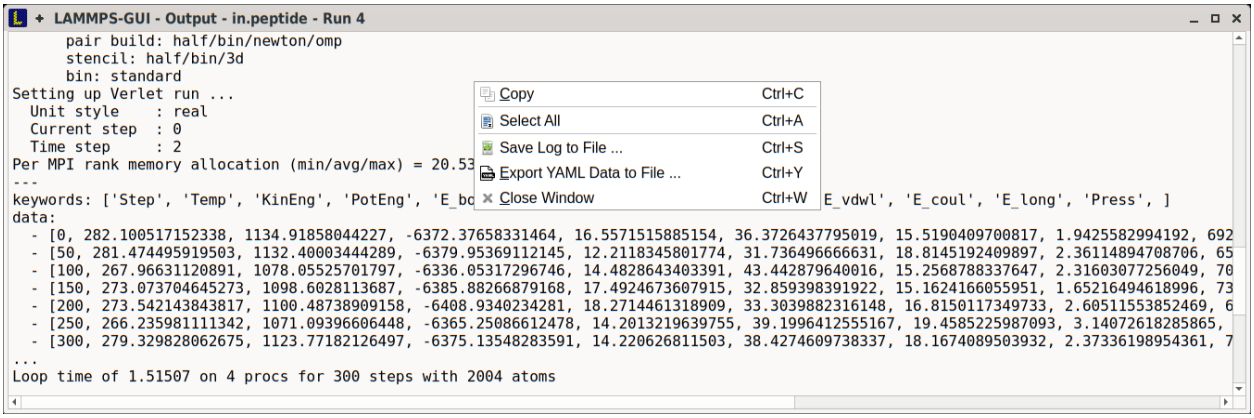
By default, when starting a run, an *Output* window opens that displays the screen output of the running LAMMPS calculation, as shown below. This text would normally be seen in the command line window.



LAMMPS-GUI captures the screen output from LAMMPS as it is generated and updates the *Output* window regularly during a run.

By default, the *Output* window is replaced each time a run is started. The runs are counted and the run number for the current run is displayed in the window title. It is possible to change the behavior of LAMMPS-GUI in the preferences dialog to create a *new Output* window for every run or to not show the current *Output* window. It is also possible to show or hide the *current Output* window from the *View* menu.

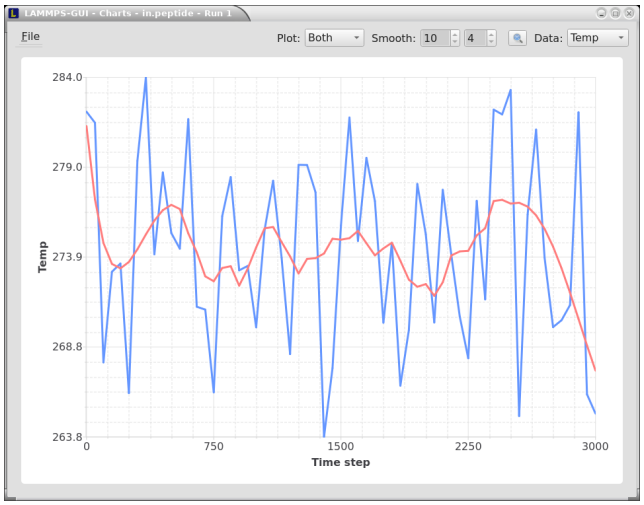
The text in the *Output* window is read-only and cannot be modified, but keyboard shortcuts to select and copy all or parts of the text can be used to transfer text to another program. Also, the keyboard shortcut *Ctrl-S* (*Command-S* on macOS) is available to save the *Output* buffer to a file. The "Select All" and "Copy" functions, as well as a "Save Log to File" option are also available from a context menu by clicking with the right mouse button into the *Output* window text area.



Should the *Output* window contain embedded YAML format text (see above for a demonstration), for example from using *thermo_style yaml* or *thermo_modify line yaml*, the keyboard shortcut *Ctrl-Y* (*Command-Y* on macOS) is available to save only the YAML parts to a file. This option is also available from a context menu by clicking with the right mouse button into the *Output* window text area.

Charts Window

By default, when starting a run, a *Charts* window opens that displays a plot of thermodynamic output of the LAMMPS calculation as shown below.



The drop down menu on the top right allows selection of different properties that are computed and written to thermo output. Only one property can be shown at a time. The plots are updated regularly with new data as the run progresses, so they can be used to visually monitor the evolution of available properties. The update interval can be set in the *Preferences* dialog. By default, the raw data for the selected property is plotted as a blue graph. As soon as there are a sufficient number of data points, there will be a second graph shown in red with a smoothed version of the data. From the drop down menu on the top left, you can select whether to plot only the raw data, only the smoothed data or both. The smoothing uses a Savitzky-Golay convolution filter The window width (left) and order (right) parameters can be set in the boxes next to the drop down menu. Default settings are 10 and 4 which means that the smoothing window includes 10 points each to the left and the right of the current data point and a fourth order polynomial is fit to the data in the window.

You can use the mouse to zoom into the graph (hold the left button and drag to mark an area) or zoom out (right click) and you can reset the view with a click to the "lens" button next to the data drop down menu.

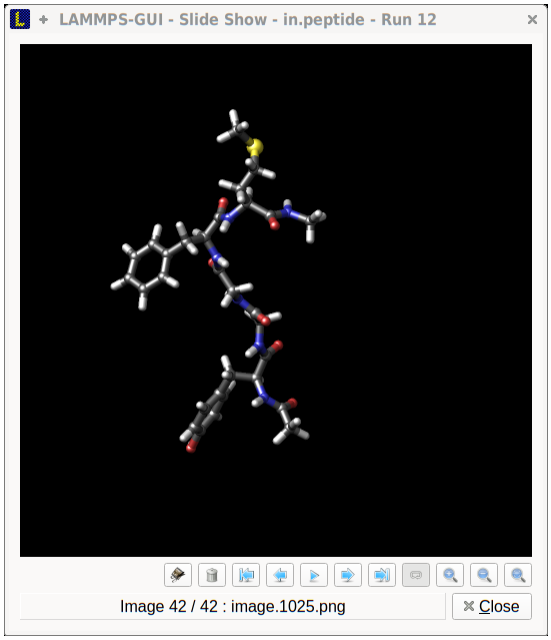
The window title shows the current run number that this chart window corresponds to. Same as for the *Output* window, the chart window is replaced on each new run, but the behavior can be changed in the *Preferences* dialog.

From the *File* menu on the top left, it is possible to save an image of the currently displayed plot or export the data in either plain text columns (for use by plotting tools like gnuplot or grace), as CSV data which can be imported for further processing with Microsoft Excel LibreOffice Calc or with Python via pandas, or as YAML which can be imported into Python with PyYAML or pandas.

Thermo output data from successive run commands in the input script is combined into a single data set unless the format, number, or names of output columns are changed with a *thermo_style* or a *thermo_modify* command, or the current time step is reset with *reset_timestep*, or if a *clear* command is issued. This is where the YAML export from the *Charts* window differs from that of the *Output* window: here you get the compounded data set starting with the last change of output fields or timestep setting, while the export from the log will contain *all* YAML output but *segmented* into individual runs.

Image Slide Show

By default, if the LAMMPS input contains a *dump image* command, a "Slide Show" window opens which loads and displays the images created by LAMMPS as they are written. This is a convenient way to visually monitor the progress of the simulation.

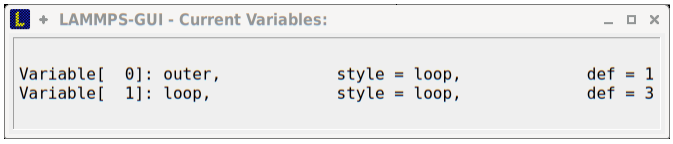


The various buttons at the bottom right of the window allow single stepping through the sequence of images or playing an animation (as a continuous loop or once from first to last). It is also possible to zoom in or zoom out of the displayed images. The button on the very left triggers an export of the slide show animation to a movie file, provided the FFmpeg program is installed.

When clicking on the "garbage can" icon, all image files of the slide show will be deleted. Since their number can be large for long simulations, this option enables to safely and quickly clean up the clutter caused in the working directory by those image files without risk of deleting other files by accident when using wildcards.

Variable Info

During a run, it may be of interest to monitor the value of input script variables, for example to monitor the progress of loops. This can be done by enabling the "Variables Window" in the *View* menu or by using the *Ctrl-Shift-W* keyboard shortcut. This shows info similar to the *info variables* command in a separate window as shown below.



Like for the *Output* and *Charts* windows, its content is continuously updated during a run. It will show "(none)" if there are no variables defined. Note that it is also possible to *set index style variables*, that would normally be set via command line flags, via the "Set Variables…" dialog from the *Run* menu. LAMMPS-GUI automatically defines the variable "gui_run" to the current value of the run counter. That way it is possible to automatically record a separate log for each run attempt by using the command

```
log logfile-${gui_run}.txt
```

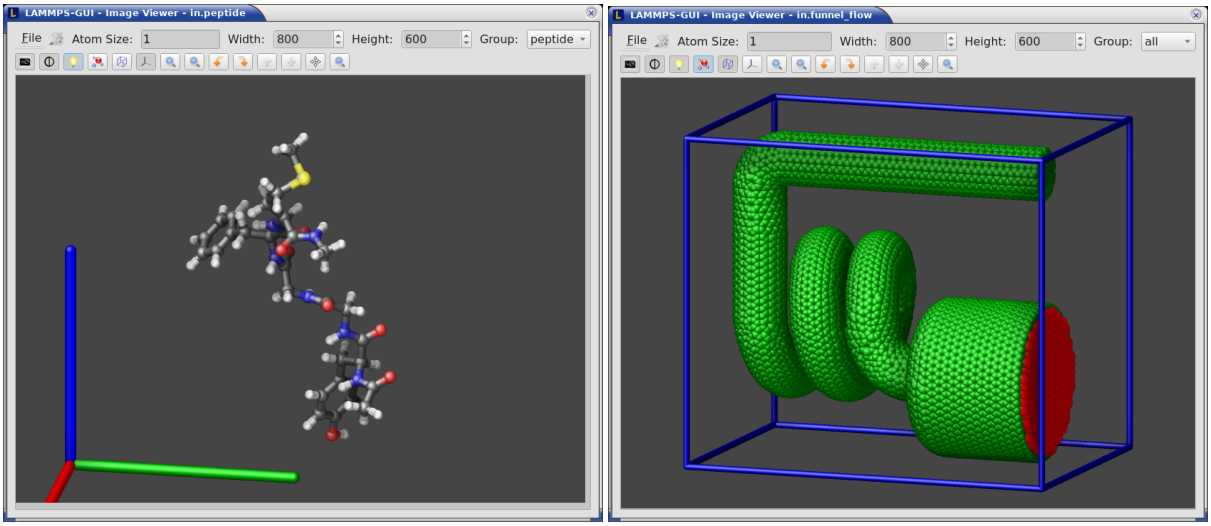
at the beginning of an input file. That would record logs to files logfile-1.txt, logfile-2.txt, and so on for successive runs.

Snapshot Image Viewer

By selecting the *Create Image* entry in the *Run* menu, or by hitting the *Ctrl-I* (*Command-I* on macOS) keyboard shortcut, or by clicking on the "palette" button in the status bar of the *Editor* window, LAMMPS-GUI sends a custom *write_dump image* command to LAMMPS and reads back the resulting snapshot image with the current state of the system into an image viewer. This functionality is *not* available *during* an ongoing run. In case LAMMPS is not yet initialized, LAMMPS-GUI tries to identify the line with the first run or minimize command and execute all commands from the input buffer up to that line, and then executes a "run 0" command. This initializes the system so an image of the initial state of the system can be rendered. If there was an error in that process, the snapshot image viewer does not appear.

When possible, LAMMPS-GUI tries to detect which elements the atoms correspond to (via their mass) and then colorize them in the image and set their atom diameters accordingly. If this is not possible, for instance when using reduced (= 'lj') *units*, then LAMMPS-GUI will check the current pair style and if it is a Lennard-Jones type potential, it will extract the *sigma* parameter for each atom type and assign atom diameters from those numbers. For cases where atom diameters are not auto-detected, the *Atom size* field can be edited and a suitable value set manually. The default value is inferred from the x-direction lattice spacing.

If elements cannot be detected the default sequence of colors of the *dump image* command is assigned to the different atom types.



The default image size, some default image quality settings, the view style and some colors can be changed in the *Preferences* dialog window. From the image viewer window further adjustments can be made: actual image size, high-quality (SSAO) rendering, anti-aliasing, view style, display of box or axes, zoom factor. The view of the system can be rotated horizontally and vertically. It is also possible to only display the atoms within a group defined in the input script (default is "all"). The image can also be re-centered on the center of mass of the selected group. After each change, the image is rendered again and the display updated. The small palette icon on the top left is colored while LAMMPS is running to render the new image; it is grayed out when LAMMPS is finished. When there are many atoms to render and high quality images with anti-aliasing are requested, re-rendering may take several seconds. From the *File* menu of the image window, the current image can be saved to a file (keyboard shortcut *Ctrl-S*) or copied to the clipboard (keyboard shortcut *Ctrl-C*) for pasting the image into another application.

From the *File* menu it is also possible to copy the current *dump image* and *dump_modify* commands to the clipboard so they can be pasted into a LAMMPS input file so that the visualization settings of the snapshot image can be repeated for the entire simulation (and thus be repeated in the slide show viewer). This feature has the keyboard shortcut *Ctrl-D*.

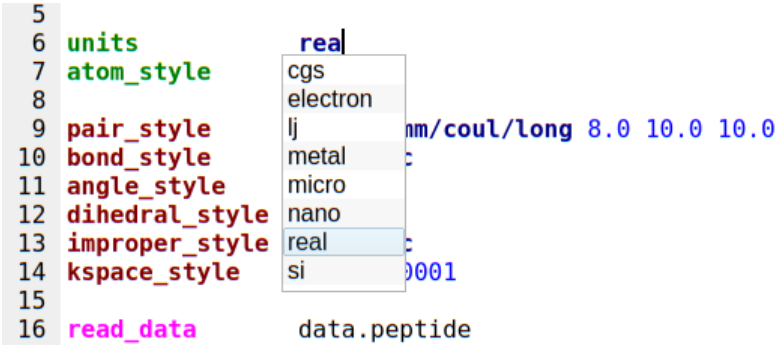
Editor Window

The *Editor* window of LAMMPS-GUI has most of the usual functionality that similar programs have: text selection via mouse or with cursor moves while holding the Shift key, Cut (*Ctrl-X*), Copy (*Ctrl-C*), Paste (*Ctrl-V*), Undo (*Ctrl-Z*), Redo (*Ctrl-Shift-Z*), Select All (*Ctrl-A*). When trying to exit the editor with a modified buffer, a dialog will pop up asking whether to cancel the exit operation, or to save or not save the buffer contents to a file.

The editor has an auto-save mode that can be enabled or disabled in the *Preferences* dialog. In auto-save mode, the editor buffer is automatically saved before running LAMMPS or before exiting LAMMPS-GUI.

Context Specific Word Completion

By default, LAMMPS-GUI displays a small pop-up frame with possible choices for LAMMPS input script commands or styles after 2 characters of a word have been typed.



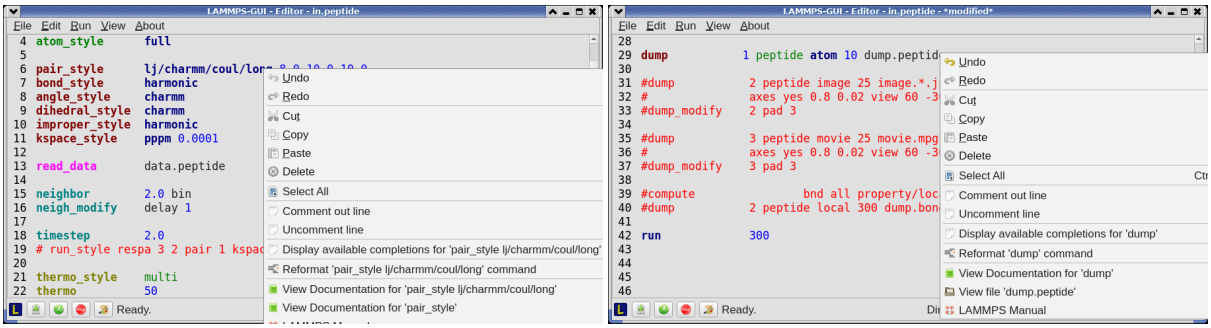
The word can then be completed through selecting an entry by scrolling up and down with the cursor keys and selecting with the 'Enter' key or by clicking on the entry with the mouse. The automatic completion pop-up can be disabled in the *Preferences* dialog, but the completion can still be requested manually by either hitting the 'Shift-TAB' key or by right-clicking with the mouse and selecting the option from the context menu. Most of the completion information is retrieved from the active LAMMPS instance and thus it shows only available options that have been enabled when compiling LAMMPS. That list, however, excludes accelerated styles and commands; for improved clarity, only the non-suffix version of styles are shown.

Line Reformatting

The editor supports reformatting lines according to the syntax in order to have consistently aligned lines. This primarily means adding whitespace padding to commands, type specifiers, IDs and names. This reformatting is performed manually by hitting the 'Tab' key. It is also possible to have this done automatically when hitting the 'Enter' key to start a new line. This feature can be turned on or off in the *Preferences* dialog for *Editor Settings* with the "Reformat with 'Enter'" checkbox. The amount of padding for multiple categories can be adjusted in the same dialog.

Internally this functionality is achieved by splitting the line into "words" and then putting it back together with padding added where the context can be detected; otherwise a single space is used between words.

Context Specific Help



A unique feature of LAMMPS-GUI is the option to look up the LAMMPS documentation for the command in the current line. This can be done by either clicking the right mouse button or by using the *Ctrl-?* keyboard shortcut. When using the mouse, there are additional entries in the context menu that open the corresponding documentation page in the online LAMMPS documentation in a web browser window. When using the keyboard, the first of those entries is chosen.

If the word under the cursor is a file, then additionally the context menu has an entry to open the file in a read-only text viewer window. If the file is a LAMMPS restart file, instead the menu entry offers to *inspect the restart*.

The text viewer is a convenient way to view the contents of files that are referenced in the input. The file viewer also supports on-the-fly decompression based on the file name suffix in a *similar fashion as available with LAMMPS*. If the necessary decompression program is missing or the file cannot be decompressed, the viewer window will contain a corresponding message.

Inspecting a Restart file

When LAMMPS-GUI is asked to "Inspect a Restart", it will read the restart file into a LAMMPS instance and then open three different windows. The first window is a text viewer with the output of an *info command* with system information stored in the restart. The second window is text viewer containing a data file generated with a *write_data command*. The third window is a *Snapshot Image Viewer* containing a visualization of the system in the restart.

If the restart file is larger than 250 MBytes, a dialog will ask for confirmation before continuing, since large restart files may require large amounts of RAM since the entire system must be read into RAM. Thus restart file for large simulations that have been run on an HPC cluster may overload a laptop or local workstation. The *Show Details...* button will display a rough estimate of the additional memory required.

Menu

The menu bar has entries *File*, *Edit*, *Run*, *View*, and *About*. Instead of using the mouse to click on them, the individual menus can also be activated by hitting the *Alt* key together with the corresponding underlined letter, that is *Alt-F* activates the *File* menu. For the corresponding activated sub-menus, the key corresponding the underlined letters can be used to select entries instead of using the mouse.

File

The *File* menu offers the usual options:

- *New* clears the current buffer and resets the file name to *unknown*
- *Open* opens a dialog to select a new file for editing in the *Editor*
- *View* opens a dialog to select a file for viewing in a *separate* window (read-only) with support for on-the-fly decompression as explained above.
- *Inspect restart* opens a dialog to select a file. If that file is a *LAMMPS restart* three windows with *information about the file are opened*.
- *Save* saves the current file; if the file name is *unknown* a dialog will open to select a new file name
- *Save As* opens a dialog to select and new file name (and folder, if desired) and saves the buffer to it. Writing the buffer to a different folder will also switch the current working directory to that folder.
- *Quit* exits LAMMPS-GUI. If there are unsaved changes, a dialog will appear to either cancel the operation, or to save, or to not save the modified buffer.

In addition, up to 5 recent file names will be listed after the *Open* entry that allows re-opening recently opened files. This list is stored when quitting and recovered when starting again.

Edit

The *Edit* menu offers the usual editor functions like *Undo*, *Redo*, *Cut*, *Copy*, *Paste*, and a *Find and Replace* dialog (keyboard shortcut *Ctrl-F*). It can also open a *Preferences* dialog (keyboard shortcut *Ctrl-P*) and allows deleting all stored preferences and settings, so they are reset to their default values.

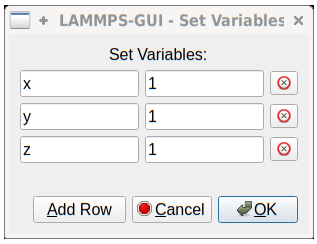
Run

The *Run* menu has options to start and stop a LAMMPS process. Rather than calling the LAMMPS executable as a separate executable, the LAMMPS-GUI is linked to the LAMMPS library and thus can run LAMMPS internally through the *LAMMPS C-library interface* in a separate thread.

Specifically, a LAMMPS instance will be created by calling lammps_open_no_mpi(). The buffer contents are then executed by calling lammps_commands_string(). Certain commands and features are only available after a LAMMPS instance is created. Its presence is indicated by a small LAMMPS L logo in the status bar at the bottom left of the main window. As an alternative, it is also possible to run LAMMPS using the contents of the edited file by reading the file. This is mainly provided as a fallback option in case the input uses some feature that is not available when running from a string buffer.

The LAMMPS calculations are run in a concurrent thread so that the GUI can stay responsive and be updated during the run. The GUI can retrieve data from the running LAMMPS instance and tell it to stop at the next timestep. The *Stop LAMMPS* entry will do this by calling the lammps_force_timeout() library function, which is equivalent to a *timer timeout 0* command.

The *Set Variables...* entry opens a dialog box where *index style variables* can be set. Those variables are passed to the LAMMPS instance when it is created and are thus set *before* a run is started.



The *Set Variables* dialog will be pre-populated with entries that are set as index variables in the input and any variables that are used but not defined, if the built-in parser can detect them. New rows for additional variables can be added through the *Add Row* button and existing rows can be deleted by clicking on the *X* icons on the right.

The *Create Image* entry will send a *dump image* command to the LAMMPS instance, read the resulting file, and show it in an *Image Viewer* window.

The *View in OVITO* entry will launch OVITO with a *data file* containing the current state of the system. This option is only available if LAMMPS-GUI can find the OVITO executable in the system path.

The *View in VMD* entry will launch VMD with a *data file* containing the current state of the system. This option is only available if LAMMPS-GUI can find the VMD executable in the system path.

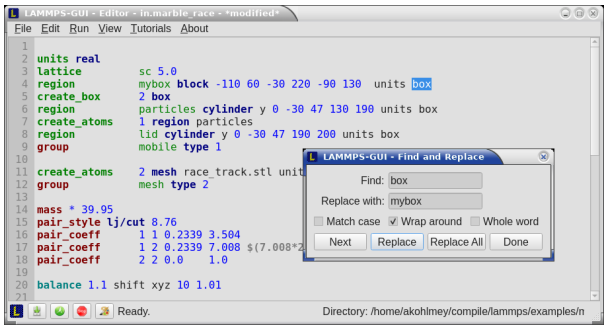
View

The *View* menu offers to show or hide additional windows with log output, charts, slide show, variables, or snapshot images. The default settings for their visibility can be changed in the *Preferences* dialog.

About

The *About* menu finally offers a couple of dialog windows and an option to launch the LAMMPS online documentation in a web browser. The *About LAMMPS-GUI* entry displays a dialog with a summary of the configuration settings of the LAMMPS library in use and the version number of LAMMPS-GUI itself. The *Quick Help* displays a dialog with a minimal description of LAMMPS-GUI. The *LAMMPS-GUI Howto* entry will open this documentation page from the online documentation in a web browser window. The *LAMMPS Manual* entry will open the main page of the LAMMPS online documentation in a web browser window. The *LAMMPS Tutorial* entry will open the main page of the set of LAMMPS tutorials authored and maintained by Simon Gravelle at https://lammpstutorials.github.io/ in a web browser window.

Find and Replace



The *Find and Replace* dialog allows searching for and replacing text in the *Editor* window.

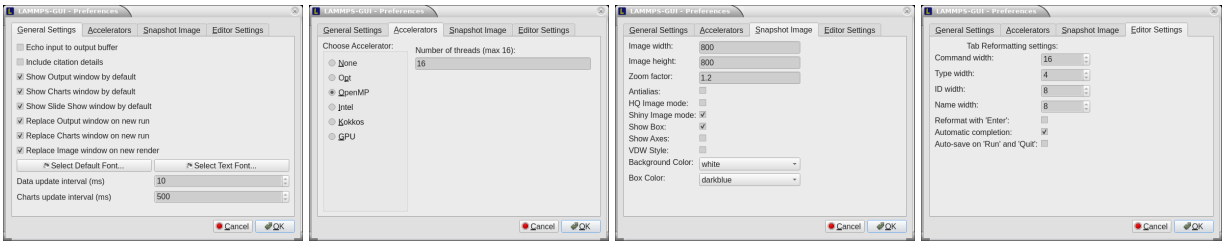
The dialog can be opened either from the *Edit* menu or with the keyboard shortcut *Ctrl-F*. You can enter the text to search for. Through three check-boxes the search behavior can be adjusted:

- If checked, "Match case" does a case sensitive search; otherwise the search is case insensitive.
- If checked, "Wrap around" starts searching from the start of the document, if there is no match found from the current cursor position until the end of the document; otherwise the search will stop.
- If checked, the "Whole word" setting only finds full word matches (white space and special characters are word boundaries).

Clicking on the *Next* button will search for the next occurrence of the search text and select / highlight it. Clicking on the *Replace* button will replace an already highlighted search text and find the next one. If no text is selected, or the selected text does not match the selection string, then the first click on the *Replace* button will only search and highlight the next occurrence of the search string. Clicking on the *Replace All* button will replace all occurrences from the cursor position to the end of the file; if the *Wrap around* box is checked, then it will replace **all** occurrences in the **entire** document. Clicking on the *Done* button will dismiss the dialog.

Preferences

The *Preferences* dialog allows customization of the behavior and look of LAMMPS-GUI. The settings are grouped and each group is displayed within a tab.



General Settings:

- *Echo input to log:* when checked, all input commands, including variable expansions, are echoed to the *Output* window. This is equivalent to using *-echo screen* at the command line. There is no log *file* produced by default, since LAMMPS-GUI uses *-log none*.
- *Include citation details:* when checked full citation info will be included to the log window. This is equivalent to using *-cite screen* on the command line.
- *Show log window by default:* when checked, the screen output of a LAMMPS run will be collected in a log window during the run
- *Show chart window by default:* when checked, the thermodynamic output of a LAMMPS run will be collected and displayed in a chart window as line graphs.
- *Show slide show window by default:* when checked, a slide show window will be shown with images from a dump image command, if present, in the LAMMPS input.
- *Replace log window on new run:* when checked, an existing log window will be replaced on a new LAMMPS run, otherwise each run will create a new log window.
- *Replace chart window on new run:* when checked, an existing chart window will be replaced on a new LAMMPS run, otherwise each run will create a new chart window.

- *Replace image window on new render:* when checked, an existing chart window will be replaced when a new snapshot image is requested, otherwise each command will create a new image window.
- *Path to LAMMPS Shared Library File:* this option is only visible when LAMMPS-GUI was compiled to load the LAMMPS library at run time instead of being linked to it directly. With the *Browse..* button or by changing the text, a different shared library file with a different compilation of LAMMPS with different settings or from a different version can be loaded. After this setting was changed, LAMMPS-GUI needs to be re-launched.
- *Select Default Font:* Opens a font selection dialog where the type and size for the default font (used for everything but the editor and log) of the application can be set.
- *Select Text Font:* Opens a font selection dialog where the type and size for the text editor and log font of the application can be set.
- *Data update interval:* Allows to set the time interval between data updates during a LAMMPS run in milliseconds. The default is to update the data (for charts and output window) every 10 milliseconds. This is good for many cases. Set this to 100 milliseconds or more if LAMMPS-GUI consumes too many resources during a run. For LAMMPS runs that run *very* fast (for example in tutorial examples), however, data may be missed and through lowering this interval, this can be corrected. However, this will make the GUI use more resources. This setting may be changed to a value between 1 and 1000 milliseconds.
- *Charts update interval:* Allows to set the time interval between redrawing the plots in the *Charts* window in milliseconds. The default is to redraw the plots every 500 milliseconds. This is just for the drawing, data collection is managed with the previous setting.

Accelerators:

This tab enables selection of an accelerator package for LAMMPS to use and is equivalent to using the *-suffix* and *-package* flags on the command line. Only settings supported by the LAMMPS library and local hardware are available. The *Number of threads* field allows setting the maximum number of threads for the accelerator packages that use threads.

Snapshot Image:

This tab allows setting defaults for the snapshot images displayed in the *Image Viewer* window, such as its dimensions and the zoom factor applied. The *Antialias* switch will render images with twice the number of pixels for width and height and then smoothly scale the image back to the requested size. This produces higher quality images with smoother edges at the expense of requiring more CPU time to render the image. The *HQ Image mode* option turns on screen space ambient occlusion (SSAO) mode when rendering images. This is also more time consuming, but produces a more 'spatial' representation of the system shading of atoms by their depth. The *Shiny Image mode* option will render objects with a shiny surface when enabled. Otherwise the surfaces will be matted. The *Show Box* option selects whether the system box is drawn as a colored set of sticks. Similarly, the *Show Axes* option selects whether a representation of the three system axes will be drawn as colored sticks. The *VDW Style* checkbox selects whether atoms are represented by space filling spheres when checked or by smaller spheres and sticks. Finally there are a couple of drop down lists to select the background and box colors.

Editor Settings:

This tab allows tweaking settings of the editor window. Specifically the amount of padding to be added to LAMMPS commands, types or type ranges, IDs (e.g. for fixes), and names (e.g. for groups). The value set is the minimum width for the text element and it can be chosen in the range between 1 and 32.

The three settings which follow enable or disable the automatic reformatting when hitting the 'Enter' key, the automatic display of the completion pop-up window, and whether auto-save mode is enabled. In auto-save mode the editor buffer is saved before a run or before exiting LAMMPS-GUI.

Keyboard Shortcuts

Almost all functionality is accessible from the menu of the editor window or through keyboard shortcuts. The following shortcuts are available (On macOS use the Command key instead of Ctrl/Control).

Shortcut	Function	Shortcut	Function	Shortcut	Function
Ctrl+N	New File	Ctrl+Z	Undo edit	Ctrl+Enter	Run Input
Ctrl+O	Open File	Ctrl+Shift+Z	Redo edit	Ctrl+/	Stop Active Run
Ctrl+Shift+F	View File	Ctrl+C	Copy text	Ctrl+Shift+V	Set Variables
Ctrl+S	Save File	Ctrl+X	Cut text	Ctrl+I	Snapshot Image
Ctrl+Shift+S	Save File As	Ctrl+V	Paste text	Ctrl+L	Slide Show
Ctrl+Q	Quit Application	Ctrl+A	Select All	Ctrl+F	Find and Replace
Ctrl+W	Close Window	TAB	Reformat line	Shift+TAB	Show Completions
Ctrl+Shift+Enter	Run File	Ctrl+Shift+W	Show Variables	Ctrl+P	Preferences
Ctrl+Shift+A	About LAMMPS	Ctrl+Shift+H	Quick Help	Ctrl+Shift+G	LAMMPS-GUI Howto
Ctrl+Shift+M	LAMMPS Manual	Ctrl+?	Context Help	Ctrl+Shift+T	LAMMPS Tutorial

Further editing keybindings are documented with the Qt documentation. In case of conflicts the list above takes precedence.

All other windows only support a subset of keyboard shortcuts listed above. Typically, the shortcuts *Ctrl-/* (Stop Run), *Ctrl-W* (Close Window), and *Ctrl-Q* (Quit Application) are supported.

8.6.4 Moltemplate Tutorial

In this tutorial, we are going to use the tool *Moltemplate* to set up a classical molecular dynamic simulation using the *OPLS-AA force field*. The first task is to describe an organic compound and create a complete input deck for LAMMPS. The second task is to map the OPLS-AA force field to a molecular sample created with an external tool, e.g. PACKMOL, and exported as a PDB file. The files used in this tutorial can be found in the tools/moltemplate/tutorial-files folder of the LAMMPS source code distribution.

Simulating an organic solvent

This example aims to create a cubic box of the organic solvent formamide.

The first step is to create a molecular topology in the LAMMPS-template (LT) file format representing a single molecule, which will be stored in a Moltemplate object called _FAM inherits OPLSAA {}. This command states that the object _FAM is based on an existing object called OPLSAA, which contains OPLS-AA parameters, atom type definitions, partial charges, masses and bond-angle rules for many organic and biological compounds.

The atomic structure is the starting point to populate the command write('Data Atoms') {}, which will write the Atoms section in the LAMMPS data file. The OPLS-AA force field uses the atom_style full, therefore, this column format is used: # atomID molID atomType charge coordX coordY coordZ. The atomIDs are replaced with Moltemplate $-type variables, which are then substituted with unique numerical IDs. The same logic is applied to the molID, except that the same variable is used for the whole molecule. The atom types are assigned using @-type variables. The assignment of atom types (e.g. @atom:177, @atom:178) is done using the OPLS-AA atom types defined in the "In Charges" section of the file oplsaa.lt, looking for a reasonable match with the description of the atom. The resulting file (formamide.lt) follows:

```
_FAM inherits OPLSAA {

  # atomID    molID  atomType  charge  coordX  coordY  coordZ
  write('Data Atoms') {
    $atom:C00 $mol   @atom:177  0.00    0.100   0.490   0.0
    $atom:O01 $mol   @atom:178  0.00    1.091  -0.250   0.0
    $atom:N02 $mol   @atom:179  0.00   -1.121  -0.181   0.0
    $atom:H03 $mol   @atom:182  0.00   -2.013   0.272   0.0
    $atom:H04 $mol   @atom:182  0.00   -1.056  -1.190   0.0
    $atom:H05 $mol   @atom:221  0.00    0.144   1.570   0.0
  }

  # A list of the bonds in the molecule:
  # BondID   AtomID1   AtomID2
  write('Data Bond List') {
    $bond:C1 $atom:C00 $atom:O01
    $bond:C2 $atom:C00 $atom:H05
    $bond:C3 $atom:C00 $atom:N02
    $bond:C4 $atom:N02 $atom:H03
    $bond:C5 $atom:N02 $atom:H04
  }
}
```

You don't have to specify the charge in this example because they will be assigned according to the atom type. Analogously, only a "Data Bond List" section is needed as the atom type will determine the bond type. The other bonded interactions (e.g. angles, dihedrals, and impropers) will be automatically generated by Moltemplate.

If the simulation is non-neutral, or Moltemplate complains that you have missing bond, angle, or dihedral types, this means at least one of your atom types is incorrect.

The second step is to create a master file with instructions to build a starting structure and the LAMMPS commands to run an NPT simulation. The master file (solv_01.lt) follows:

```
# Import the force field.
import /usr/local/moltemplate/moltemplate/force_fields/oplsaa.lt
import formamide.lt # after oplsaa.lt, as it depends on it.

# Create the input sample.
```

(continues on next page)

(continued from previous page)

```
solv = new _FAM [5].move( 4.6, 0, 0)
                [5].move( 0, 4.6, 0)
                [5].move( 0, 0, 4.6)
solv[*][*][*].move(-11.5, -11.5, -11.5)

# Set the simulation box.
write_once("Data Boundary") {
 -11.5 11.5 xlo xhi
 -11.5 11.5 ylo yhi
 -11.5 11.5 zlo zhi
}

# Create an input deck for LAMMPS.
write_once("In Init"){
 # Input variables.
 variable run    string solv_01   # output name
 variable ts     equal  1         # timestep
 variable temp   equal  300       # equilibrium temperature
 variable p      equal  1.        # equilibrium pressure
 variable d      equal  1000      # output frequency
 variable equi   equal  5000      # Equilibration steps
 variable prod   equal  30000     # Production steps

 # PBC (set them before the creation of the box).
 boundary p p p
}

# Run an NPT simulation.
write_once("In Run"){
 # Derived variables.
 variable tcouple equal \$\{ts\}*100
 variable pcouple equal \$\{ts\}*1000

 # Output.
 thermo          \$d
 thermo_style custom step etotal evdwl ecoul elong ebond eangle &
 edihed eimp ke pe temp press vol density cpu
 thermo_modify flush yes

 # Trajectory.
 dump TRJ all dcd \$d \$\{run\}.dcd
 dump_modify TRJ unwrap yes

 # Thermalisation and relaxation, NPT ensemble.
 timestep       \$\{ts\}
 fix            NPT all npt temp \$\{temp\} \$\{temp\} \$\{tcouple\} iso \$p \$p \$\{pcouple\}
 velocity all create \$\{temp\} 858096 dist gaussian
 # Short runs to update the PPPM settings as the box shinks.
 run    \$\{equi\} post no
 run    \$\{equi\} post no
 run    \$\{equi\} post no
 run    \$\{equi\}
```

(continues on next page)

(continued from previous page)

```
# From now on, the density shouldn't change too much.
run     \$\{prod\}
unfix NPT
}
```

The first two commands insert the content of files oplsaa.lt and formamide.lt into the master file. At this point, we can use the command solv = new _FAM [N] to create N copies of a molecule of type _FAM. In this case, we create an array of 5*5*5 molecules on a cubic grid using the coordinate transformation command .move(4.6, 0, 0). See the Moltemplate documentation to learn more about the syntax. As the sample was created from scratch, we also specify the simulation box size in the "Data Boundary" section.

The LAMMPS setting for the force field are specified in the file oplsaa.lt and are written automatically in the input deck. We also specify the boundary conditions and a set of variables in the "In Init" section. The remaining commands to run an NPT simulation are written in the "In Run" section. Note that in this script, LAMMPS variables are protected with the escape character \ to distinguish them from Moltemplate variables, e.g. \$\{run\} is a LAMMPS variable that is written in the input deck as ${run}.

Compile the master file with:

```
moltemplate.sh -overlay-all solv_01.lt
```

And execute the simulation with the following:

```
mpirun -np 4 lmp -in solv_01.in -l solv_01.log
```

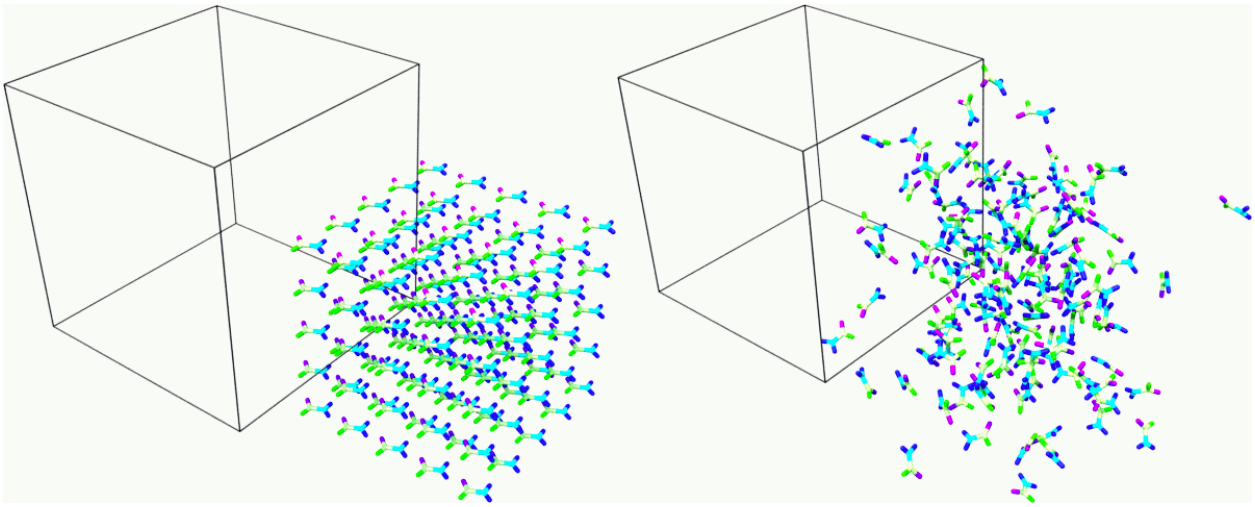


Fig. 10: Snapshot of the sample at the beginning and end of the simulation. Rendered with Ovito.

Mapping an existing structure

Another helpful way to use Moltemplate is mapping an existing molecular sample to a force field. This is useful when a complex sample is assembled from different simulations or created with specialized software (e.g. PACKMOL). As in the previous example, all molecular species in the sample must be defined using single-molecule Moltemplate objects. For this example, we use a short polymer in a box containing water molecules and ions in the PDB file model.pdb.

It is essential to understand that the order of atoms in the PDB file and in the Moltemplate master script must match, as we are using the coordinates from the PDB file in the order they appear. The order of atoms and molecules in the PDB file provided is as follows:

- 500 water molecules, with atoms ordered in this sequence:

```
ATOM      1  O   MOL D   1       5.901   7.384   1.103  0.00  0.00      DUM
ATOM      2  H   MOL D   1       6.047   8.238   0.581  0.00  0.00      DUM
ATOM      3  H   MOL D   1       6.188   7.533   2.057  0.00  0.00      DUM
```

- 1 polymer molecule.
- 1 Ca^{2+} ion.
- 2 Cl^{-} ions.

In the master LT file, this sequence of molecules is matched with the following commands:

```
# Create the sample.
wat=new SPC[500]
pol=new PolyNIPAM[1]
cat=new Ca[1]
ani=new Cl[2]
```

Note that the first command would create 500 water molecules in the same position in space, and the other commands will use the coordinates specified in the corresponding molecular topology block. However, the coordinates will be overwritten by rendering an external atomic structure file. Note that if the same molecule species are scattered in the input structure, it is recommended to reorder and group together for molecule types to facilitate the creation of the input sample.

The molecular topology for the polymer is created as in the previous example, with the atom types assigned as in the following schema:

The molecular topology of the water and ions is stated directly into the master file for the sake of space, but they could also be written in a separate file(s) and imported before the sample is created.

The resulting master LT file defining short annealing at a fixed volume (NVT) follows:

```
# Use the OPLS-AA force field for all species.
import /usr/local/moltemplate/moltemplate/force_fields/oplsaa.lt
import PolyNIPAM.lt

# Define the SPC water and ions as in the OPLS-AA
Ca inherits OPLSAA {
  write("Data Atoms"){
    $atom:a1  $mol:. @atom:354 0.0  0.00000 0.00000 0.000000
  }
}
Cl inherits OPLSAA {
  write("Data Atoms"){
    $atom:a1  $mol:. @atom:344 0.0  0.00000 0.00000 0.000000
```

(continues on next page)

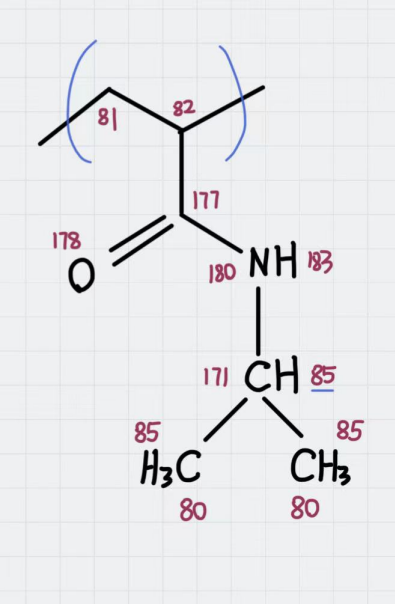


Fig. 11: Atom types assigned to the polymer's repeating unit.

(continued from previous page)

```
  }
}
SPC inherits OPLSAA {
  write("Data Atoms"){
    $atom:O  $mol:. @atom:76 0.  0.0000000 0.00000 0.000000
    $atom:H1 $mol:. @atom:77 0.  0.8164904 0.00000 0.5773590
    $atom:H2 $mol:. @atom:77 0. -0.8164904 0.00000 0.5773590
  }
  write("Data Bond List") {
    $bond:OH1 $atom:O $atom:H1
    $bond:OH2 $atom:O $atom:H2
  }
}

# Create the sample.
wat=new SPC[500]
pol=new PolyNIPAM[1]
cat=new Ca[1]
ani=new Cl[2]

# Periodic boundary conditions:
write_once("Data Boundary"){
  0 26 xlo xhi
  0 26 ylo yhi
  0 26 zlo zhi
}

# Define the input variables.
write_once("In Init"){
```

(continues on next page)

(continued from previous page)

```
  # Input variables.
  variable run    string sample01  # output name
  variable ts     equal  2         # timestep
  variable temp   equal  298.15    # equilibrium temperature
  variable p      equal  1.        # equilibrium pressure
  variable equi   equal  30000     # equilibration steps

  # PBC (set them before the creation of the box).
  boundary p p p
  neighbor        3 bin
}

# Run an NVT simulation.
write_once("In Run"){
  # Set the output.
  thermo          1000
  thermo_style    custom step etotal evdwl ecoul elong ebond eangle &
  edihed eimp pe ke temp press atoms vol density cpu
  thermo_modify flush yes
  compute pe1 all pe/atom pair
  dump TRJ all custom 100 \$\{run\}.dump id xu yu zu c_pe1

  # Minimise the input structure, just in case.
  minimize        .01 .001 1000 100000
  write_data \$\{run\}.min

  # Set the constrains.
  group watergroup type @atom:76 @atom:77
  fix 0 watergroup shake 0.0001 10 0 b @bond:042_043 a @angle:043_042_043

  # Short annealing.
  timestep        \$\{ts\}
  fix       1 all nvt temp \$\{temp\} \$\{temp\} \$(100*dt)
  velocity    all create \$\{temp\} 315443
  run             \$\{equi\}
  unfix 1
}
```

In this example, the water model is SPC and it is defined in the oplsaa.lt file with atom types @atom:76 and @atom:77. For water we also use the group and fix shake commands with Moltemplate @-type variables, to ensure consistency with the numerical values assigned during compilation. To identify the bond and angle types, look for the extended @atom IDs, which in this case are:

```
replace{ @atom:76 @atom:76_b042_a042_d042_i042 }
replace{ @atom:77 @atom:77_b043_a043_d043_i043 }
```

From which we can identify the following "Data Bonds By Type": @bond:042_043 @atom:*_b042*_a*_d*_i* @atom:*_b043*_a*_d*_i* and "Data Angles By Type": @angle:043_042_043 @atom:*_b*_a043*_d*_i* @atom:*_b*_a042*_d*_i* @atom:*_b*_a043*_d*_i*

Compile the master file with:

```
moltemplate.sh -overlay-all -pdb model.pdb sample01.lt
```

And execute the simulation with the following:

```
mpirun -np 4 lmp -in sample01.in -l sample01.log
```

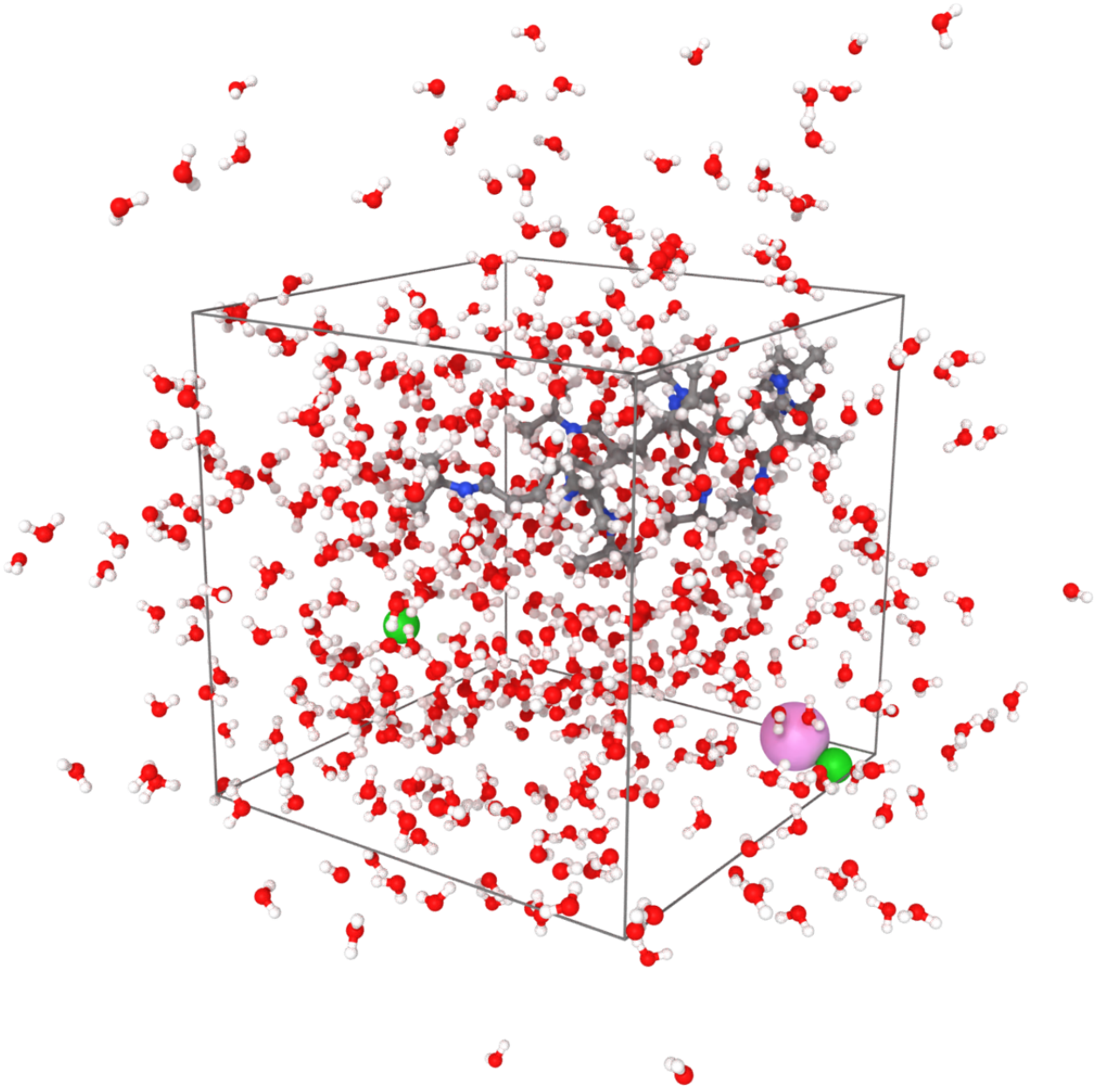


Fig. 12: Sample visualized with Ovito loading the trajectory into the DATA file written after minimization.

(OPLS-AA) Jorgensen, Maxwell, Tirado-Rives, J Am Chem Soc, 118(45), 11225-11236 (1996).

8.6.5 PyLammps Tutorial

Contents

Overview

PyLammps is a Python wrapper class for LAMMPS which can be created on its own or use an existing lammps Python object. It creates a simpler, more "pythonic" interface to common LAMMPS functionality, in contrast to the lammps wrapper for the LAMMPS *C language library interface API* which is written using Python ctypes. The lammps wrapper is discussed on the *Use Python with LAMMPS* doc page.

Unlike the flat ctypes interface, PyLammps exposes a discoverable API. It no longer requires knowledge of the underlying C++ code implementation. Finally, the IPyLammps wrapper builds on top of PyLammps and adds some additional features for IPython integration into Jupyter notebooks, e.g. for embedded visualization output from *dump style image*.

Comparison of lammps and PyLammps interfaces

lammps.lammps

- uses ctypes
- direct memory access to native C++ data with optional support for NumPy arrays
- provides functions to send and receive data to LAMMPS
- interface modeled after the LAMMPS *C language library interface API*
- requires knowledge of how LAMMPS internally works (C pointers, etc)
- full support for running Python with MPI using mpi4py

lammps.PyLammps

- higher-level abstraction built on *top* of original ctypes based interface
- manipulation of Python objects
- communication with LAMMPS is hidden from API user
- shorter, more concise Python
- better IPython integration, designed for quick prototyping
- designed for serial execution

Quick Start

System-wide Installation

Step 1: Building LAMMPS as a shared library

To use LAMMPS inside of Python it has to be compiled as shared library. This library is then loaded by the Python interface. In this example we enable the MOLECULE package and compile LAMMPS with PNG, JPEG and FFMPEG output support enabled.

Step 1a: For the CMake based build system, the steps are:

```
mkdir $LAMMPS_DIR/build-shared
cd  $LAMMPS_DIR/build-shared

# MPI, PNG, Jpeg, FFMPEG are auto-detected
cmake ../cmake -DPKG_MOLECULE=yes -DBUILD_LIB=yes -DBUILD_SHARED_LIBS=yes
make
```

Step 1b: For the legacy, make based build system, the steps are:

```
cd $LAMMPS_DIR/src

# add packages if necessary
make yes-MOLECULE

# compile shared library using Makefile
make mpi mode=shlib LMP_INC="-DLAMMPS_PNG -DLAMMPS_JPEG -DLAMMPS_FFMPEG"␣
↪JPG_LIB="-lpng -ljpeg"
```

Step 2: Installing the LAMMPS Python package

PyLammps is part of the lammps Python package. To install it simply install that package into your current Python installation with:

```
make install-python
```

Note

Recompiling the shared library requires re-installing the Python package

Installation inside of a virtualenv

You can use virtualenv to create a custom Python environment specifically tuned for your workflow.

Benefits of using a virtualenv

- isolation of your system Python installation from your development installation
- installation can happen in your user directory without root access (useful for HPC clusters)
- installing packages through pip allows you to get newer versions of packages than e.g., through apt-get or yum package managers (and without root access)
- you can even install specific old versions of a package if necessary

Prerequisite (e.g. on Ubuntu)

```
apt-get install python-virtualenv
```

Creating a virtualenv with lammps installed

```
# create virtualenv named 'testing'
virtualenv $HOME/python/testing

# activate 'testing' environment
source $HOME/python/testing/bin/activate
```

Now configure and compile the LAMMPS shared library as outlined above. When using CMake and the shared library has already been build, you need to re-run CMake to update the location of the python executable to the location in the virtual environment with:

```
cmake . -DPython_EXECUTABLE=$(which python)

# install LAMMPS package in virtualenv
(testing) make install-python

# install other useful packages
(testing) pip install matplotlib jupyter mpi4py

...

# return to original shell
(testing) deactivate
```

Creating a new instance of PyLammps

To create a PyLammps object you need to first import the class from the lammps module. By using the default constructor, a new *lammps* instance is created.

```
from lammps import PyLammps
L = PyLammps()
```

You can also initialize PyLammps on top of this existing *lammps* object:

```
from lammps import lammps, PyLammps
lmp = lammps()
L = PyLammps(ptr=lmp)
```

Commands

Sending a LAMMPS command with the existing library interfaces is done using the command method of the lammps object instance.

For instance, let's take the following LAMMPS command:

```
region box block 0 10 0 5 -0.5 0.5
```

In the original interface this command can be executed with the following Python code if *L* was a lammps instance:

```
L.command("region box block 0 10 0 5 -0.5 0.5")
```

With the PyLammps interface, any command can be split up into arbitrary parts separated by white-space, passed as individual arguments to a region method.

```
L.region("box block", 0, 10, 0, 5, -0.5, 0.5)
```

Note that each parameter is set as Python literal floating-point number. In the PyLammps interface, each command takes an arbitrary parameter list and transparently merges it to a single command string, separating individual parameters by white-space.

The benefit of this approach is avoiding redundant command calls and easier parameterization. In the original interface parameterization needed to be done manually by creating formatted strings.

```
L.command("region box block %f %f %f %f %f %f" % (xlo, xhi, ylo, yhi, zlo, zhi))
```

In contrast, methods of PyLammps accept parameters directly and will convert them automatically to a final command string.

```
L.region("box block", xlo, xhi, ylo, yhi, zlo, zhi)
```

System state

In addition to dispatching commands directly through the PyLammps object, it also provides several properties which allow you to query the system state.

L.system
Is a dictionary describing the system such as the bounding box or number of atoms

L.system.xlo, L.system.xhi
bounding box limits along x-axis

L.system.ylo, L.system.yhi
bounding box limits along y-axis

L.system.zlo, L.system.zhi
bounding box limits along z-axis

L.communication
configuration of communication subsystem, such as the number of threads or processors

L.communication.nthreads
number of threads used by each LAMMPS process

L.communication.nprocs
number of MPI processes used by LAMMPS

L.fixes
List of fixes in the current system

L.computes
List of active computes in the current system

L.dump
List of active dumps in the current system

L.groups
List of groups present in the current system

Working with LAMMPS variables

LAMMPS variables can be both defined and accessed via the PyLammps interface.

To define a variable you can use the *variable* command:

```
L.variable("a index 2")
```

A dictionary of all variables is returned by L.variables

you can access an individual variable by retrieving a variable object from the L.variables dictionary by name

```
a = L.variables['a']
```

The variable value can then be easily read and written by accessing the value property of this object.

```
print(a.value)
a.value = 4
```

Retrieving the value of an arbitrary LAMMPS expressions

LAMMPS expressions can be immediately evaluated by using the eval method. The passed string parameter can be any expression containing global thermo values, variables, compute or fix data.

```
result = L.eval("ke") # kinetic energy
result = L.eval("pe") # potential energy

result = L.eval("v_t/2.0")
```

Accessing atom data

All atoms in the current simulation can be accessed by using the L.atoms list. Each element of this list is an object which exposes its properties (id, type, position, velocity, force, etc.).

```
# access first atom
L.atoms[0].id
L.atoms[0].type

# access second atom
L.atoms[1].position
L.atoms[1].velocity
L.atoms[1].force
```

Some properties can also be used to set:

```
# set position in 2D simulation
L.atoms[0].position = (1.0, 0.0)

# set position in 3D simulation
L.atoms[0].position = (1.0, 0.0, 1.)
```

Evaluating thermo data

Each simulation run usually produces thermo output based on system state, computes, fixes or variables. The trajectories of these values can be queried after a run via the L.runs list. This list contains a growing list of run data. The first element is the output of the first run, the second element that of the second run.

```
L.run(1000)
L.runs[0] # data of first 1000 time steps

L.run(1000)
L.runs[1] # data of second 1000 time steps
```

Each run contains a dictionary of all trajectories. Each trajectory is accessible through its thermo name:

```
L.runs[0].thermo.Step # list of time steps in first run
L.runs[0].thermo.Ke   # list of kinetic energy values in first run
```

Together with matplotlib plotting data out of LAMMPS becomes simple:

```
import matplotlib.plot as plt
steps = L.runs[0].thermo.Step
ke    = L.runs[0].thermo.Ke
plt.plot(steps, ke)
```

Error handling with PyLammps

Using C++ exceptions in LAMMPS for errors allows capturing them on the C++ side and rethrowing them on the Python side. This way you can handle LAMMPS errors through the Python exception handling mechanism.

Warning

Capturing a LAMMPS exception in Python can still mean that the current LAMMPS process is in an illegal state and must be terminated. It is advised to save your data and terminate the Python instance as quickly as possible.

Using PyLammps in IPython notebooks and Jupyter

If the LAMMPS Python package is installed for the same Python interpreter as IPython, you can use PyLammps directly inside of an IPython notebook inside of Jupyter. Jupyter is a powerful integrated development environment (IDE) for many dynamic languages like Python, Julia and others, which operates inside of any web browser. Besides auto-completion and syntax highlighting it allows you to create formatted documents using Markup, mathematical formulas, graphics and animations intermixed with executable Python code. It is a great format for tutorials and showcasing your latest research.

To launch an instance of Jupyter simply run the following command inside your Python environment (this assumes you followed the Quick Start instructions):

```
jupyter notebook
```

IPyLammps Examples

Examples of IPython notebooks can be found in the python/examples/pylammps subdirectory. To open these notebooks launch *jupyter notebook* inside this directory and navigate to one of them. If you compiled and installed a LAMMPS shared library with exceptions, PNG, JPEG and FFMPEG support you should be able to rerun all of these notebooks.

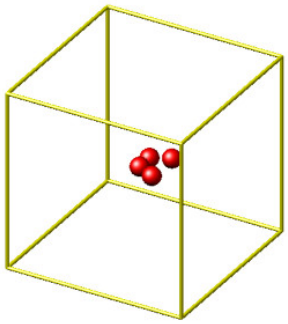
Validating a dihedral potential

This example showcases how an IPython Notebook can be used to compare a simple LAMMPS simulation of a harmonic dihedral potential to its analytical solution. Four atoms are placed in the simulation and the dihedral potential is applied on them using a datafile. Then one of the atoms is rotated along the central axis by setting its position from Python, which changes the dihedral angle.

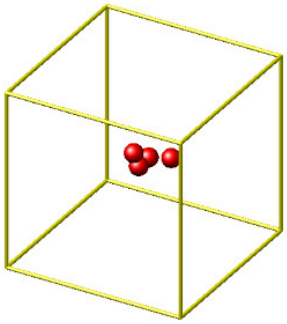
```
phi = [d \* math.pi / 180 for d in range(360)]

pos = [(1.0, math.cos(p), math.sin(p)) for p in phi]

pe = []
for p in pos:
    L.atoms[3].position = p
    L.run(0)
    pe.append(L.eval("pe"))
```

By evaluating the potential energy for each position we can verify that trajectory with the analytical formula. To compare both solutions, we plot both trajectories over each other using matplotlib, which embeds the generated plot inside the IPython notebook.

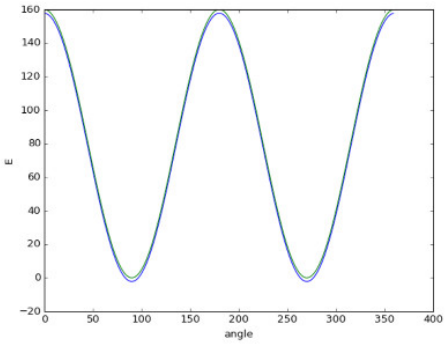
```
In [11]: L.image(zoom=1.0)
Out[11]:
```



```
In [14]: L.image(zoom=1.0)
Out[14]:
```



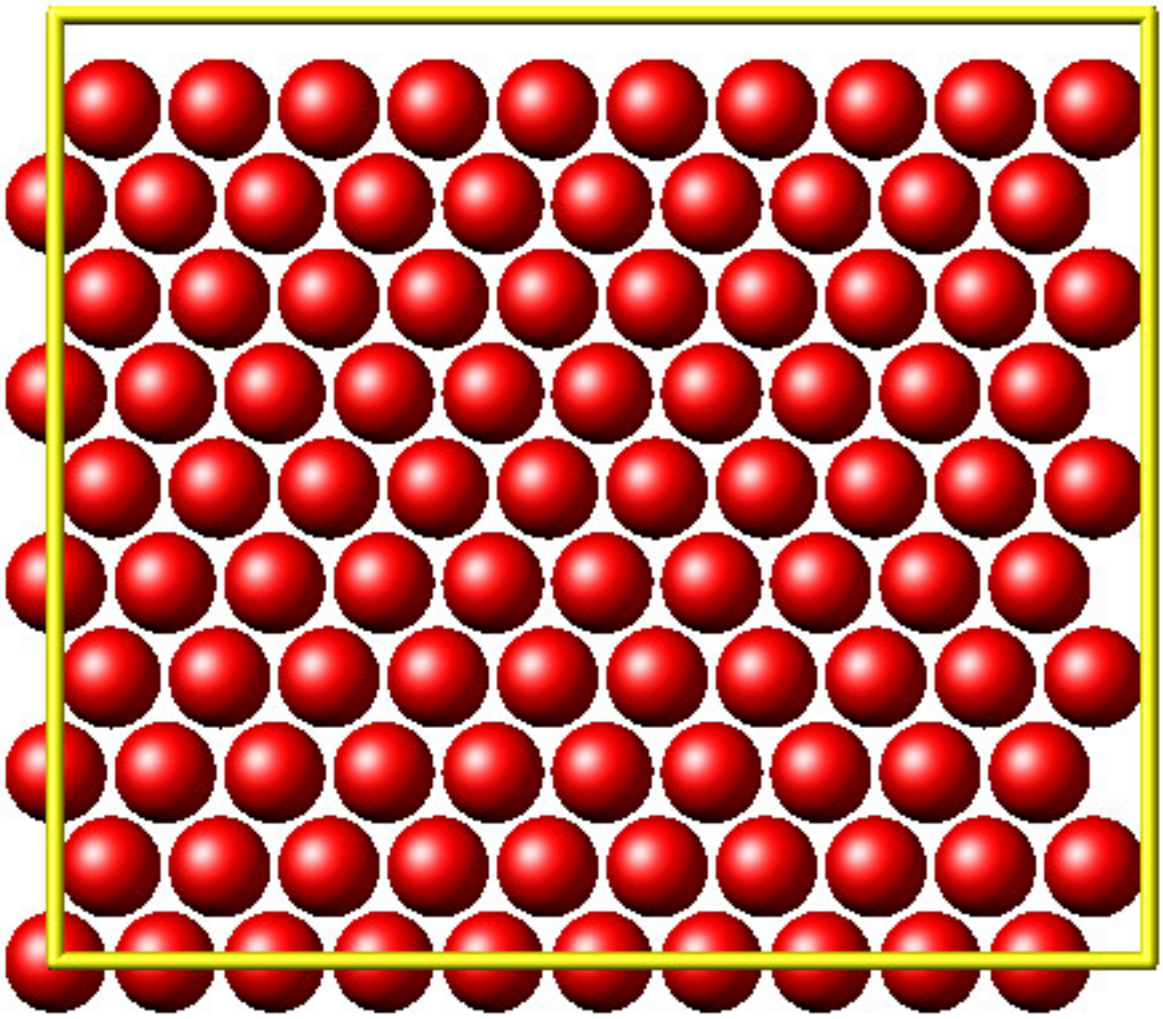
```
In [27]: plt.plot(range(360), pe, range(360), E_analytical)
         plt.xlabel('angle')
         plt.ylabel('E')
```



Running a Monte Carlo relaxation

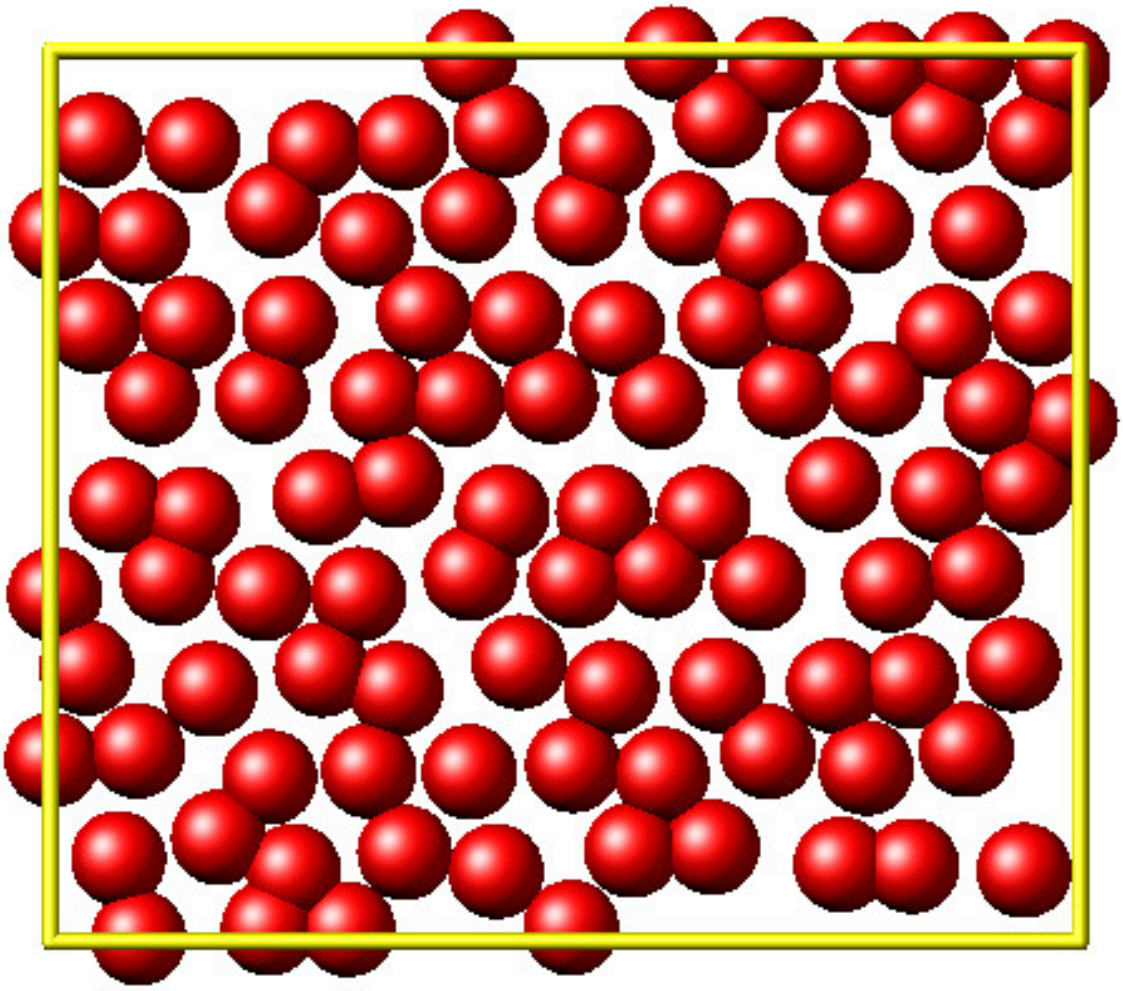
This second example shows how to use PyLammps to create a 2D Monte Carlo Relaxation simulation, computing and plotting energy terms and even embedding video output.

Initially, a 2D system is created in a state with minimal energy.



It is then disordered by moving each atom by a random delta.

```
random.seed(27848)
deltaperturb = 0.2

for i in range(L.system.natoms):
    x, y = L.atoms[i].position
    dx = deltaperturb \* random.uniform(-1, 1)
    dy = deltaperturb \* random.uniform(-1, 1)
    L.atoms[i].position  = (x+dx, y+dy)

L.run(0)
```



Finally, the Monte Carlo algorithm is implemented in Python. It continuously moves random atoms by a random delta and only accepts certain moves.

```
estart = L.eval("pe")
elast = estart

naccept = 0
energies = [estart]

niterations = 3000
deltamove = 0.1
kT = 0.05

natoms = L.system.natoms

for i in range(niterations):
    iatom = random.randrange(0, natoms)
    current_atom = L.atoms[iatom]
```

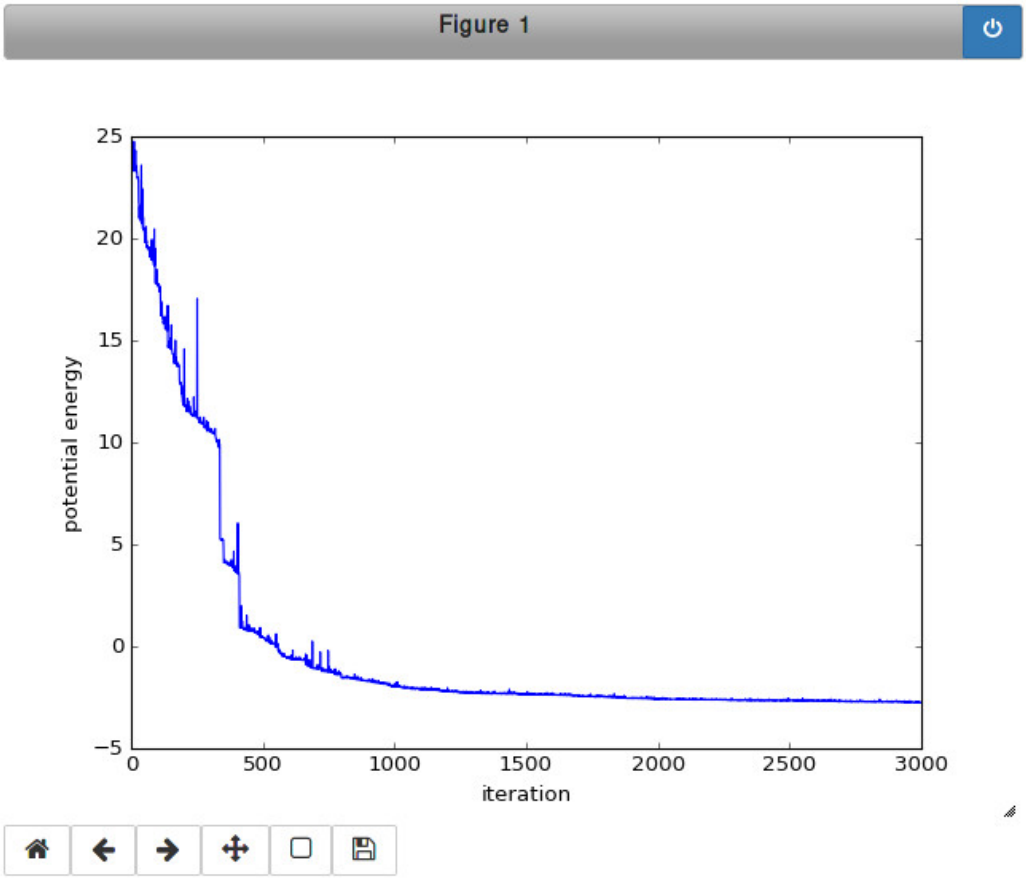
(continues on next page)

(continued from previous page)

```
x0, y0 = current_atom.position

dx = deltamove \* random.uniform(-1, 1)
dy = deltamove \* random.uniform(-1, 1)

current_atom.position = (x0+dx, y0+dy)

L.run(1, "pre no post no")

e = L.eval("pe")
energies.append(e)

if e <= elast:
    naccept += 1
    elast = e
elif random.random() <= math.exp(natoms\*(elast-e)/kT):
    naccept += 1
    elast = e
else:
    current_atom.position = (x0, y0)
```

The energies of each iteration are collected in a Python list and finally plotted using matplotlib.

```
In [20]: plt.xlabel('iteration')
         plt.ylabel('potential energy')
         plt.plot(energies)
```




```
Out[20]: [<matplotlib.lines.Line2D at 0x7f26a40123c8>]
```

The IPython notebook also shows how to use dump commands and embed video files inside of the IPython notebook.

Using PyLammps and mpi4py (Experimental)

PyLammps can be run in parallel using mpi4py. This python package can be installed using

```
pip install mpi4py
```

Warning

Usually, any PyLammps command must be executed by *all* MPI processes. However, evaluations and querying the system state is only available on MPI rank 0. Using these functions from other MPI ranks will raise an exception.

The following is a short example which reads in an existing LAMMPS input file and executes it in parallel. You can find in.melt in the examples/melt folder. Please take note that the PyLammps.eval() is called only from MPI rank 0.

```
from mpi4py import MPI
from lammps import PyLammps
```

(continues on next page)

(continued from previous page)

```
L = PyLammps()
L.file("in.melt")

if MPI.COMM_WORLD.rank == 0:
    print("Potential energy: ", L.eval("pe"))

MPI.Finalize()
```

To run this script (melt.py) in parallel using 4 MPI processes we invoke the following mpirun command:

```
mpirun -np 4 python melt.py
```

Feedback and Contributing

If you find this Python interface useful, please feel free to provide feedback and ideas on how to improve it to Richard Berger (richard.berger@outlook.com). We also want to encourage people to write tutorial style IPython notebooks showcasing LAMMPS usage and maybe their latest research results.

8.6.6 Using LAMMPS on Windows 10 with WSL

written by Richard Berger

It's always been tricky for us to have LAMMPS users and developers work on Windows. We primarily develop LAMMPS to run on Linux clusters. To teach LAMMPS in workshop settings, we had to redirect Windows users to Linux Virtual Machines such as VirtualBox or Unix-like compilation with Cygwin.

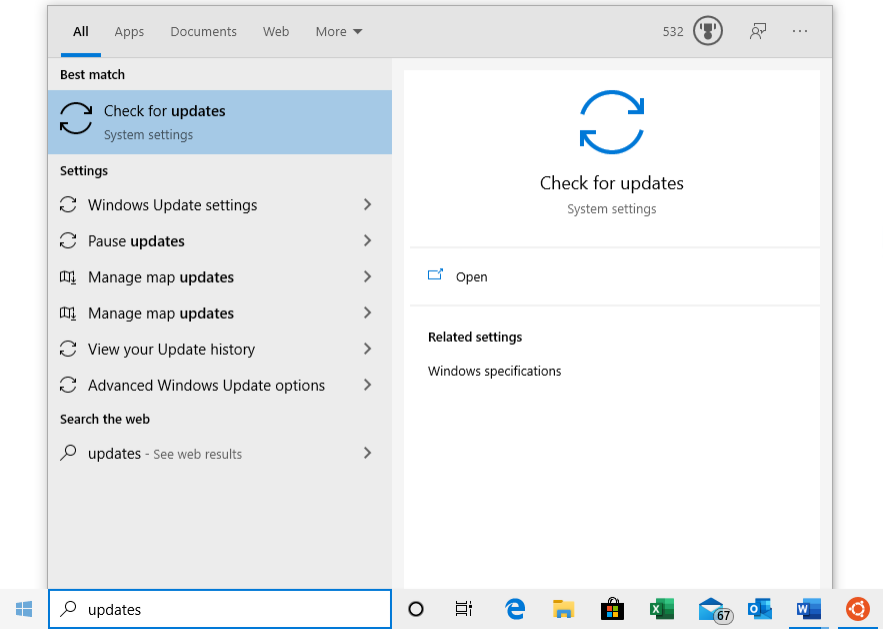
With the latest updates in Windows 10 (Version 2004, Build 19041 or higher), Microsoft has added a new way to work on Linux-based code. The Windows Subsystem for Linux (WSL). With WSL Version 2, you now get a Linux Virtual Machine that transparently integrates into Windows. All you need is to ensure you have the latest Windows updates installed and enable this new feature. Linux VMs are then easily installed using the Microsoft Store.

In this tutorial, I'll show you how to set up and compile LAMMPS for both serial and MPI usage in WSL2.

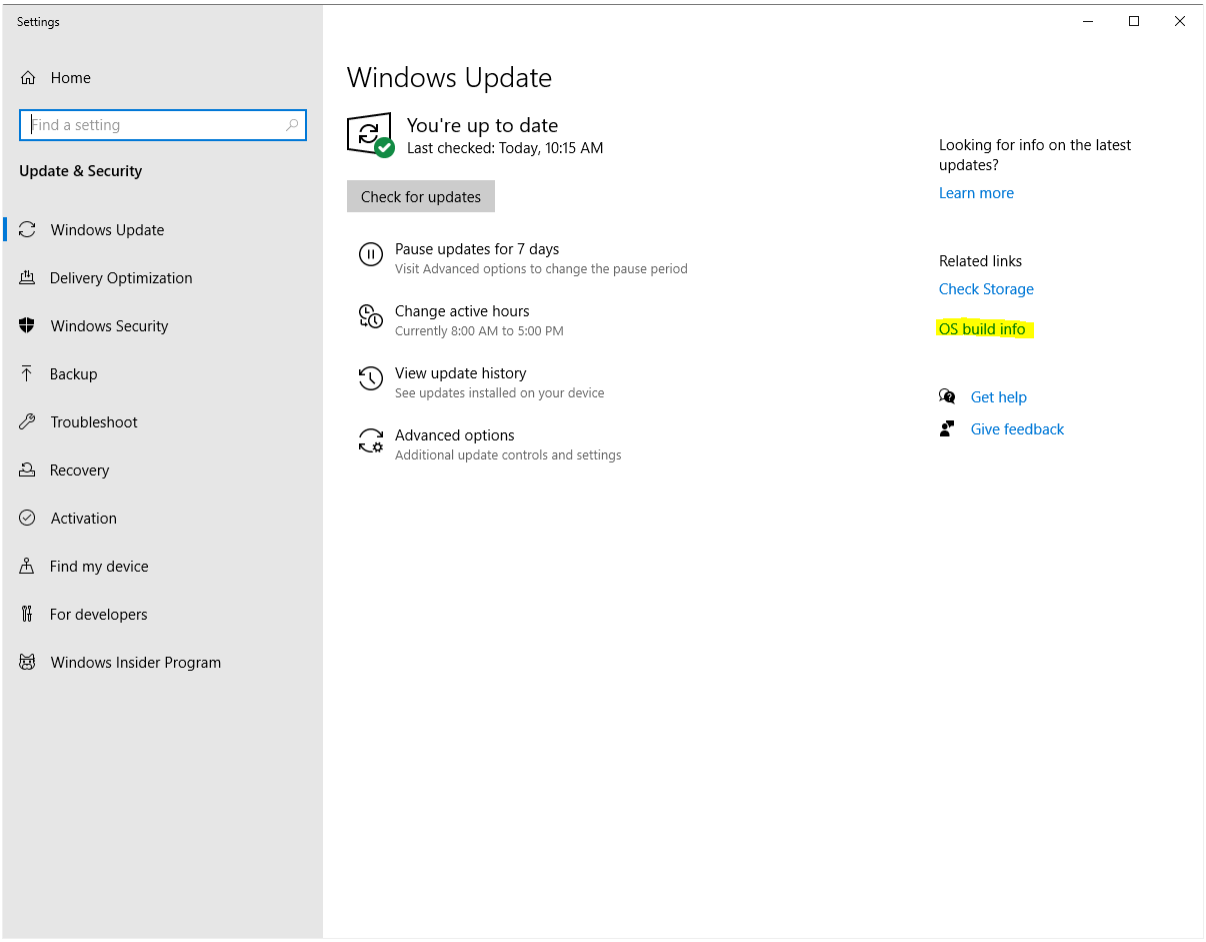
Installation

Upgrade to the latest Windows 10

Type "Updates" in Windows Start and select "Check for Updates".

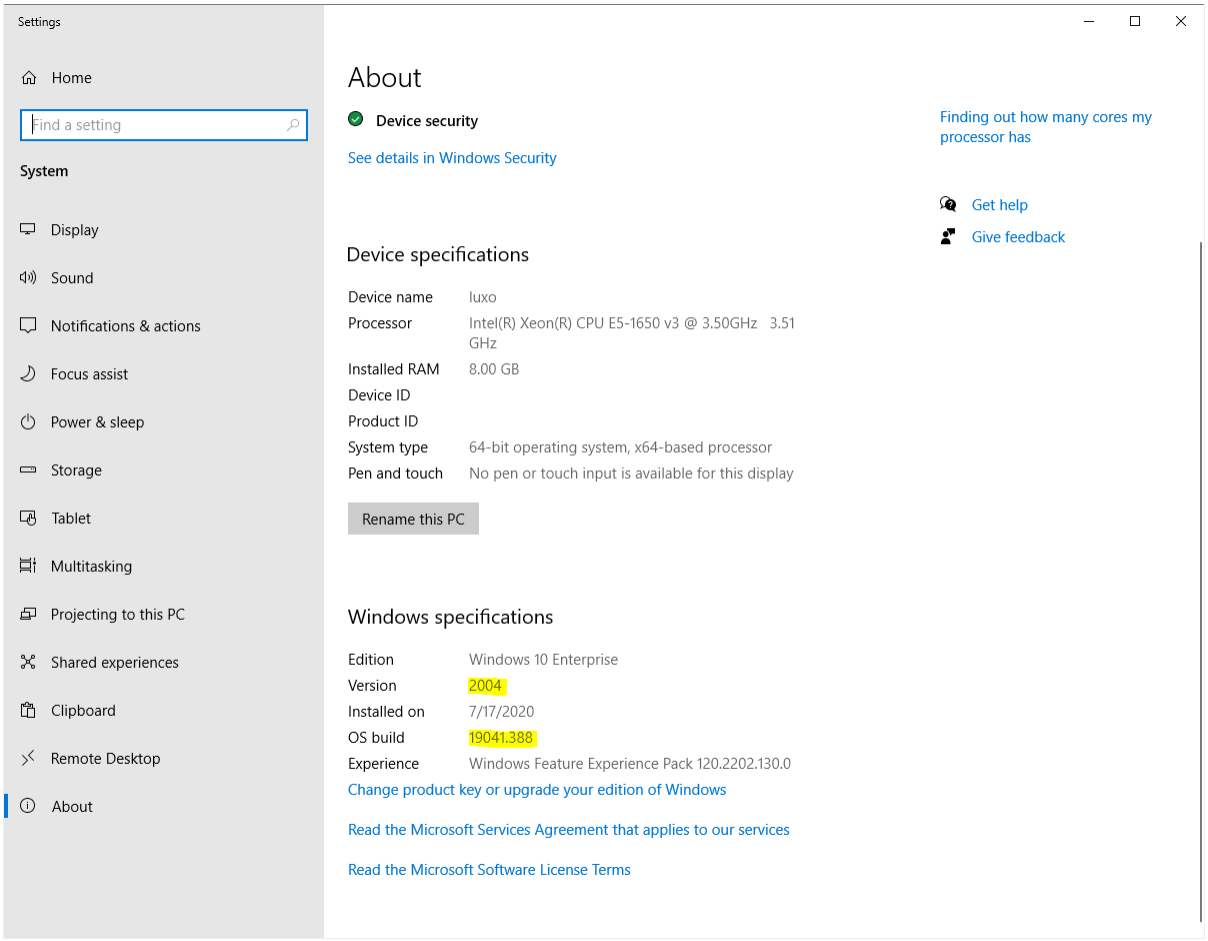


Install all pending updates and reboot your system as many times as necessary. Continue until your Windows installation is updated.



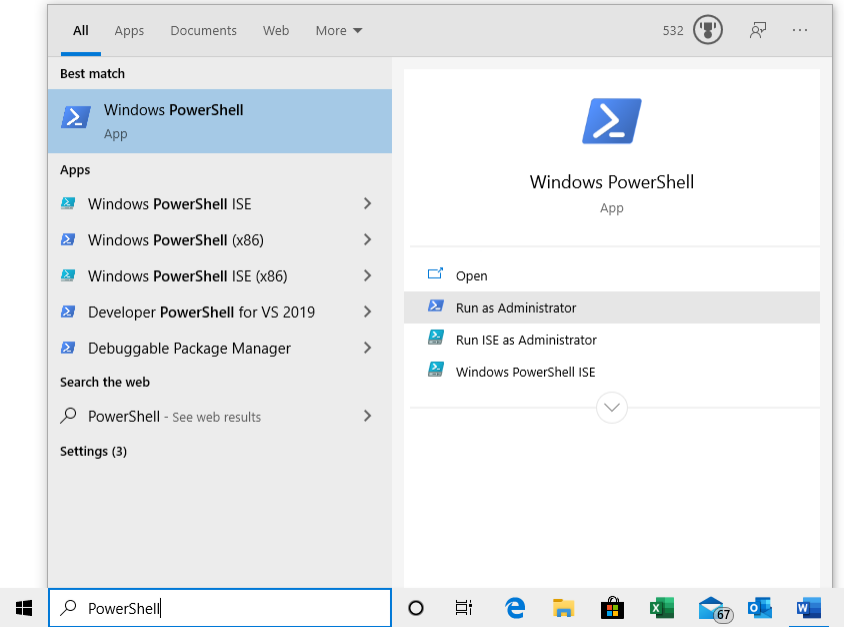
Verify your system has at least **version 2004 and build 19041 or later**. You can find this information by clicking on

"OS build info".



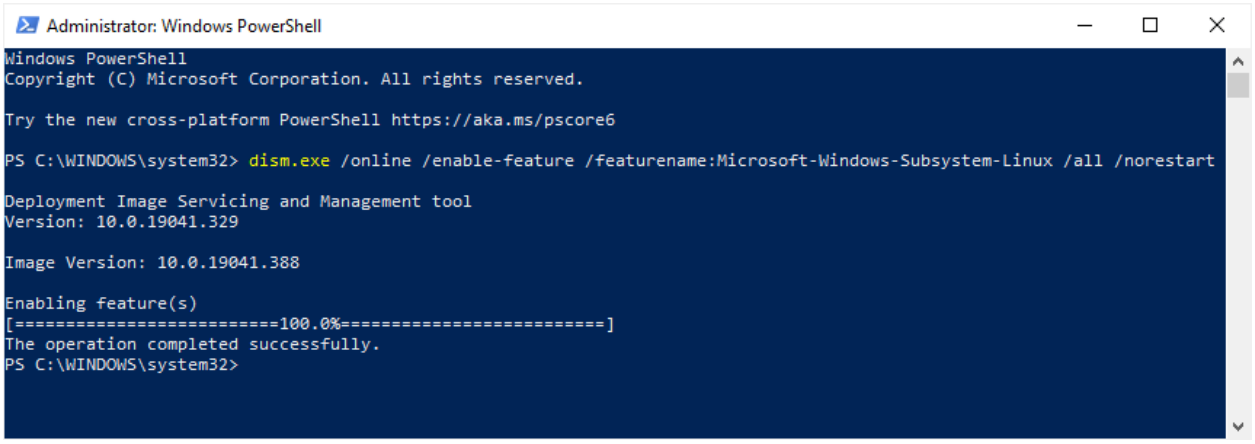
Enable WSL

Next, we must install two additional Windows features to enable WSL support. Open a PowerShell window as an administrator. Type "PowerShell" in Windows Start and select "Run as Administrator".



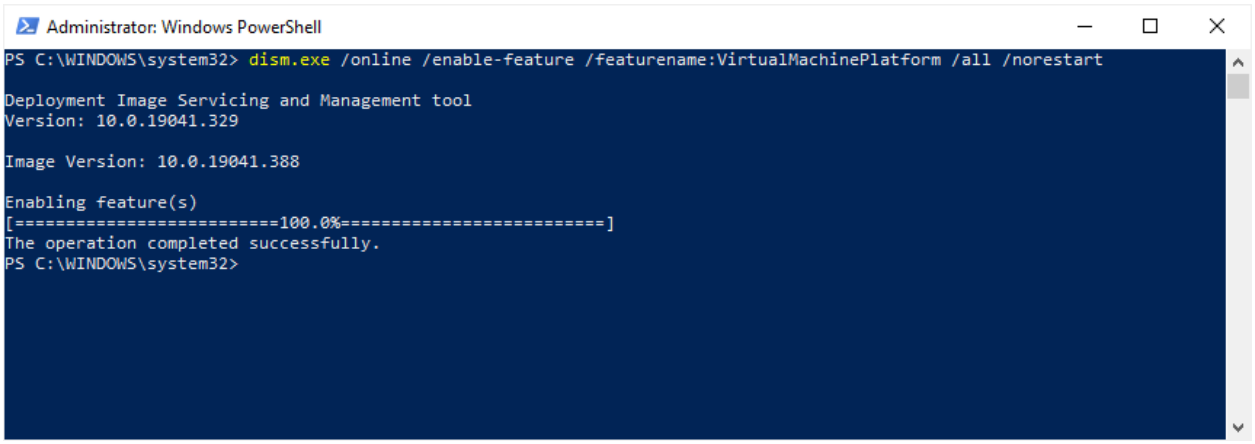
Windows will ask you for administrator access. After you accept a new command line window will appear. Type in the following command to install WSL:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```



Next, enable the VirtualMachinePlatform feature using the following command:

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```



Finally, reboot your system.

Update WSL kernel component

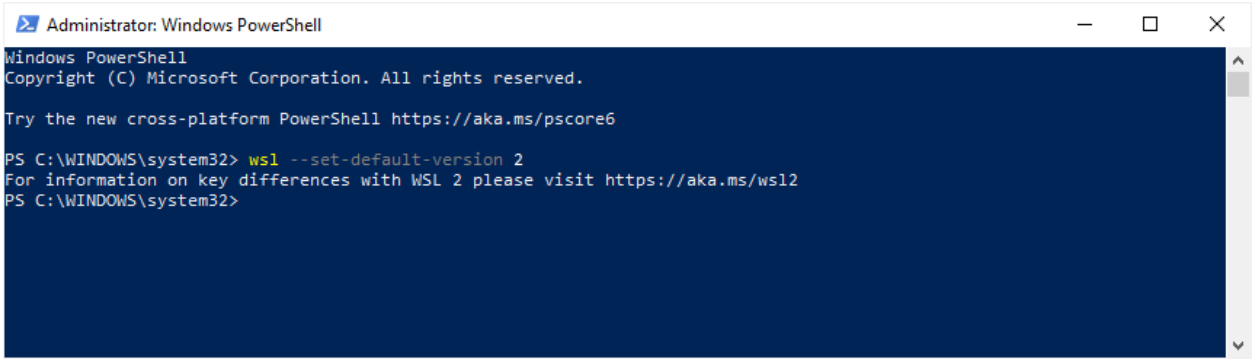
Download and install the WSL Kernel Component Update. Afterwards, reboot your system.

Set WSL2 as default

Again, open PowerShell as administrator and run the following command:

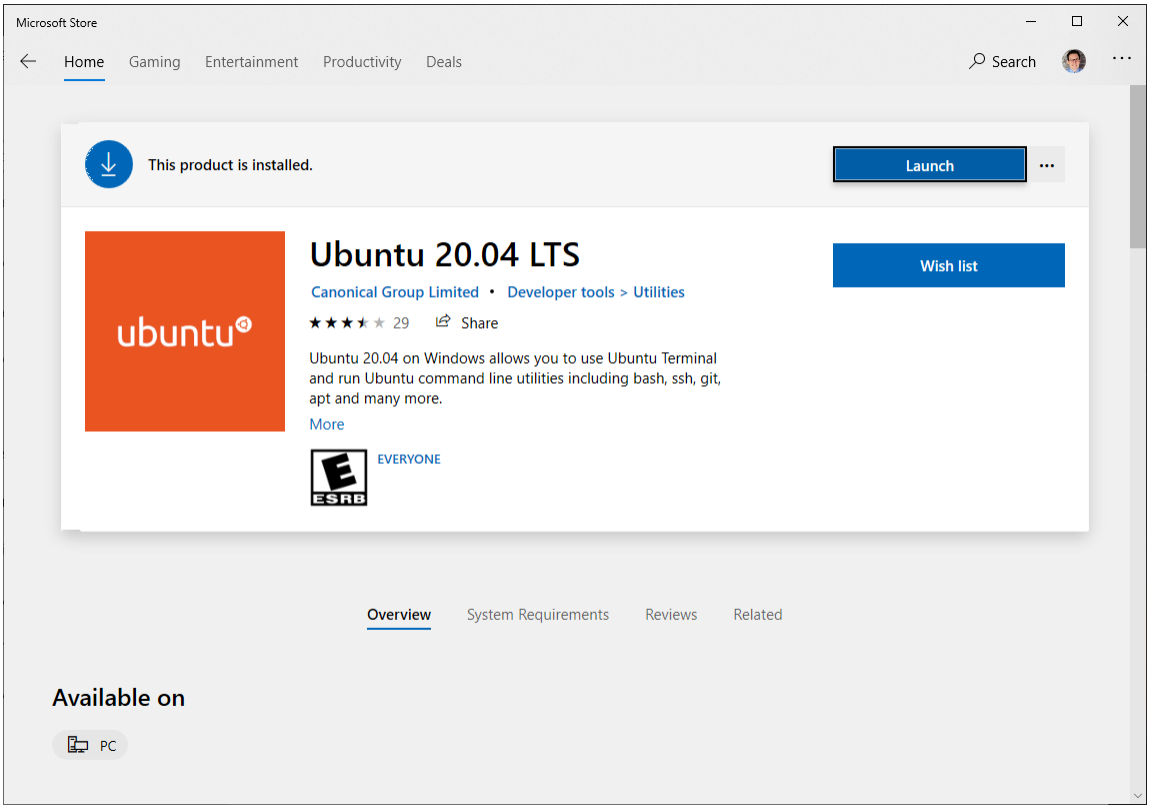
```
wsl --set-default-version 2
```

This command ensures that all future Linux installations will use WSL version 2.



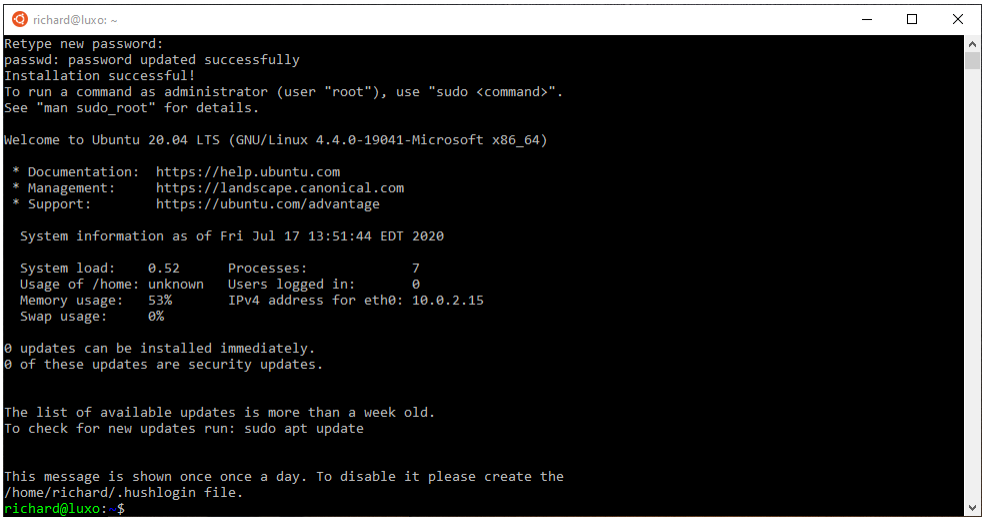
Install a Linux Distribution

Next, we need to install a Linux distribution via the Microsoft Store. Install Ubuntu 20.04 LTS. Once installed, you can launch it like any other application from the Start Menu.



Initial Setup

The first time you launch the Ubuntu Linux console, it will prompt you for a UNIX username and password. You will need this password to perform sudo commands later. Once completed, your Linux shell is ready for use. All your actions and commands will run as the Linux user you specified.

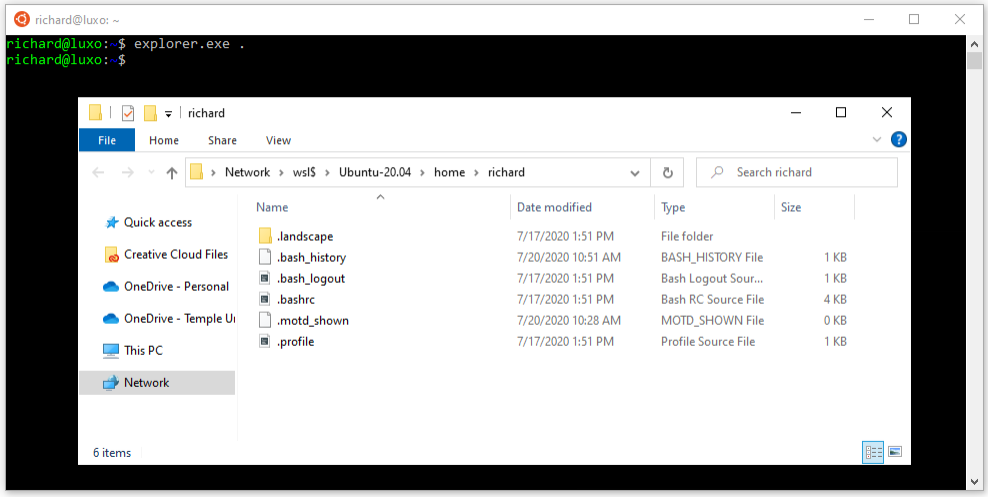


Windows Explorer / WSL integration

Your Linux installation will have its own Linux filesystem, which contains the Ubuntu files. Your Linux user will have a regular Linux home directory in /home/<USERNAME>. This directory is different from your Windows User directory. Windows and Linux filesystems are connected through WSL.

All hard drives in Windows are accessible in the /mnt directory in Linux. E.g., WSL maps the C hard drive to the /mnt/c directory. That means you can access your Windows User directory in /mnt/c/Users/<WINDOWS_USERNAME>.

The Windows Explorer can also access the Linux filesystem. To illustrate this integration, open an Ubuntu console and navigate to a directory of your choice. To view this location in Windows Explorer, use the explorer.exe . command (do not forget the final dot!).



Compiling LAMMPS

You now have a fully functioning Ubuntu installation and can follow most guides to install LAMMPS on a Linux system. Here are some of the essential steps to follow:

Install prerequisite packages

Before we can begin, we need to download the necessary compiler toolchain and libraries to compile LAMMPS. In our Ubuntu-based Linux installation, we will use the apt package manager to install additional packages.

First, upgrade all existing packages using apt update and apt upgrade.

```
sudo apt update
sudo apt upgrade -y
```

Next, install the following packages with apt install:

```
sudo apt install -y cmake build-essential ccache gfortran openmpi-bin libopenmpi-dev \
                    libfftw3-dev libjpeg-dev libpng-dev python3-dev python3-pip \
                    python3-virtualenv libblas-dev liblapack-dev libhdf5-serial-dev \
                    hdf5-tools
```

Download LAMMPS

Obtain a copy of the LAMMPS source code and go into it using the cd command.

Option 1: Download a LAMMPS tarball using wget

```
wget https://github.com/lammps/lammps/archive/stable_3Mar2020.tar.gz
tar xvzf stable_3Mar2020.tar.gz
cd lammps
```

Option 2: Download a LAMMPS development version from GitHub

```
git clone --depth=1 https://github.com/lammps/lammps.git
cd lammps
```

Configure and Compile LAMMPS with CMake

A beginner-friendly way to compile LAMMPS is to use CMake. Create a build directory to compile LAMMPS and move into it. This directory will store the build configuration and any binaries generated during compilation.

```
mkdir build
cd build
```

There are countless ways to compile LAMMPS. It is beyond the scope of this tutorial. If you want to find out more about what can be enabled, please consult the extensive documentation.

To compile a minimal version of LAMMPS, we're going to use a preset. Presets are a way to specify a collection of CMake options using a file.

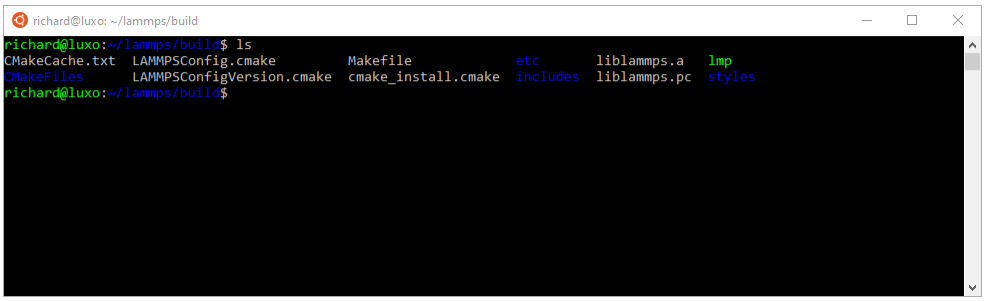
```
cmake ../cmake/presets/basic.cmake ../cmake
```

This command configures the build and generates the necessary Makefiles. To compile the binary, run the make command.

```
make -j 4
```

The -j option specifies how many parallel processes will perform the compilation. This option can significantly speed up compilation times. Use a number that corresponds to the number of processors in your system.

After the compilation completes successfully, you will have an executable called lmp in the build directory.



Please take note of the absolute path of your build directory. You will need to know the location to execute the LAMMPS binary later.

One way of getting the absolute path of the current directory is through the $PWD variable:

```
# prints out the current value of the PWD variable
echo $PWD
```

Let us save this value in a temporary variable LAMMPS_BUILD_DIR for future use:

```
LAMMPS_BUILD_DIR=$PWD
```

The full path of the LAMMPS binary then is $LAMMPS_BUILD_DIR/lmp.

Running an example script

Now that we have a LAMMPS binary, we will run a script from the examples folder.

Switch into the examples/melt folder:

```
cd ../examples/melt
```

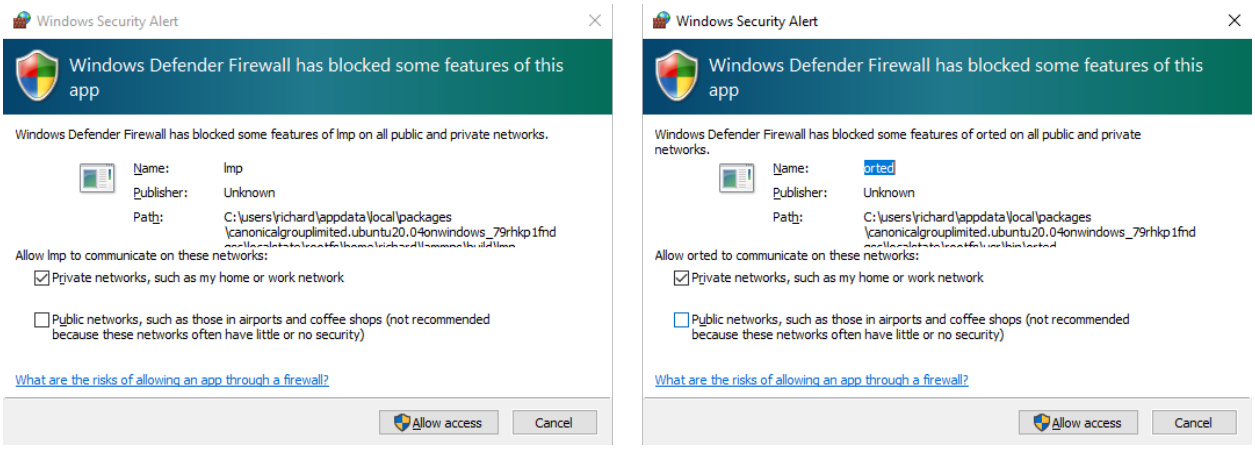
To run this example in serial, use the following command line:

```
$LAMMPS_BUILD_DIR/lmp -in in.melt
```

To run the same script in parallel using MPI with 4 processes, do the following:

```
mpirun -np 4 $LAMMPS_BUILD_DIR/lmp -in in.melt
```

If you run LAMMPS for the first time, the Windows Firewall might prompt you to confirm access. LAMMPS is accessing the network stack to enable parallel computation. Allow the access.



In either serial or MPI case, LAMMPS executes and will output something similar to this:

```
LAMMPS (30 Jun 2020)
...
...
...
```

(continues on next page)

(continued from previous page)

```
Total # of neighbors = 151513
Ave neighs/atom = 37.878250
Neighbor list builds = 12
Dangerous builds not checked
Total wall time: 0:00:00
```

Congratulations! You've successfully compiled and executed LAMMPS on WSL!

Final steps

It is cumbersome to always specify the path of your LAMMPS binary. You can avoid this by adding the absolute path of your build directory to your PATH environment variable.

```
export PATH=$LAMMPS_BUILD_DIR:$PATH
```

You can then run LAMMPS input scripts like this:

```
lmp -in in.melt
```

or

```
mpirun -np 4 lmp -in in.melt
```

Note

The value of this PATH variable will disappear once you close your console window. To persist this setting edit the $HOME/.bashrc file using your favorite text editor and add this line:

```
export PATH=/full/path/to/your/lammps/build:$PATH
```

Example: If the LAMMPS executable *lmp* has the following absolute path:

```
/home/<USERNAME>/lammps/build/lmp
```

the PATH variable should be:

```
export PATH=/home/<USERNAME>/lammps/build:$PATH
```

Once set up, all your Ubuntu consoles will always have access to your lmp binary without having to specify its location.

Conclusion

I hope this gives you good overview on how to start compiling and running LAMMPS on Windows. WSL makes preparing and running scripts on Windows a much better experience.

If you are completely new to Linux, I highly recommend investing some time in studying Linux online tutorials. E.g., tutorials about Bash Shell and Basic Unix commands (e.g., Linux Journey). Acquiring these skills will make you much more productive in this environment.

See also

- Windows Subsystem for Linux Documentation